Cheap Remarks about Concurrent Programs*

Michael Walker and Colin Runciman

University of York, UK
{msw504, colin.runciman}@york.ac.uk

Abstract. We present CoCo, the Concurrency Commentator, a tool that automatically generates algebraic specifications for concurrent Haskell functions operating on some shared state. Specifications are collections of properties about *refinement and equivalence of side-effects*, rather than equality of final results. The tool is based on testing, rather than static analysis or theorem proving. We demonstrate how use of CoCo can inform understanding of program behaviour, presenting case studies about concurrent stacks and semaphores.

1 Introduction

Concurrent programs with shared mutable state are notoriously difficult to get right, but concurrency is a very useful paradigm for structuring many real-world programs.

One way to deal with complexity is to write down our expectations, and verify them through formal reasoning or testing. These verified expectations give us a formal specification of our code. They can serve both as test cases and as parts of documentation.

Our previous work on Déjà Fu[13] allowed us to test such expectations, in the context of Haskell, but the programmer still had to devise them in the first place.

In this paper, we aim to aid programmers in this task. We propose a tool which, given a collection of functions operating on some shared mutable state and some *observation function*, can discover behavioural properties. A list of these properties is useful in a number of ways: (1) it can be an addition to existing documentation; (2) the programmer may gain new insights about their code; and (3) the absence of sensible properties, or the presence of unexpected ones, may indicate an error in the code.

As we use testing, our technique is potentially unsound. Each property discovered is a conjecture only supported by a finite number of test cases. So some reported properties may not hold in general. Despite this, we still believe our tool is useful.

We have implemented our technique as a Haskell program.

* Student project paper (primarily the work of the first author).

Contributions We present a method, based on systematic concurrency testing, to automatically discover properties of stateful operations. Furthermore, we demonstrate the viability of this method by providing an implementation of it in Haskell. We provide illustrative results of the tool from a number of examples.

Roadmap The rest of the paper is structured as follows: §2 introduces two key concerns in the implementation of our tool. §3 gives an illustrative example. §4 gives a detailed discussion of how the tool generates terms and discovers properties. §5 presents two case studies. §6 considers related work and how our contributions differ from it. §7 presents conclusions and discusses avenues for future work.

2 Two Key Concerns

Scheduling One reason why testing concurrent programs is difficult is the nondeterminism of scheduling: the same program with the same inputs may produce different results depending on the schedules chosen at execution time. Traditional testing techniques rely on the result of executing a test to be deterministic.

Systematic concurrency testing (SCT)[3, 6, 8, 9] is a way of tackling this problem of nondeterminism. It aims to test a variety of schedules, making use of local knowledge of the program to reduce the number of schedules needed to be confident of an accurate result.

We have previously developed Déjà Fu[13] as an SCT tool for Haskell, based on a typeclass-abstraction over the primitive concurrency operations.

Observational refinement One kind of expectation we might hope to verify by testing is the effect of some operation upon a shared state. This sort of comparison has been formalised as observational refinement[7]. Informally, an operation X refines an operation Y if all observable behaviours of X are also observable behaviours of Y.

So we shall require the programmer to supply an *observation function*, and compare observations rather than comparing states directly. By formulating their chosen observation function, the programmer can influence the properties discovered. We do not compare result values at all, only the state observations.

3 An Illustrative Example

Let us now show an example use of CoCo. We consider some concurrent shared variables in the Haskell libraries. An `MVar` is a mutable memory cell which may be *full* or *empty*: attempting to take from an empty `MVar` blocks until it is full, and attempting to put into a full `MVar` blocks until it is empty. Instead of the standard version of the functions from Haskell's Control.Concurrent library module, we instead use typeclass-generalised versions which Déjà Fu can test.

```
initialise :: Maybe Int -> Concurrency (MVar Concurrency Int)
initialise (Just i) = newMVar i
initialise Nothing  = newEmptyMVar

observe :: MVar Concurrency Int -> Concurrency (Maybe Int)
observe = tryReadMVar

interfere :: MVar Concurrency Int -> Maybe Int -> Concurrency ()
interfere v (Just i) = tryTakeMVar v >> void (tryPutMVar v i)
interfere v Nothing  = tryTakeMVar v
```

Fig. 1. Initialisation, observation, and interference functions.

