Contents

1 Get the Code ................................................. 2
2 Find the Right Place ........................................... 2
3 Understand the Existing Code ......................... 3
  3.1 Meta Tags ................................................. 3
  3.2 A.Structured .............................................. 3
  3.3 Monadic Code ............................................. 5
  3.4 CompState, CSM and CSMR ............................. 7
  3.5 Warn and Die ............................................ 7
  3.6 The PassM monad ........................................ 7
  3.7 Generics ................................................... 8
     3.7.1 Traversal Strategies ............................... 8
     3.7.2 A Typical Traversal ............................... 8
     3.7.3 How the Typical Traversal Works ............... 9
4 Add Your Own Code ........................................ 9
  4.1 Conventions .............................................. 9
  4.2 Be Lazy (it’s the Haskell way) ...................... 10
     4.2.1 Write Only What You Need ................. 11
     4.2.2 Re-Use Everyone Else’s Code ............. 11
     4.2.3 Clean Up Code ................................ 11
5 Test Your Code ............................................. 11
  5.1 Running the Tests ..................................... 12
  5.2 Test Frameworks ....................................... 12
  5.3 HUnit .................................................... 12
     5.3.1 Labelling Tests ................................. 12
     5.3.2 Other items ..................................... 13
  5.4 Wiring In The Tests ................................... 13
6 Comment Your Code ..................................... 14
1 Get the Code

All details about checking out the code, committing your changes, the mailing list, and how to keep track of the repository, currently reside on this page: [http://www.cs.kent.ac.uk/research/groups/sys/wiki/Tock](http://www.cs.kent.ac.uk/research/groups/sys/wiki/Tock).

Tock is held in a darcs repository. Darcs is broadly similar to CVS/SVN. You can find more details on the Darcs website ([http://darcs.net/](http://darcs.net/)) or in the manual ([http://darcs.net/manual/](http://darcs.net/manual/)) but the following few commands will usually suffice:

- `darcs whatsnew` – shows what changes have been made but not committed (like: svn diff)
- `darcs record` – records a patch (like: svn commit)
- `darcs pull` – pulls changes from the parent repository (like: svn update)
- `darcs send` – sends changes (against the parent repository) via email

On Tock we favour utilising the strengths of Darcs, and making each patch independent and small. Obviously this is up to the judgement of the programmer, but one-line patches to fix a bug are perfectly acceptable. Patches that change over a hundred lines are to be avoided unless it really is a big change.

One other note: if at all possible, record separate patches for the tests and implementation that passes those tests. This makes it easier for someone else in future to check that the tests passed and failed appropriately before the implementation was changed/added.

2 Find the Right Place

Tock’s modules are currently arranged into four directories. They are:

1. “common” – All modules that are used by many/most parts of the program.
2. “frontends” – All modules relating to lexing, parsing, preprocessing occam and Rain, as well as early steps in compilation like resolving names, checking types, etc.
3. “transformations” – All modules relating to transforming the tree either for simplicity, efficiency or simply to remove elements (e.g. parallel assignment) not supported by the backends.
4. “checks” – All modules relating to usage checks and other compiler checks.
5. “backends” – All modules related to the final step of turning AST into actual code.

The separation is by no means hard-and-fast, or perfect, but it’s better than nothing. Tests are in the same directory as the thing they test.

The directories should provide a quick idea of where to find what you are interested in. Data types and functions common to the whole compiler, such as the AST definition and type helper functions are in “common”. The other parts of the compiler are in the obvious order (frontends, transformations, backends).

The `Main` module in the main tock directory is the actual module for the tock executable. It merely deals with the command-line options and joins together the various passes according to the options given.

If you want to add a new frontend or backend, then add a new command-line option (look in the modules `Main` and `CompState`) for it and handle the option accordingly in the `Main` module. To add a new pass, add it to the appropriate place in the list in the `PassList` module, or add it to the appropriate pass-item already listed there (e.g. `simplifyTypes`).
3 Understand the Existing Code

There are (unfortunately, but realistically) several barriers to understanding the existing Tock code:

1. It’s written in Haskell.
2. It makes heavy use of monads.
3. It uses generics.
4. You need to understand the AST well.
5. The C and C++ backends are quite dense and tricky.

