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 recovers a declarative view of concurrent Haskell functions operating on some shared state. This declarative view is presented as a collection of automatically discovered properties. These properties are about *refinement and equivalence of effects*, rather than equality of final results. The tool is based on testing in a dynamically pruned search-space, rather than static analysis or theorem proving. Case studies about concurrent stacks and semaphores demonstrate how use of CoCo can inform understanding of program behaviour.

1 Introduction

Concurrency is a necessary paradigm for many applications, and yet it is difficult to get right in an imperative setting, where the order of effects is both important and unpredictable. Declarative programming, whether logical or functional, offers the promise of a simpler alternative; the programmer describes the desired program, and does not need to worry about the explicit order of effects.

Haskell is a purely functional language. Concurrency in Haskell is modelled with a monad abstraction which is built on top of effectful operations on shared state[11]. Once again, the order of effects is both important and unpredictable. Concurrent Haskell programs are not so declarative.

In this paper we present CoCo, a tool to recover a declarative view of concurrent programs. CoCo takes as input a collection of operations on some shared mutable state. CoCo outputs are declarative properties of equivalence and refinement between concurrent expressions: see §3 and §5 for examples.

A list of such declarative 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 program; and (3) the absence of expected properties, or the presence of unexpected ones, may indicate an error.

Our technique uses testing, so is potentially unsound. A property is a conjecture supported only by a finite set of test cases. So some reported properties may not hold in general.

Contributions We present a method, based on program synthesis and systematic concurrency testing, to discover properties of stateful operations in a declarative language. Furthermore, we demonstrate the viability of this method by implementing the CoCo tool in Haskell. We then obtain illustrative results from CoCo for some simple applications.

Roadmap The rest of the paper is structured as follows: §2 introduces three key concerns in the implementation of a tool such as CoCo. §3 gives an introductory example. §4 gives a detailed discussion of how CoCo 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 evaluates the approach.

2 Key Concerns of Observing Concurrent Programs

In implementing a tool to discover properties of *concurrent* programs, we have some concerns which are not applicable to sequential programs. Firstly, concurrent programs are nondeterministic; so if we simply compared results of single executions, the discovered properties may not hold in the general case. Secondly, mutable state is subject to interference from other threads; so if we do not consider concurrent interference, the discovered properties may not hold when there are more threads involved. Finally, we need to decide what it means for two concurrent programs to be related.

Nondeterminism If we restrict the nondeterminism in our program to schedule nondeterminism, we can use systematic concurrency testing (SCT)[4,7,9,10] techniques. These techniques aim to test a variety of schedules, making use of runtime knowledge of the program to reduce the number of required executions, without necessarily sacrificing completeness.

We have previously developed Déjà Fu[14] an SCT tool for Haskell, based on a typeclass-abstraction over the primitive concurrency operations. CoCo uses Déjà Fu to produce the set of results of a generated program fragment.

Interference We do not know what sort of interference may lead to interesting results. So CoCo requires the programmer to supply a function with effects, which is executed concurrently during property discovery, to provide this interference. By supplying different sorts of interference, the programmer can see how the API they provide behaves in different concurrent contexts.

Properties We formulate our properties in terms of *observational refinement*[8], where the observations we take are snapshots of the shared state. CoCo requires the programmer to supply an observation function to produce these snapshots. By varying their observation function, the programmer can see different aspects of the API they provide.