When we use CoCo, we must provide the functions and values which may appear in expressions. We must also provide a way to initialise the state, an observation function, and an interference function. For the examples in this section, we shall use the functions shown in Figure 1. The `interfere` function lets us consider cases with and without concurrent interference. When two expressions appear to be observationally equivalent when there is no interference, we consider the interfering case, allowing us to distinguish atomic and nonatomic operations which happen to have the same effect.

Firstly we shall consider the three basic operations over `MVar`s: put, take, and read. To *put* is to block until the `MVar` is empty and then set its value. To *take* is to block until the `MVar` is full, remove its value, and return the value. To *read* is to *take*, but without also emptying the `MVar`. Allowing shared values of type `Int`, we have the following type signatures:

```
putMVar  :: MVar Concurrency Int -> Int -> Concurrency ()
takeMVar :: MVar Concurrency Int -> Concurrency Int
readMVar :: MVar Concurrency Int -> Concurrency Int
```

Here `Concurrency` is an implementation of the concurrency-typeclass. The `MVar` type is parameterised by the monad type; `MVar` is actually a *family* of types, with the concrete type determined by the monad. The return type of each function is of the form `Concurrency x`, meaning that the result of the function produces an `x` value and also has some side-effects in the concurrency execution context.

Given these functions, CoCo outputs exactly these properties:

```
 readMVar @  ===  readMVar @ >> readMVar @                    (1)
 readMVar @  -[-  takeMVar @ >>= \x -> putMVar @ x            (2)
 takeMVar @  ===  readMVar @ >> takeMVar @                    (3)
putMVar @ x  -[-  putMVar @ x >> readMVar @                   (4)
```

Here `@` is the state argument, in this case the `MVar`. We use `===` for observational equivalence, and `-[-` for *strict* observational refinement.

Property (1) shows that `readMVar` is idempotent; (2) shows that it is not merely a take followed by a put, it is rather a distinct operation; (3) and (4)

show that it does not modify the `MVar`, and blocks when empty. We see the effect of the interference operation in (2) and (4): in a single-threaded context, these would be equivalences; it is only when interference by another thread is introduced that the equivalence breaks down and the distinction is revealed.

Sometimes when expressing properties it is necessary to call upon other functions which we do not necessarily want to discover properties about. Such functions are commonly called *background* functions. If we include `tryPutMVar`, a non-blocking version of `putMVar`, as a background function, CoCo also finds these additional properties:

```
 readMVar @  -[-  readMVar @ >> tryPutMVar @ x
 readMVar @  ===  readMVar @ >>= \x -> tryPutMVar @ x          (5)
 readMVar @  ===  takeMVar @ >>= \x -> tryPutMVar @ x          (6)
putMVar @ x  ===  putMVar @ x >> tryPutMVar @ x
putMVar @ x  -[-  putMVar @ x >> tryPutMVar @ x1
```

Properties (5) and (6) show how important the choice of interference function is. These properties are actually refinements, not equivalences. In (5), if the interference were to empty a full `MVar` then the right term could restore its original value. In (6), the `MVar` becomes empty during the execution of the right term, but not the left. However, CoCo reports them as equivalences. As `interfere` *always* restores the `MVar` to its original state, it will never empty an originally full `MVar` in (5), and that it does not become empty during the execution of the left term of (6) is unimportant! Property (6) differs from (2) in that the `putMVar` in (2) will block if the interference puts into the `MVar`, whereas the `tryPutMVar` in (6) will not.

The above example takes about 6 seconds to run in total, and the output displayed here is the output of the tool, aside from the property numbers.

4 How CoCo Works

A simplified version of our approach is to generate all terms up to some syntactic size limit, compute and store their results, and then find properties by comparing the results of each pair of terms. This would be very slow, however. Following the lead of QuickSpec[2, 10] we make three key improvements:

1. We generate *schemas* with *holes*, rather than *terms* with *variables*, §4.1.
2. We only compute the results of the most general term of every schema, §4.2.
3. We interleave property discovery with schema generation, and aggressively prune redundant schemas, §4.3.