The last point is somewhat unavoidable, without an inspired re-write. Knowledge of occam will help a lot with understanding the AST, except perhaps for the Structured item (see below). Haskell knowledge can be solved with a book or two (or other web resources); monads and generics are each covered in a section below.

3.1 Meta Tags

Scattered throughout the definition of the AST you will find many Meta items. Meta is technically for any annotations about that part of the program. Currently, Meta is only used for source position. It is included in every appropriate AST structure as the first item after the data constructor name. This allows use to easily use a (generic-based) findMeta function for finding the first meta-tag in an item.

3.2 A.Structured

The main item in the AST that I (Neil) found confusing at first was the Structured item. Thankfully, I’ve recently changed Structured to be parameterised (which helped), but I’ve left this explanation in anyway, it case it helps. Note that the AST module is always imported as A, hence all the ‘A.’ prefixes on the AST items discussed here.

Structured is the body of most occam constructs, such as SEQ, PAR, ALT, CASE. Because occam allows the inter-mingling of processes and declarations, and also replication on most of its constructs (SEQ, PAR, ALT, IF) Structured eliminates redundancy by grouping this together. SEQ and PAR have a A.Structured A.Process as their ‘body’, whereas, for example, ALT has a A.Structured A.Alternative.

Here is the definition of the Structured item:

```haskell
data Structured a =
  Rep Meta Replicator (Structured a)
  | Spec Meta Specification (Structured a)
  | ProcThen Meta Process (Structured a)
  | Only Meta a
  | Several Meta [Structured a]
```

So for example, given this occam pseudo-code:

```occam
SEQ
  proc1
  proc2
```

Here is how it would be represented in the AST (Taking `proc1` and `proc2` to be of type Process, and using `m` for all meta-tags):
A.\texttt{Seq} m
\begin{verbatim}
  (A.\texttt{Several} m
   | A.\texttt{Only} m proc1
   , A.\texttt{Only} m proc2
  )
\end{verbatim}

You can see the combination of \texttt{A.Seq} with \texttt{A.Several} and \texttt{A.Only} to nest the processes. Here’s another example of some occam and corresponding Haskell:

\begin{verbatim}
SEQ
  proc1
  \texttt{PAR}
  proc2
  proc3
\end{verbatim}
\begin{verbatim}
A.\texttt{Seq} m
\begin{verbatim}
  (A.\texttt{Several} m
   | A.\texttt{Only} m proc1
   , A.\texttt{Only} m
     (A.\texttt{Par} m A.\texttt{PlainPar}
       (A.\texttt{Several} m
        | A.\texttt{Only} m proc2
        , A.\texttt{Only} m proc3
       )
     )
  )
\end{verbatim}
\end{verbatim}

Which no doubt looks quite nasty! But things work differently if you nest two blocks of the same type, mainly because of the associativity of the various blocks in occam. Consider these two SEQ blocks:

\begin{verbatim}
SEQ
  proc1
  SEQ
  proc2
  proc3
\end{verbatim}
\begin{verbatim}
A.\texttt{Seq} m
\begin{verbatim}
  (A.\texttt{Several} m
   | A.\texttt{Only} m proc1
   , (A.\texttt{Several} m
       | A.\texttt{Only} m proc2
       , A.\texttt{Only} m proc3
      )
  )
\end{verbatim}
\end{verbatim}

You can see that instead of creating a second \texttt{A.Seq} inside the \texttt{A.Several}, it has instead simply nested another \texttt{A.Several}. In fact, we could later flatten the two nested \texttt{A.Severals} into one if we wanted; in all \texttt{A.Structured} items, this should always be possible to do (without altering the behaviour of the program).