We define a *behaviour* of a concurrent program as a pair of a final observation, taken after the program terminates, and a possible *failure*. Failures are states like a deadlock, or an uncaught exception. By considering the set of a program's possible behaviours, rather than simply final observations, we can distinguish between operations which may fail and those which do not. Properties that we report are of the form `A === B`, meaning that the sets of behaviours of `A` and `B` are equal; and `A ->- B`, meaning that the set of behaviours of `A` is a strict subset of the set of behaviours of `B`.

```
type C = Concurrency

sig :: Sig (MVar C Int) (Maybe Int) (Maybe Int)
sig = Sig
  { initialise  = maybe newEmptyMVar newMVar
  , expressions =
    [ -- example 1
      lit "putMVar"  (putMVar  :: MVar C Int -> Int -> C ())
    , lit "takeMVar" (takeMVar :: MVar C Int -> C Int)
    , lit "readMVar" (readMVar :: MVar C Int -> C Int)
    ]
  , backgroundExpressions =
    [ -- example 2
      lit "tryPutMVar" (tryPutMVar :: MVar C Int -> Int -> C Bool)
    ]
  , interfere  = \v _ -> putMVar v 42
  , observe    = \v _ -> tryReadMVar v
  , backToSeed = \v _ -> tryReadMVar v
  }
```

Fig. 1. CoCo signature for `MVar`s holding `Int`s.

3 An Illustrative Example

Let us now show an example use of CoCo. We consider a type of concurrent shared variable in the Haskell libraries. An `MVar` is a mutable memory cell which may be *full* or *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. We shall examine 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 emptying the `MVar`. Each function has a non-blocking *try* variant, which returns an indicator of success.

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. In this case, 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 effects in the concurrency execution context. The `MVar` type is parameterised by the monad type; `MVar` is an abstract type, with the concrete type determined by the monad.

Term	Seed	Final State	Deadlocks
Read	`Nothing`	`Just 42`	No
	`Just 0`	`Just 0`	No
Take / Put	`Nothing`	`Just 42`	No
	`Just 0`	`Just 0`	No
		`Just 42`	Yes

Table 1. The behaviours of the terms in property (2).

Signatures When we use CoCo, we must provide the functions and values which may appear in properties. We must also provide a way to initialise the state, an observation function, and an interference function. We call this collection of programmer-supplied definitions the *signature*.

Figure 1 shows a signature for `MVar` operations. The initialisation function constructs an empty or a full `MVar`. The interference function simply stores a new value. The observation function takes a snapshot of the state. The `backToSeed` function is used to check whether the state has been changed: if the original and final seed values are the same, the state is unchanged.

It is essential to provide an initialisation function which gives a representative collection of states, and an interference function which can disrupt the functions of interest. If our initialisation function only produced a full `MVar`, we could find properties which do not hold when the `MVar` is empty. Because our interference function only writes to the `MVar`, we may find properties which do not hold when there are multiple consumers. Developing a fuller understanding of the functions under test may require examining the different property-sets found under different execution conditions.

`MVar` properties Given `putMVar`, `takeMVar`, and `readMVar`, CoCo produces:

```
 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`.

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 that it does not block when the `MVar` is full. Property (4) may appear to be type-incorrect, but remember that CoCo does not consider equality of term results, only the effects.

We see the effect of the interference in (2): with no other producers, this would be an equivalence; it is only when interference by another thread is introduced that the equivalence breaks down and the distinction is revealed. Table 1 shows the possible behaviours. This property is a strict refinement because, while the behaviours for the seed value `Nothing` are the same, the behaviours of the left term for the seed value `Just 0` are a strict subset of the behaviours of the right.

Background expressions Sometimes when expressing properties it is necessary to call upon other expressions which are of secondary interest. Such expressions are commonly called *background* expressions. A property is only reported if each side includes at least one non-background expression. If we include `tryPutMVar` as a background expression, CoCo discovers these additional properties:

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

Property (5) shows how important the choice of interference function is. The left and right terms are not equivalent. If the interference were to empty a full `MVar` then the right term could restore its original value. As our interference function only produces, rather than consumes, it will never alter the value in a full `MVar`.

The above example takes about 4 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 behaviours, and then find properties by comparing the sets of behaviours of each pair of terms. This would be slow, however. Following the lead of QuickSpec[3,12] we make three key improvements:

1. We generate *schemas* with *holes*, rather than *terms* with *variables* (§4.1)
2. We only compute the set of behaviours 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 *behaviours*. Furthermore, we generate lambda-terms in a restricted setting whereas QuickSpec does not do so at all.

4.1 Representing and Generating Expression Schemas

We can greatly reduce the number of expressions by not generating alpha-equivalent ones. Instead of generating an expression like `push @ x >> push @ y` we will instead generate 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.