The main difference between our approach and QuickSpec is how we handle monadic operations, and that QuickSpec compares *equality* of term *results* whereas we compare *refinement* of term *side-effects*. Furthermore, we generate lambda-terms in a restricted setting whereas QuickSpec does not do so at all.

```
data Expr s h
  = Lit  String Dynamic
  | Var  TypeRep (Var h)
  | Bind TypeRep (Expr s h) (Expr s h)
  | Ap   TypeRep (Expr s h) (Expr s h)
  | State

data Var h
  = Hole  h
  | Named String
  | Bound Int

type Schema s = Expr s ()
type Term   s = Expr s Void
```

Fig. 2. Representation of Haskell expressions

4.1 Representing and Generating Expression Schemas

We can greatly reduce the number of expressions considered by not generating alpha-equivalent ones. Instead of producing an expression like

```
push @ x >> push @ y
```

we will first produce the expression

```
push @ ? >> push @ ?
```

where each `?` is a *hole* for a variable. These expressions-with-holes are called *schemas*. One schema can be instantiated into many *terms* by assigning variable names to groups of holes. The push-push schema has two semantically-distinct term instances: the single-variable and the two-variable cases.

Our expression representation is shown in Figure 2. The `Expr` type is parameterised by the state type and a *hole* type. The state parameter ensures expressions that assume different execution contexts cannot be inadvertently combined. The hole parameter allows for a statically-enforced distinction between schemas and terms. In most of the implementation we hide the details of this representation and instead provide *smart constructor* functions to ensure only well-typed expressions can be constructed.

Schema generation Generating new schemas is very straightforward. We give expressions a notion of *size*, corresponding roughly to the size of the `Expr` tree. Schemas are generated in size order, all of the schemas of one size at a time. We do this by simply trying all correctly-sized pairings of known schemas and keeping the type-correct ones.

Generation of schemas is interleaved with evaluation and testing. In this way we can use only the smallest known-equivalent of subschemas when generating new schemas.

Monadic expressions The expressions of most interest to us are *monadic* expressions produced by binding smaller monadic expressions into larger ones. We simplify this task by taking inspiration from Haskell's do-notation. Do-notation is a syntactic sugar for expressing sequences of monadic operations in an imperative style. For example, these two expressions are equivalent:

```
do x <- pop @
   push @ x
   push @ x
```

and

```
pop @ >>= (\x -> push @ x >> push @ x)
```

Rather than generating lambda-terms, we use a kind of first-class do-notation where the monadic bind operation binds the result of evaluating the *binder* to zero or more holes in the *body*. Restricting ourselves to this simpler case allows us to avoid many of the complexities of trying to generate lambda-terms directly.

For example, the generation of the schema `pop @ >>= \x -> push @ x` proceeds as follows:

1. Combine `pop` and `@` to produce `pop @`
2. Combine `push` and `@` to produce `push @`
3. Combine `push @` and `?` to produce `push @ ?`
4. Combine `pop @` and `push @ ?` to produce both `pop @ >> push @ ?` and `pop @ >>= \x -> push @ x`.

To avoid name clashes, bound variables use de Bruijn indices[4]. Names are only assigned when expressions are displayed to the user.

4.2 Evaluating Most General Terms

Time spent evaluating terms dominates the execution cost of CoCo. Déjà Fu performs both schedule bounding[8, 9] and partial-order reduction[6, 3, 14] to greatly reduce the number of executions considered[12], but in the worst case the number of executions is exponential in the number of threads, pre-emptive context switches, and blocking operations[8].

What is more, our term evaluation always involves at least two threads: the term thread executing the term itself, and an *interference thread*. The term thread may fork additional threads. The interference thread is essential to distinguish refinement from equality in some cases. For example, the following equivalence holds only when there is no interference:

```
readMVar @  ===  takeMVar @ >>= \x -> putMVar @ x
```

To avoid repeated work, we compute the results of all the terms for a schema when it is generated. We annotate each schema with some metadata, including its result-sets, and compare these cached values later when discovering properties. While possibly a significant space cost, storing this data reduces the execution time of some of our test applications from hours to minutes.

Deriving terms from schemas One schema may have many term instances. For example, given the schema

```
f (? :: Int) (? :: Bool) (? :: Bool) (? :: Int)
```

we can produce four semantically-distinct terms, here ordered from most general to most constrained:

```
f (w :: Int) (x :: Bool) (y :: Bool) (z :: Int)
f (w :: Int) (x :: Bool) (y :: Bool) (w :: Int)
f (w :: Int) (x :: Bool) (x :: Bool) (z :: Int)
f (w :: Int) (x :: Bool) (x :: Bool) (w :: Int)
```

We use a simple reduce-and-conquer algorithm to eliminate holes one type at a time:

1. Pick a type. Produce the set of all holes of that type, each represented by its position in the `Expr` tree.
2. For each partition of the hole-set make a distinct copy of the schema and in each case assign to each subset in the partition a distinct variable name.
3. If there are remaining hole types, continue recursively from (1).
4. Finally, sort the terms by number of distinct variables.