Here is another example:

\begin{verbatim}
PAR
  proc1
  \texttt{PAR} i = 0 \texttt{FOR} 10
  proc2
\end{verbatim}
A.Par m A.PlainPar
   (A.Several m
    |A.Only m proc1
    .A.Rep m rep (A.Only m proc2)
   )
)

I have used ‘rep’ as a short-hand for the replicator (which is not the focus here). Hopefully it is now clear how A.Structured is used as a body for things. There is only one more aspect to explain; specifications.

SEQ
   proc1
   INT x:
   proc2

According to occam scoping rules, x is in scope for proc2. This is represented in the AST as follows:

A.Seq m
   (A.Several m
    |A.Only m proc1
    .A.Specification m spec (A.Only m proc2)
   )
)

Where spec is shorthand for the full specification of x. The specification (third argument of A.Specification) is in scope for the whole of the body (the fourth argument). Multiple specifications lead to nested A.Specifications:

SEQ
   proc1
   INT x:
   INT16 y:
   proc2

A.Seq m
   (A.Several m
    |A.Only m proc1
    .A.Specification m specX
     (A.Specification m specY (A.Only m proc2))
   )
)

The A.ProcThen item is used in situations where we need to perform a process in the middle of a A.Structured a block (where a is not A.Process). For example, VALOF uses this to execute a process then return a result. Some specifications with initialisation may need to be transformed into a specification, with initialisation code (a process) followed by some more of the A.Structured item.

3.3 Monadic Code

Monads are generally considered a tricky topic. I do not plan to cover their full scope or try to comprehensively explain them from scratch here. I attempt a brief summary, but mainly I just thought it might be helpful to provide some comments on how real monadic code in Tock works. For a full explanation of monads, try looking for web resources or asking a Tock developer directly.

We will look at a real example from the current Tock codebase. This is the doProcess function inside the removeParAssign function in the SimplifyProcs module. Its purpose is to turn parallel assignments into multiple (sequential) single assignments.
doProcess :: A.

doProcess (A.Assign m vs@(_:_)) (A.ExpressionList _ es)
        = do ts <- mapM typeOfVariable vs
                spec <- sequence
                [makeNonceVariable "assign_temp" m t A.Original | t <- ts]
                let temps = [A.Variable m n | A.Specification _ n _ <- spec]
                let first = [A.Assign m [v] (A.ExpressionList m [e]) | (v, e) <- zip temps es]
                let second = [A.Assign m [v] (A.ExpressionList m [A.ExprVariable m v'])
                                      | (v, v') <- zip vs temps]
                return $ A.Seq m $ fold1 (| as spec -> A.Spec m spec m)
                                (A.Several m (map (A.Only m) (first ++ second))) spec

doProcess p = doGeneric p

The type of this function is to take a A.Process (the A. is simply because it’s an AST fragment; the AST module is always imported as A) and give back a monadic action in the PassM monad that will yield a Process.

Here is a description of what the code does. The pattern match matches any assignment that has two or more items on the LHS (vs@(_:_)) matches any list that has at least two elements – the final match can be the empty list). The first line gets the type of the variables on the LHS and stores this list of types in ts. specs is a list of specifications of nonce variables to be assignment temporaries, one for each type in the type list (ts). temps is the list of Variable items. first is the list of assignments from the RHS of the original assignment to the temporaries. second is the list of assignments from the temporaries to the original LHS. Finally, we use the foldl function to nest the specifications, and make a sequential list of the assignments (first then second).

The function has a standard (i.e. unrelated to monads) Haskell pattern-match as its header. The direct value of the function is a do block; a do block is technically of type ‘monadic action’.

The indentation rule of the do block is fairly simple; each item in the do block should have the same level of indentation, and finishes when something is found with less indentation (standard Haskell indentation rules – the “offside rule”).

The first line (ts <- mapM typeOfVariable vs) is already somewhat complex. Firstly, the type of typeOfVariable is:

typeOfVariable :: (CSM m, Die m) => A.Variable -> m A.Type

CSM and Die are two typeclasses to which PassM belongs. So in our case, m can be PassM. Therefore the effective type for us is A.Variable -> PassM A.Type. mapM is a monadic version of map:

mapM :: Monad m => (a -> m b) -> [a] -> m [b]

It basically takes a monadic function, and applies it to each element of the given list, returning a monadic action that will yield the mapped elements.