```
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.

Figure 2 shows our expression representation. 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. Each `Expr` constructor carries around a type (except the state, which is implicit). We hide the details of this representation and provide *smart constructor* functions to ensure only well-typed expressions can be constructed.

Schema generation Generating new schemas is straightforward. We give expressions a notion of *size* and generate schemas in size order. The needed expressions of size 1 are supplied in the user's signature. For larger sizes we combine pairs of appropriately sized known schemas and keep the type-correct ones.

We interleave generation with evaluation and property discovery. In this way we can partition schemas into equivalence classes and use only the smallest of known-equivalent schemas when generating new ones.

Monadic expressions The expressions of most interest to us are *monadic* expressions. Such expressions allow us to combine smaller effects to create 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, which has explicit variable bindings and makes the sequencing of effects clear. 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 schema `pop @ >>= \x -> push @ x` is generated like so:

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`.

Bound variables use de Bruijn indices[5]. 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. In the worst case the number of executions needed for a term is exponential in the number of threads, pre-emptive context switches, and blocking operations[9].

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, such as in property (2).

To avoid repeated work, we compute the behaviours of all the terms for a schema when it is generated. We annotate each schema with some metadata, including its behaviour-sets, and compare these cached behaviours 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 a schema with two holes of two types, we can produce four semantically distinct terms, here ordered from most general to most constrained:

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

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:

1. Pick a type and find the set of all holes of that type.
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).

Evaluating terms To compute the behaviours of every term for a schema, we need only consider the most general term. The behaviours of all less-general terms can be derived from the most general case by restricting to cases where the variables are equal. For example, given the behaviours of `f x y`, we throw away those where `x` $\neq$ `y` to obtain the behaviours of `f x x`.

Déjà Fu allows us to make an observation of the final state even if evaluation of the term deadlocks. This is essential, as an operation which deadlocks may have altered the state before blocking.

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. We solve this problem by first producing all properties between the pair of schemas, and then pruning properties which are simple consequences of another. Property P_2 is made redundant by property P_1 if (1) both P_1 and P_2 are equivalences or both are refinements; and (2) P_1 has a more general allocation of variables to holes. As `->-` is *strict* refinement, it is impossible for both `S === T` and `S ->- T` to hold.

Smallest schemas To avoid discovering the same property multiple times, we maintain a set of *smallest schemas*. At first we assume all schemas 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.

Neutral schemas A schema `N` is neutral if and only if, for all other schemas `S`, these identities hold: `N >> S === S === S >> N`. For example, `readMVar` is not a neutral `MVar` operation, as it may block, but the non-blocking alternative `tryReadMVar` is neutral. A sufficient condition for a schema to be neutral is if its most general term instance is (1) always atomic; (2) never fails; and (3) never modifies the state.

We use a heuristic method based on execution traces to determine if a schema is atomic, and use the seed values to determine if it modifies the state. If a schema is judged to be neutral, we do not use it when constructing larger schemas.

Projection to a common namespace We compute the behaviours of every term individually, yet we construct properties from pairs of terms. Each term introduces its own variable namespace: the variable "x" in one term is unrelated to the variable "x" in another. When discovering properties, we must first 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 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.

```
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.

5 Case Studies

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

5.1 Concurrent Stacks

Lock-based stacks Mutable stacks are commonly used for synchronisation amongst multiple threads. A simple mutable stack is just an immutable 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 @                          (6)
peek @  ->-  (push x @) ||| (pop @)                     (7)
peek @  ->-  pop @ >>= \m -> whenJust push @ m          (8)
```

Here `whenJust` is defined as `\f s -> maybe (pure ()) (`f` s)` and `|||` is concurrent composition. Property (6) may seem surprising: the left term returns the top of stack whereas the right term returns the value pushed. Remember that CoCo does not consider equality of results when determining properties, only the effect on the state. Property (7) is a consequence of (6). Property (8) 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 which 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 `push`es. As we know that the composition of two `push`es is not atomic, this strongly suggests that `push2` is. We can also see the effect of `push2` on the state, and that it is incorrect!