Evaluating terms In order to compute the results of every term for a schema, we need only consider the most general term. The results of all less-general terms can be derived from these most general results by restricting to cases where the variables are equal. For example, given the results of `f x y`, we produce the results of `f x x` by throwing away all those results of the former where `x` $\neq$ `y`.

We make use of a Déjà Fu feature called "subconcurrency," which cannot be implemented using the standard non-instrumented concurrency primitives. The `subconcurrency` function allows executing an action with concurrency-effects and examining its result in the same execution context, even if it deadlocks. We evaluate terms like so:

```
evaluate term seed = do
    state  <- initialise seed
    result <- subconcurrency (do fork (interfere state seed)
                                 term state)
    obs    <- observe state
    return (result, obs)
```

If instead we did not use `subconcurrency`, the observation of the state would never be made if execution of the term deadlocked. Lack of observation results in misleading properties, as many deadlocking operations may first alter the state. The implementation of `subconcurrency` requires more information about the state of threads than the GHC Haskell runtime system provides, and so can only be implemented when using the Déjà Fu abstraction.

Pruning neutral schemas A schema is neutral if it is (1) atomic; (2) never blocks or throws an exception; and (3) does not modify the shared state. For example, `readMVar` is not a neutral `MVar` operation as it may block, however there is a non-blocking alternative, `tryReadMVar`, which is neutral.

Neutral schemas are so-called because if `N` is a neutral schema and `S` is some other, then these identities hold:

```
N >> S  ===  S
S >> N  ===  S
```

If a schema is neutral, we do not consider it when constructing new schemas.

4.3 Property Discovery and Schema Pruning

Not only do we interleave generation with evaluation, we also interleave it with property-discovery. After all schemas of a given size are generated and their most general terms evaluated, we compare each such new schema against all smaller ones to discover equivalences and refinements.

As one schema may correspond to many terms, we may discover many properties between a pair of schemas. In practice, most of these properties are consequences of more general ones. For example, here the second property is redundant as it is a consequence of the first:

```
f x y  ===  g y
f x x  ===  g x
```

We solve this problem by first producing all properties between the pair of schemas, and then pruning properties which are simple consequences of another. We ensure that properties are discovered in such an order that a property can only be made redundant by one discovered earlier. One property is made redundant by another if (1) both properties are equivalences or both are refinements; and (2) the other has a more general allocation of holes to variables.

In order to avoid discovering the same property multiple times, we maintain a set of *smallest schemas*. Initially, all schemas are assumed to be smallest. If a syntactically smaller schema is a refinement of a larger one, the larger is annotated as "not smallest". When generating new monadic binds:

- A schema `S >> T` is only generated if both `S` and `T` are smallest schemas.
- A schema `S >>= \x -> T[x]` is only generated if `T` is a smallest schema.

We also only consider properties `S === T` or `S -[- T` where both `S` and `T` are smallest schemas.

Projection to a common namespace We compute the results of every term individually. Yet we construct properties from pairs of terms, so we have a problem. Each term introduces its own variable namespace: the variable "x" in one term is totally unrelated to the variable "x" in another. When discovering properties, we need to project both terms into a common namespace. Each variable in each term can either be given a unique name, or identified with a variable in the other term.

We define a function to produce all such projections:

```
data These a b = This a | That b | These a b

projections :: Expr s1 m1 h1
            -> Expr s2 m2 h2
            -> [[(These String String, TypeRep)]]
```

The `These` type is a three-valued sum, where `This x` means that a variable `x` from the left term remains distinct; `That y` means that a variable `y` from the right term remains distinct; and `These x y` means that variables `x` and `y` from the left and right terms, respectively, are identified. Given a list of `These` values, we can produce a consistent renaming of variables across both terms.

We never reduce the number of distinct variables in a term. To do so would only reproduce another term generated from the same schema.

As a pair of terms may have many projections, we may discover many properties between them: at most one for each projection. In practice, most of these properties are consequences of more general ones. We only keep the most general.