So in our case, mapM typeOfVariable will have type [A.Variable] -> PassM [A.Type]. The argument is then vs. The notation ts <- means that the value yielded by the monadic action is labelled as ts. You may think of it as the monadic version of the let notation in normal Haskell. Note that ts is of type [A.Type]; it is not monadic. The action is actually performed by this statement, and the result is put into ts.

The second line is again interesting. The list is a standard Haskell list comprehension. The type of makeNonceVariable is:

makeNonceVariable :: CSM m => String -> Meta -> A.Type ->
                    A.NameType -> A.AbbrevMode -> m A.Specification

The list comprehension is therefore of type [PassM A.Specification]; that is, it is a list of monadic actions, each of which will yield a A.Specification. This is then given to the sequence function which has this type:

sequence :: Monad m => [m a] -> m [a]
In other words, `sequence` takes a list of monadic actions, performs each of them (in sequence!) and returns the resulting list of elements (inside the monad, of course). So `sequence` performs all our actions and gives us back a list of `A.Specifications`. This list is then labelled as `specs`.

The next three lines in our code fragment begin with `let`. These are all non-monadic lines, and are just plain Haskell. Note that there is a slight difference between the `let` notation inside a `do` block to the normal `let..in` notation; there is no `in` keyword in the version of `let` in the `do` block. This is a technicality, but one that can trip you up; if you add the `in` keyword you’ll get a not-very-helpful parser error.

Finally, the last line of our function features `return`. `return` is a standard monadic function that is used very frequently. Its type is:

```haskell
return :: Monad m => a -> m a
```

All it does is turn a plain (non-monadic) value into a simple monadic action that yields the value. More simply, it lifts the value into the monad. In our function we have a plain value, but we need to lift it inside the monad to satisfy the types. This is quite complex in relation to the types, so is perhaps best viewed as an analogue to C or Java’s `return` statement.

### 3.4 CompState, CSM and CSMR

Our compiler state (which is relatively small, considering it is for the whole compiler) is a data-type named `CompState`. If you look at the definition you will see that it is currently divided into four parts:

1. The options set on the command-line. This includes things such as warning level, and choice of backend. These should not change during compilation.
2. Items recorded by the pre-processor.
3. The symbol table and similar structures primarily generated by early passes (but definitely added to later on).
4. Some state used by various passes.

This may seem like a slight mish-mash but separating it out into separate state (and separate monads) is likely more trouble than it’s worth.

There are two monads associated with `CompState`. `CSM` is shorthand for a state monad with `CompState` as the state. If however you only need read-access to the state for your particular function (e.g. you only need it to check what command-line options were used), then use `CSMR` instead, which only provides read-only access to the state. This makes the type signature a little clearer as to whether you may or may not modify the state in that function.

### 3.5 Warn and Die

We have one type-class (of monads) for each of warnings and errors. The `Die` type-class should be used for fatal errors, whereas the `Warn` type-class is used for recording warnings. Both use an optional `Meta` item for source position, and a `String` for an error message. Wherever possible (and it usually should be) provide a `Meta` item with a source position. From a user’s perspective, an error with a source position is much more useful than one without!

### 3.6 The PassM monad

It has been mentioned above that the PassM monad is the most common monad. Here is its actual type:

```haskell
type PassM = ErrorT ErrorReport (StateT CompState (WriterT [WarningReport] IO))
```
The `PassM` monad is (currently!) a stack of four monads: an error monad, a state monad, a writer monad and the `IO` monad. The error monad allows for exception-like mechanisms. In Tock, we throw an error whenever we can proceed no further with the compilation. Examples include parser errors, type errors and parallel safety problems. The state in question is the CompState type (see the module of the same name), which holds things like the name-type dictionary (aka symbol table). The writer monad keeps track of all the warnings encountered, ready to print them all out at the end of compilation (but allows us to be flexible and ignore them if, for example, we only want to display fatal errors when the compilation fails, not the warnings as well). The `IO` monad is included for various reasons, such as being able to read in files.