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

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

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

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

Fig. 4. A lock-free mutable stack.

Choice of observation Properties are discovered using a programmer-supplied observation function, so different functions can be used to discover different properties. By changing the observation of our 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 @                          (9)
       push x1 @  ===  push x @ >> push x1 @                (10)
whenJust push @ m  ===  whenJust (push2 x) @ m
```

Properties (9) and (10) show the power of supplying a custom observation function: in the left and right terms, the stack states are *not* equal. In both (9) and (10) the left term increases the stack depth by one, and the right by two. We now see 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.

It is important to bear in mind that there is no *best* observation to make, no *best* interference to consider, and no *best* set of properties to discover. Each choice of observation and interference will reveal something about the functions under test. By considering different cases, we can arrive at a fuller understanding of our code.

Choice of implementation Due to their blocking behaviour, `MVar`s can have poor performance under 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 with contention. Figure 4 shows our implementation, which is similar to the `MVar` stack.

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

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, each of which contains `push`, `pop`, `peek`, `whenJust`, and `|||`. CoCo then reports 19 properties, including these three:

```
   popM @  ===  popC @                                    (11)
  peekM @  ===  peekC @                                   (12)
pushM x @  ===  pushC x @                                 (13)
```

Here we use the list observation again. Functions with names ending `M` are for `MVar` stacks, functions with names ending `C` for `CRef` stacks. Properties (11), (12), and (13) 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, we may 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[8].

5.2 Semaphores

A semaphore is a common synchronisation primitive. 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.

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` are types we use as type *variables*. The `commLit` function indicates that the supplied binary function is *commutative*, which is used to prune the generated schemas further.

The `new`, `wait`, `signal`, and `remaining` functions are provided by the `QSemN` library module. We construct a new semaphore 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.

CoCo finds 57 properties in this example, so in the remainder of the section we only discuss selected properties.

Waiting and signalling CoCo tells us that the effect of waiting for zero resource and of signalling the availability of zero resource are the same — neither affects the state of the semaphore:

```
  wait @ 0  ===  wait @ 0 >> wait @ 0                    (14)
signal @ 0  ===  wait @ 0 >> wait @ 0
```

```
type C = Concurrency

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

Fig. 5. CoCo signature for the QSemN type.

Property (14) also shows that waiting for zero resource is not a neutral operation, as if it were CoCo would prune the property away. This suggests that `wait` may block.

CoCo also generates properties revealing another implementation detail, that the programmer can `wait` for a negative value instead of calling signal:

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

We might suspect that the more general property `signal @ x === wait @ (-x)` holds for all positive `x`. CoCo finds this form if we extend our signature with `abs` and `negate`:

```
signal @ (abs x)  ===  wait @ (negate (abs x))                (15)
```

A lack of composability CoCo reports some strict refinements involving `signal` and `wait` where we might expect equivalences:

```
      signal @ 0  ->-  signal @ x >> wait @ x                 (16)
signal @ (x + x1)  ->-  signal @ x >> signal @ x1             (17)
signal @ (x + x1)  ->-  (signal @ x) ||| (signal @ x1)        (18)
```

We have just seen with property (15) that funny things happen with negative numbers, so it should be no surprise that these refinements are only equivalences when `x` and `x1` are positive.

Term size	1	2	3	4	5	6	7	8
Schemas	15	29	56	88	238	385	1689	2740
Properties	0	0	0	0	1	1	55	55
Time (s)	0.03	0.03	0.45	0.45	9.2	9.2	970	970
Time / schema^2	1.3e-4	3.6e-5	1.4e-4	5.8e-5	1.6e-4	6.2e-5	3.4e-4	1.3e-4

Table 2. Scaling behaviour of the semaphore case study.