Background expressions Often when discovering properties it is useful to have some definitions around, but which are not themselves the primary subjects of observations. These are called *background* expressions. We only discover properties if both schemas contain at least one non-background expression.

5 Case Studies

We now present two illustrative case studies: concurrent stacks in §5.1, and semaphores in §5.2.

5.1 Concurrent Stacks

The stack is a simple and widely-used data structure. It has a very simple and efficient representation as a linked list, where `push` prepends a value, `peek` returns the head, and `pop` returns the tail.

```
newtype LockStack m a = LockStack (MVar m [a])

push :: MonadConc m => a -> LockStack m a -> m ()
push a (LockStack v) = modifyMVar v (\as -> return (a:as, ()))

pop :: MonadConc m => LockStack m a -> m (Maybe a)
pop (LockStack v) =
  modifyMVar v (\as -> (drop 1 as, listToMaybe as))

peek :: MonadConc m => LockStack m a -> m (Maybe a)
peek (LockStack v) = fmap listToMaybe (readMVar v)
```

Fig. 3. A lock-based mutable stack.

Lock-based stacks Mutable stacks are commonly used for synchronisation amongst multiple threads. A very simple mutable stack is just a linked list inside an `MVar` shared variable, as in Figure 3. We now run CoCo on those functions, where the initialisation function constructs a stack from a list, the observation function converts it back to a list, and the interference function sets the contents of the stack to a given list. CoCo discovers the following properties:

```
peek @  -[-  push x @ >> pop @                          (7)
peek @  -[-  (pop @) ||| (push x @)                     (8)
peek @  -[-  pop @ >>= \m -> whenJust push @ m          (9)
```

Here `|||` is concurrent composition. Property (7) may seem surprising: the left term returns the top of stack whereas the right term returns the value pushed. CoCo does not consider equality of results when determining properties, only the effect on the state. Both terms leave the stack as it was originally. Property (8) is a consequence of (7). Property (9) is analogous to the `readMVar` properties presented in §3, as we might expect given how the stack operations are defined.

Buggy functions Suppose we add an *incorrect* `push2` function, which is meant to push two values atomically, but actually only pushes the second value twice. CoCo finds this property:

```
push2 x1 x @  -[-  push x @ >> push x @
```

As this is a strict refinement, we now know that `push2` is more deterministic in some way than two `pop`s. As we know that the composition of two `pop`s is not atomic, this strongly suggests that `push2` is. We can also see the effect of `push2` on the state, and quite clearly that it is incorrect!

Choice of observation As CoCo uses a programmer-supplied observation function in its property-discovery process, the programmer can supply different observations in order to discover different properties. By changing the observation of our

```
newtype TreiberStack m a = TreiberStack (CRef m [a])

push :: MonadConc m => a -> TreiberStack m a -> m ()
push a (TreiberStack r) = modifyCRefCAS r (\as -> (a:as, ()))

pop :: MonadConc m => TreiberStack m a -> m (Maybe a)
pop (TreiberStack r) =
  modifyCRefCAS r (\as -> (drop 1 as, listToMaybe as))

peek :: MonadConc m => TreiberStack m a -> m (Maybe a)
peek (TreiberStack r) = fmap listToMaybe (readCRef r)
```

Fig. 4. A lock-free mutable stack.

stack from equality-as-a-list to `peek`, we discover a new collection of properties. Here we have fixed the `push2` function to behave correctly and also removed `|||` from the signature.

```
          peek @  -[-  push x @ >> pop @
          peek @  ===  pop @ >>= \m -> whenJust push @ m
        push x @  ===  pop @ >> push x @
       push x1 @  ===  push2 x x1 @                          (10)
       push x1 @  ===  push x @ >> push x1 @                 (11)
whenJust push @ m  ===  whenJust (push2 x) @ m
```

Properties (10) and (11) show the power of supplying a custom observation function: in the left and right terms, the stack states are *not* equal. In both (10) and (11) the left term increases the stack depth by one, and the right by two. We now see clearly that `push2` leaves its second argument on the top of the stack. We could not directly observe this before, as a single push would leave the stack sizes out of balance. Throwing away unnecessary details, in this case the tail of the stack, allows us to see more than we previously could.