3.7 Generics

Generics are a technique used to easily query or modify specifically-typed parts of a big data structure without writing tree traversal code manually. So for example, we may want to apply a certain function to all the `A.Expression` in our AST without having to write code to traverse every other type in the tree looking for expressions.

We use the `Data.Generics` module of GHC to do our generics (also known as the Scrap Your Boilerplate or SYB approach). The deep-down mechanics are very confusing. How to use it is simpler, and is explained in the three excellent SYB papers (found on the web here: `http://www.cs.vu.nl/boilerplate/#papers`). Things are made slightly tricky again because we usually perform custom traversals.

The `everywhere` (or `everywhereM`) function(s) described in the SYB papers traverse an entire tree structure looking to apply your transformation. Unfortunately this includes examining each character of each string in every `Meta` tag, which makes things (unacceptably) slow. Therefore Adam implemented a custom traversal pattern. Since you will almost always want this traversal, you do not have to worry too much about the internals (which are described in section 3.7.1), just about how to use it (see section 3.7.2).

3.7.1 Traversal Strategies

The `Data.Generics` library provides a `gmapM` function which maps a monadic transformation over all sub-elements of a term. So for example, `gmapM return (A.Specification m spec innerStr)` will apply the return function (in monads, this is the identity transformation) over the items `m`, `spec` and `innerStr` in turn. Note that there is no recursion into `innerStr`.

To provide recursion easily, you can use the `everywhereM` function. This is defined as follows:

```
everywhereM f x = do x' <- gmapM (everywhereM f) x
             f x'
```

It applies itself over all sub-elements (which will recurse all the way through the tree) then applies the modifier function to the result. As described above, however, it can be quite inefficient.

3.7.2 A Typical Traversal

Here’s an example; the wrapper from our previous code example:

```haskell
removeParAssign :: Data t => t -> PassM t
removeParAssign = doGeneric 'extM' doProcess
  where
    doGeneric :: Data t => t -> PassM t
    doGeneric = makeGeneric removeParAssign

  doProcess :: A.Process -> PassM A.Process
  doProcess (A.Assign m vs@(_:_:_:)) (A.ExpressionList _ es)) = ...
  doProcess p = doGeneric p
```
You can follow this template for any new passes you write. All you need to customise is the name of the pass (of course), and change the type and name of the doProcess function if you want to transform something other than a A.Process. For example:

twiddleExpressions :: Data t => t -> PassM t
twiddleExpressions = doGeneric 'extM' doExpression

where
  doGeneric :: Data t => t -> PassM t
  doGeneric = makeGeneric twiddleExpressions

  doExpression :: A.Expression -> PassM A.Expression
  doExpression (...) = ...  -- First pattern match
  doExpression (...) = ...  -- Second pattern match
  doExpression p = doGeneric p

Note that you must include the last case for your doProcess/doExpression function; otherwise you will get an error if your pattern-matches are not exhaustive (which they rarely will be). The net effect is to apply doExpression to all expressions in the given AST, in an efficient manner. In other words, it’s a compiler pass that operates on the expressions in the tree.

3.7.3 How the Typical Traversal Works

The makeGeneric function is defined as follows:

makeGeneric top = (gmapM top)
  'extM' (return :: String -> PassM String)
  'extM' (return :: Meta -> PassM Meta)

So it applies the given function using gmapM, except for Strings and Meta tags which it skips. We apply this to our top-level function to get our doGeneric function that handles all the data items we are not interested in. In our top-level we extend this with the specific cases that we are interested in; in this case, expressions.

The last thing required is to apply doGeneric to all the expressions that we are not interested in. Note that we do not apply the top-level function (twiddleExpressions). If we did, we would get infinite recursion between doExpression and twiddleExpressions. Instead we apply doGeneric, which has no specific case for expressions, and will therefore recurse through the expression item down to the next sub-items.