Types Signalling or awaiting a negative quantity is a breach of the semaphore protocol. Perhaps a better interface for semaphores would only allow nonnegative quantities. The change might avoid accidental breakage in the future if the semantics of negative values are unwittingly changed.

CoCo supports many types, but not all. If the programmer wishes to use types outside of the built-in collection, they must provide some information: a way to enumerate values, an equality predicate, and a symbol to use in variable names. In this way, the programmer can extend CoCo to work with arbitrary types, or alter the behaviour of existing types. If we have `signal` and `wait` use natural numbers rather than integers, properties (16–18) become equivalences:

```
      signal @ 0  ===  signal @ n >> wait @ n
signal @ (n + n1)  ===  signal @ n >> signal @ n1
signal @ (n + n1)  ===  (signal @ n) ||| (signal @ n1)
```

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

Scaling Table 2 shows how CoCo scales as the term size increases. The execution time grows rapidly, but the time to compare pairs of schemas So reducing the number of schemas is the most effective way to reduce the execution time. One such area for future improvement is in cases where one schema is an instance of another. Such schemas may arise when the signature includes constants. For example, the schema `signal @ 1` is an instance of `signal @ x`. The 'most general term' rule does not apply here, as these are *different* schemas. If CoCo were able to synthesise preconditions, it would be possible in some cases to eliminate constants from signatures, solving this problem.

6 Related Work

QuickSpec and Speculate The main related work to ours is QuickSpec[3,12], which automatically discovers equational laws of pure functions by generating schemas and testing terms. The Speculate[1] tool is similar to QuickSpec but in addition can discover *conditional equations* and *inequalities*. Speculate properties may have preconditions, which the tool can find. CoCo does not support conditional equations, but they could be 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. Neither QuickSpec nor Speculate support functions with effects or generating lambda-terms.

Bach The Bach[13] tool uses a database of examples of input/output values from functions to synthesise properties using a Datalog-based oracle. It uses a notion of *evidence* to decide whether an inferred property holds: negative evidence consists of counterexamples; positive evidence consists of witnesses. Bach requires functions to have at most one output for each input, to construct negative evidence. This makes Bach unsuitable for concurrency, which is nondeterministic.

Daikon The Daikon[6] tool discovers *likely invariants* of C, C++, Java, and Perl programs. It observes variables during program execution, applying machine learning techniques to discover properties. Daikon does not synthesise and test program terms. 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 Evaluation

Our broad aim is to help programmers overcome the difficulty of writing correct concurrent programs even in a declarative setting. To work towards this, we have presented a new tool, CoCo, to discover behavioural properties of effectful functions operating on shared state.

Applicability beyond Haskell CoCo is tied to Haskell in two ways: it has some knowledge of Haskell types, which is used to generate expressions; and it relies on the Déjà Fu tool to find the results of executing an expression under different schedules. However, it could be reimplemented for another language. For example, in Erlang the objects of interest are processes. Initialisation is to create a process in a known state. Observation is to request information from a process. Interference is to instruct a process to change its internal state. The PULSE tool for systematically testing Erlang programs[2] would play the part of Déjà Fu.

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.

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 from Déjà Fu, which is not widespread in practice. However, it has been our experience that reformulating concurrent Haskell code for the necessary abstraction is a type-directed and mechanical process, requiring little insight.

References

1. Rudy Braquehais and Colin Runciman. Speculate: Discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 40–51, New York, NY, USA, 2017. ACM.
2. Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 149–160. ACM, 2009.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. J. He, C. A. R. Hoare, and J. W. Sanders. *Data refinement refined resume*, pages 187–196. Springer Berlin Heidelberg, 1986.
9. 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.
10. 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.
11. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 295–308. ACM, 1996.
12. Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximillian Algehed. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27, 2017.
13. Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. Discovering relational specifications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 616–626. ACM, 2017.
14. 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.