Choice of implementation Due to their blocking behaviour, `MVar`s can have poor performance when there is contention. An alternative concurrency primitive is the `CRef`,[1] corresponding to a lock-free mutable location in memory. An atomic compare-and-swap operation updates `CRef` values efficiently even when there is contention.

The Treiber stack[11] is a lock-free stack implemented using compare-and-swap. It protects against the case where: (1) a thread T_1 pops from the stack, and stores references to the top and second-to-top; (2) thread T_1 then frees the top of the stack; (3) thread T_2 allocates a new element, which is given the same reference as the old top-of-stack, and pushes it to the stack; (4) thread T_1 then compares its now-invalid reference, decides that nothing has changed, and

[1] In regular GHC Haskell this is the `IORef`, here Déjà Fu deviates from the norm.

replaces the top of stack with the old second-to-top. Even though thread T_1 does not dereference the reference it stored, the freeing of that reference potentially causes a fault. In a pure garbage-collected language such as Haskell, this is not possible, as a new value cannot compare reference-equal to any reachable value, and old values cannot be changed. The implementation simplifies somewhat, as shown in Figure 4.

A feature of CoCo that differentiates it from other property-discovery tools is the ability to compare two different signatures which have compatible observation types. We can compare the `MVar` and `CRef` stacks by simply supplying both signatures to the tool:

```
                    popM @  ===  popT @                    (12)
                   peekM @  ===  peekT @                   (13)
                 pushM x @  ===  pushT x @                 (14)
          popM @ >> popM @  ===  popT @ >> popT @
       whenJust pushM @ m   ===  whenJust pushT @ m
      popM @ >> pushM x @   ===  popT @ >> pushT x @
  pushM x @ >> pushM x1 @   ===  pushT x @ >> pushT x1 @
popM @ >> whenJust pushM @ m  ===  popT @ >> whenJust pushT @ m
whenJust pushM @ m >> popM @  ===  whenJust pushT @ m >> popT @
```

Here we use the list observation again. Functions with names ending `M` are for `MVar` stacks, functions with names ending `T` for `CRef`-based Treiber stacks. Properties (12), (13), and (14) tell us what we want to know: the `CRef` stack is equivalent to the `MVar` stack.

A common approach when first writing a program is to do everything in a simple and clearly correct fashion. After checking correctness, to gradually rewrite components to meet performance requirements. At which point testing must establish that the rewritten components preserve the behaviour. The ability to determine observational equivalence of different implementations of the same API is an alternative to the more-common unit-testing for this task[7].

5.2 Semaphores

A semaphore is a synchronisation primitive used to regulate access to some resource in a multi-threaded setting. A semaphore can be thought of as a record of how many units of some abstract resource are available, with operations to adjust the record in a race-free way. *Binary semaphores* only have two states, and are used to implement locks. *Counting semaphores* have an arbitrary number of states.

An implementation of counting semaphores is provided in the Control.Concurrent.QSemN library module. As with the `MVar` and `CRef`, Déjà Fu provides a typeclass-generalised version which we use here.

Signatures Figure 5 shows the signature we provide to CoCo. CoCo supports polymorphic function types, as can be seen in the type of `|||`, where `A` and `B`

```
sig :: Sig (QSemN Concurrency) Int Int
sig = Sig
  { initialState = new . abs
  , expressions =
    [ lit "wait"   (wait   :: QSemN Concurrency -> Int -> Concurrency ())
    , lit "signal" (signal :: QSemN Concurrency -> Int -> Concurrency ())
    ]
  , backgroundExpressions =
    [ lit "|||" ((|||) :: Concurrency A -> Concurrency B -> Concurrency ())
    , lit "+" ((+) :: Int -> Int -> Int)
    , lit "-" ((-) :: Int -> Int -> Int)
    , lit "0" (0 :: Int)
    , lit "1" (1 :: Int)
    ]
  , observation = remaining
  , backToSeed = remaining
  , setState = \q n -> let i = n `div` 2 in wait q i >> signal q i
  }
```

Fig. 5. CoCo signature for the QSemN type.

are types we use as type *variables*. Furthermore, CoCo infers the necessary hole types from the monomorphic types in the signature.

The `new`, `wait`, `signal`, and `remaining` functions are provided by the `QSemN` library module. We construct a new `QSemN` by allocating an arbitrary amount of resource; we observe how much resource remains; and we interfere by taking and then replacing half of the resource. The interference thread is interleaved with the term thread, so it may cause the term thread to block.