4 Add Your Own Code

4.1 Conventions

Tock, as a project, does not have any particular coding conventions. The code is littered with slightly curious code indentation, one or two letter variable names, incredibly long expressions stretching long and wide, various bits of uncommented code, and the use of many similar but different language features (e.g. if/then/else and pattern guards, map and list comprehensions), which may be blamed in varying measures on the current developers!

Which is not to say the code is bad, just that there is not tight control on coding style. In general:

1. Use your common sense
2. Vaguely follow the style of the existing code
3. Favour readability and clarity over conciseness and cleverness
4. If you optimise, optimise only in terms of algorithms (e.g. \(O(N \log N)\) over \(O(N^2)\)) but never look for small savings. Besides the effort being wasted, it would be very hard in Haskell to judge which of several pieces of code would be faster. The compiler does not have to be blindingly fast, but it does need to be maintainable.

5. If a function foo is specifically needed by only the function bar, place foo inside the where clause of bar (unless this is particularly untenable). This keeps the code neater, and foo can always be moved to the top-level later if necessary.

6. Always give type signatures for functions at the top-level of the file (i.e. those not inside a where clause). Additionally, try to provide type signatures for every function (i.e. anything of kind \(\ast \to \ast\)) in a where clause. Providing types for values in where clauses is also never a bad thing.

7. Try not to leave warnings in the code. We have compiler options turned on to generate various warnings. Defaulting-to-type warnings can be solved by inserting a type signature, and unused binding warnings can be solved by removing the unused function, unless you know the lack of use is temporary.

8. Never allow any possibility of a non-graceful run-time error. For example, do not use head, which can fail directly with a non-helpful error message. Instead, use a case statement (with a pattern-match), and in the case where the list is empty, use the die/dieP/dieInternal functions to provide a more helpful error message (such as “list of types was not expected to be empty in function foo”). This is for practical reasons; if we used head everywhere in the code, then when the program failed with “head: empty list” it would be very hard to work out exactly which instance of head had given the error. Similarly, try to ensure that you either always match all possible cases in pattern-matching, or you provide a default case that then gives an error message. Although this is not quite as crucial, because at least the error message for a failed pattern match gives the relevant line numbers. That helps developers, but not users!

9. In lists where order is unimportant (such as test lists, or module import lists), maintain alphabetical order (to make it easier to find items in a long list). Your editor may be able to help with this.

10. Never use tabs.

11. Put spaces around operators (except colons in pattern matches).

12. When writing out lists or tuples on one line, try to make sure there is a space after the comma (clearer, Adam’s preference but Neil’s bad habit).

13. When writing out lists on several lines, put the commas between items at the beginning of each line, not at the end (can make patches clearer by not disrupting surrounding lines).

As for other patterns of working: use the Tock mailing list (tock-discuss) for any questions you may have. All questions welcome, simple or complex, and asking there will save time, trouble, and should allow us to improve documentation such as this guide. It is especially worth asking if you want to revamp an existing section of code; other people may know of a reason why this is not wise, or may suggest a good way to go about it.

4.2 Be Lazy (it’s the Haskell way)

Try and make your life easy by coding as little as possible.
4.2.1 Write Only What You Need

Tock has effectively been built using similar ideas to extreme programming. Test everything, don’t be afraid to refactor, and don’t over-engineer things. Write the minimum you need, and if you find you need more later, add it and possibly refactor. I find refactoring in Haskell – even big changes that affect most of Tock – to be quite easy. However, if you are planning to make a change that will impact a lot of code, it’s best to discuss/notify via the mailing list first!

4.2.2 Re-Use Everyone Else’s Code

Check the Utils module (and the TestUtils module when testing) as well as other modules (such as Types) to see if there is an existing function that does what you need. Anything that seems likely to have been needed before (such as getting the type of a variable) is probably in those modules. Similarly, if you ever find yourself writing a general utility function more than once, you should probably look to put it the Utils module. The guideline with the Utils module is that it should never import any other Tock modules.