In the current version of CoCo pruning of redundant schemas performs poorly with non-monadic subschemas. Many redundant properties involving equivalent arithmetical expressions are found. In the remainder of this section we discuss selected properties, not the entire list reported by CoCo.

Waiting and signalling

```
wait @ 0  ===  signal @ (0 + 0)                    (15)
wait @ 0  ===  wait @ 0 >> wait @ 0                (16)
```

Property (15) tells us that the effect of waiting for zero resource and of signalling the availability of a zero resource are the same — neither affects the state of the semaphore (incidentally, it also illustrates the suboptimal schema pruning: there is no arithmetic simplification by which `0 + 0` could be reduced). If the observations behind Property (16) were only unchanging states, CoCo would prune it away. So, waiting for zero resource is *not* a neutral operation. This suggests that it may block.

Indeed, a `QSemN` is implemented using an `MVar`, and the first action of `wait` applies `takeMVar` to it. So waiting, even if it does not change anything, may block. Only one thread can inspect the quantity at a time.

CoCo also generates properties revealing another implementation detail which is not strictly guaranteed by the abstract definition of a semaphore:

```
      signal @ 1  ===  wait @ (0 - 1)
signal @ (1 + 1)  ===  wait @ (0 - (1 + 1))
```

However, the more general property that signalling a positive resource `n` appears to be equivalent to *waiting for a negative resource* `-n` is not found. If we extend our signature with `abs` and `negate` CoCo does report:

```
signal @ (abs x)  ===  wait @ (negate (abs x))                (17)
```

Composability and thread safety CoCo reports the composability and thread-safety of `signal`:

```
signal @ (x + x1)  ===  signal @ x >> signal @ x1             (18)
signal @ (x + x1)  ===  (signal @ x) ||| (signal @ x1)        (19)
```

Property (18) is implied by (19). CoCo does *not* report similar properties for `wait`. They are only true if the sum being waited for is available; otherwise the non-atomic term may claim some but not all of the resource. Discovering conditional properties of this kind would require CoCo to be able to synthesise preconditions over the free variables of a term.

Finally, CoCo reports a somewhat surprising result about a *lack* of composability:

```
signal @ 0  -[-  signal @ x >> wait @ x                        (20)
```

Intuitively, this should be an equivalence. The only interference we have specified may occur is taking and then returning half of the original resource. So releasing a quantity of resource and then claiming it again should be fine. As we are about to see, it is due to negative values that (20) is only a refinement.

Custom types Perhaps a better interface for semaphores would only allow non-negative quantities. The strange property (17) would no longer be reported. The change might also avoid accidental breakage in the future if the semantics of negative values are unwittingly changed.

CoCo supports a large number of types, but not all. If the programmer wishes to synthesise values outside of this built-in collection, they must provide a *type information record*. For example, here is a possible definition of the type and information of natural numbers:

```
newtype Nat = Nat Int

typeInfos = (natTR, natTI) : defaultTypeInfos where
  natTR = typeRep (Proxy :: Proxy Nat)
  natTI = TypeInfo
    { listValues = map (toDyn . Nat) [0..]
    , varName    = 'n'
    }
```

The `TypeInfo` record contains a possibly-infinite list of dynamically-typed values, and a character to use when generating variable names. CoCo requires a map from run-time type representations to type information, which it uses when generating values and printing terms. In this way, the programmer can extend CoCo to work with arbitrary types, or alter the behaviour of existing types.

If we alter the signature so that `signal` and `wait` use the `Nat` type rather than `Int` property (20) becomes an equivalence:

```
signal @ 0  ===  signal @ n >> wait @ n
```

We could pursue this issue further by examining the two terms with Déjà Fu when given a negative quantity, or we could change the type of the function to forbid that case. Signalling or awaiting a negative quantity is a breach of the semaphore protocol. Ideally, illegal states should be unrepresentable.

6 Related Work

QuickSpec The main related work to ours is QuickSpec[2, 10]. The implementation of QuickSpec 2 served as a great inspiration to us during the implementation of CoCo. QuickSpec 2 addresses a number of problems with the original. The original QuickSpec does not interleave testing and pruning. It enumerates terms, rather than schemas, to discover equations. Furthermore, the programmer must specify how many variables of each type may be used, and it does not support polymorphism. In contrast, QuickSpec 2 enumerates schemas rather than terms, infers necessary hole types, supports polymorphism, and interleaves testing and pruning. Neither version of QuickSpec supports functions with side-effects, or the generation of lambda-terms.

Speculate The Speculate[1] tool is similar to QuickSpec 2 but which, in addition, can discover *conditional equations*. Properties discovered may have preconditions, which the tool can find. CoCo does not support conditional equations, but they could be very useful. To return to the semaphore case study from §5.2, the presence of conditional equations would allow us to discover the conditional property `x >= 0 ==> signal @ x === wait @ (negate x)`, without needing to introduce the `abs` function. Like QuickSpec, Speculate does not support functions with side-effects or generating lambda-terms.

Daikon The Daikon[5] tool discovers *likely invariants* of C, C++, Java, and Perl programs. It observes variables in memory during the execution of a program, and applies machine learning techniques to discover properties that seem to hold. These properties may include: pre- and post-conditions of statements, and equational relationships between variables at a given program point and functions from a library. Daikon does not synthesise and test program terms, however. It is only able to discover invariants which exist in the program. In contrast, the tools mentioned so far, including CoCo, discover properties of combinations of expressions that may not appear in the original program at all.

7 Conclusions and Further Work

We have presented a new tool, CoCo, which can automatically discover algebraic properties of side-effecting functions operating on shared state. Programmer-supplied interference functions can be used to distinguish atomic from nonatomic operations. The tool can be used to aid program understanding. Our overall aim is to help programmers overcome the barriers of testing concurrent programs.

Value of reported properties Although only supported by a finite number of test cases, the properties reported by CoCo are surprisingly accurate in practice. These properties can provide helpful insights into the behaviour of functions. As demonstrated in the semaphore case study (§5.2), surprising properties can suggest that implementations of some functions rely on unstated assumptions. It can be difficult to read concurrent source code and grasp all of its consequences.

Ease of use Ideally, a testing tool should not force the programmer to structure their code in a specific way. CoCo requires the use of the concurrency typeclass used by Déjà Fu, which is not widespread in practice. However, it has been our experience that porting standard Haskell code to the necessary abstraction is a very type-directed and mechanical process, requiring little insight.

CoCo also requires a small amount of information about types and signatures. There is a collection of default type information values, supporting many standard Haskell data types, but it does not contain everything. The CoCo library provides some support for deriving type information values automatically, but none for signature values.

Further work

There are two main avenues open for further work:

Conditional equations Speculate[1] discovers conditional equations automatically, which greatly expands the range of properties which can be found. This is useful as we see how our functions behave in different situations, rather than just in general. Currently CoCo can only discover conditional equations if a specific precondition is supplied by the programmer.

Term rewriting Both QuickSpec[2, 10] and Speculate use term rewriting to prune the discovered properties and to avoid testing many cases. This is difficult to do with concurrency, as side-effects may have spooky action at a distance, as is the case with relaxed memory[14]. Such behaviours make the effect of composing two terms far less predictable. Even so, it may still be possible in some cases to use something like term rewriting to prune properties. For example, term rewriting applied only to pure schema could improve property discovery in the semaphore case study (§5.2), despite executing fewer schemas.

References

1. Rudy Braquehais and Colin Runciman. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. Accepted for publication at the Haskell Symposium 2017.
2. Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *Proceedings of the 4th International Conference on Tests and Proofs*, TAP'10, pages 6–21. Springer-Verlag, 2010.
3. Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded partial-order reduction. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 833–848. ACM, 2013.
4. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.
5. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
6. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121. ACM, 2005.
7. J. He, C. A. R. Hoare, and J. W. Sanders. *Data refinement refined resume*, pages 187–196. Springer Berlin Heidelberg, 1986.
8. Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455. ACM, 2007.
9. Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 362–371. ACM, 2008.
10. Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximillian Algehed. Quick specifications for the busy programmer. Submitted for publication. Available at http://www.cse.chalmers.se/ nicsma/.
11. Kent T. Treiber. *Systems Programming: Coping with Parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.
12. Michael Walker. Déjà Fu: A concurrency testing library for Haskell. Technical report, University of York, Department of Computer Science, 2016.

13. Michael Walker and Colin Runciman. Déjà Fu: A concurrency testing library for Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 141–152. ACM, 2015.
14. Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 250–259. ACM, 2015.