The Utils (and to a lesser extent TestUtils) module are intended to contain functions that could have come straight out of the standard library. The standard library is also another place to look for useful functions. The URL for the latest version is: http://haskell.org/ghc/docs/6.6/html/libraries/. Make sure you always refer to the documentation for the lowest compiler version we support (currently 6.6) to avoid accidentally using a function that is only available in a later version.

Particularly useful modules are:

- **Data.List** – Contains various helper functions for dealing with lists, including map, foldl, zip, sum, and many others\(^1\). Probably the most useful module.
- **Control.Monad** – Contains all the general monadic helper functions, such as mapM, foldM, sequence. If you ever find yourself struggling to manipulate monadic types, there may be something to help you in here.
- **System.IO** – Contains all the functions for printing to the screen, reading from files, etc.

Other useful modules are **Data.Maybe, Data.Generics, Data.Map, Data.Set** and **Test.HUnit**.

4.2.3 Clean Up Code

Don’t be afraid to re-write someone else’s code, if you think it could be made clearer or simpler. If it works out (i.e. passes the tests that should exist for the code), great. If it doesn’t pan out for some reason, add a comment as to why re-writing it doesn’t work so that the next developer doesn’t attempt the same thing.

5 Test Your Code

Whether you write your tests before, at the same time as, or after the real code is not a major issue, so long as the test gets written. Personally I (Neil) favour writing them somewhat simultaneously; the test usually gives an idea of how to implement the function, but writing the function can often make you realise that your expected test output does not actually match how the function should work! So interleaving writing the tests and the function seems to be helpful.

Most of Tock is now unit-tested, with the ultimate aim of having everything in it tested. So when you add new functionality, it would be good if you also add the corresponding tests, to help towards this aim. The idea is that running `make & & ./tocktest` should give no errors (except in whatever you are currently working on). This is broadly true; at the time of writing there are around 1800 tests, with 7 errors, 5 of which are related to what I’m currently working on.

\(^1\)A lot of these are technically in the standard Prelude, but it’s easy to find all the documentation in one place in the **Data.List** module, as well as several useful functions not in the Prelude.
5.1 Running the Tests

When you run `make`, the build system will build both the `tock` and `tocktest` executables. The latter is of course the test-suite. Currently it runs tests from all the frameworks (see next section) except the cgtests. There are several options to `tocktest`:

- `--qc={off,low,medium,high}` – Sets the level of QuickCheck testing. I would suggest usually using ‘low’ (it will be faster) but occasionally running ‘medium’ or ‘high’ as a sanity check.
- `--plain` – Outputs plain text rather than playing with terminal deletion. Useful for when you want plain-text output (e.g. to redirect to a file).

5.2 Test Frameworks

We effectively use four test frameworks in Tock:

1. QuickCheck; a framework used to generate random input data for tests, then test properties of its output.
2. HUnit; a simple framework for providing standard lists of assertions.
3. cgtests; the standard occam test-suite. Useful for providing a full-system test for anything that gets used in the occam side of things. Currently, not all the cgtests pass, but there is a list on the Trac wiki for Tock of all the tests currently expected to pass/fail (current URL: http://projects.cs.kent.ac.uk/projects/tock/trac/wiki/CgtestOutput).
4. Automatic test harness. Currently very simplistic, but essentially a couple of helper functions in the `TestHarness` module allow you to easily provide an external file of occam code tests and to specify whether this code should provoke an error from the compiler (at least, everything before the final code-generation step).

5.3 HUnit

For most of your tests, HUnit will be the most appropriate. The main HUnit data type is as follows:

```haskell
data Test
    = TestCase Assertion
    | TestList [Test]
    | TestLabel String Test
```

This allows arbitrary (nested) lists of Tests to be built up. Since each Assertion already has a label, I do not favour labelling each TestCase, but labelling each TestList is not a bad idea. It makes your test easier to locate if/when it fails.

Each Assertion (actually type: IO ()) comes from functions like `assertEqual`. This function is of type:

```haskell(assertEqual :: (Eq a, Show a) => String -> a -> a -> Assertion)
```

5.3.1 Labelling Tests

The first argument is the label. I usually use this as a substitute for a test name; you'll commonly see it being simply the name of the test function (e.g. `testGenOutput`) concatenated with a number. Ideally, the label would be a wonderfully descriptive label of the test, but realistically you end up writing so many trivial test-cases that you will probably also lapse into the same habit as me. I usually combine this with writing a helper function (often called `test`) in the `where`
clause to help out with that particular family of test (often automatically feeding input into the function being tested).

You will also see that I always arbitrarily number the test-cases in a manner similar to ye olde BASIC line numbers (often taking random leaps forward to leave gaps for later). This probably appears crazy but I do it for two reasons:

1. As mentioned above, finding descriptive names gets silly when you have a decent number of tests, so numbering is preferred to text.

2. I deliberately avoid ‘auto-numbering’ tests (zipping them with [0..] would be quite easy, of course). This would leave the numbers fragile; if a change to the test lists added a test mid-list then all the numbers would be altered. With my method, because the test numbers are never changed, if someone has “testGenOutput 203” fail on them, it is easy to find, and will be uniquely identifiable, even if the test-list has changed since they checked out their version. Also, you can perform a text-search for the number of the test, rather than having to count through items in an automatically numbered list.

5.3.2 Other items

Furthermore, the next two arguments of the `assertEqual` function are the expected test output and the actual test output in that order. So you might write something like:

```haskell
assertEqual "concatString Test 0" "foobar" (concatString "foo" "bar")
```

This is of type `Assertion`; prefix it with `TestCase $` to make it a `Test`. There are many helper functions related to building up input for tests (particularly AST fragments) and for performing customised assertions (especially for testing passes in the `PassM` monad) in the `TestUtil` module; you should always look there to see if it has a helper function that would be useful to you. Similarly, if you think any of your own test helper functions could be useful in other places, add them to the `TestUtil` module.

5.4 Wiring In The Tests

If you are adding to an existing portion of Tock, then you should add your tests alongside the existing tests. Most of Tock is tested; if you add to a bit that is not tested then please consider writing tests for the old code too.

When writing a new chunk of functionality you will want to create a new module for the tests. The general pattern is to put the tests for module `Foo` into module `FooTest`. You should then import the module you are testing, any other appropriate modules, and `Test.HUnit`.

By convention, you should provide one (and only one) of the following functions in your test module:

1. `tests :: Test` (remember that a `Test` can be a `TestList`)
2. `qcTests :: Test, [QuickCheckTest]`
3. `ioqcTests :: IO Test, [QuickCheckTest]`

You should provide an export list for your module that contains only this function. This is very useful, as it means that any tests you write in your test module but forget to call in its exported `tests` function will be flagged up by the compiler as unused.

Then add your new test module to the list in the `TestMain` module. Add the appropriate import declaration, and look at the very foot of the file for the `tests` list. Wrap your function according to its type, following the pattern of the other functions there. That is, you may need to add the `noqc` or `return` wrapper functions.
6 Comment Your Code

Like any real-world chunk of code, the documentation/comments on Tock vary from ‘polished’ to ‘absent’. Needless to say, the more documentation the better. If you do happen across someone’s code that puzzles you at first, then prod the developer who wrote it into adding some comments. If they wrote it and it’s not clear, it’s their fault! But much better is if we all document the code we write in the first place.

Naturally, standard rules apply; don’t just repeat in the comment what the code clearly says already. Document the purpose of the code, any interesting/odd methodology, tricks or problems.

Haddock is a documentation system for Haskell akin to Javadoc, Doxygen, etc. Starting to use it is very simple; instead of writing \texttt{\textdagger Some comment} before a function, write \texttt{\textdagger| Some comment} instead. It is not obvious in this style of mark-up but there is a space between the dashes and the pipe (it is required). See the Haddock documentation for other markup (latest version can be found here: \url{http://www.haskell.org/haddock/doc/html/index.html}). You can use \texttt{make haddock} to create the HTML documentation in the ‘doc’ directory.