Applications of Applicative Proof Search

Liam O'Connor
UNSW Australia and CSIRO Data61
liamoc@cse.unsw.edu.au

Abstract

In this paper, we develop a library of typed proof search procedures, and demonstrate their remarkable utility as a mechanism for proof-search and automation. We describe a framework for describing proof-search procedures in Agda, with a library of tactical combinators based on applicative functors. This framework is very general, so we demonstrate the approach with two common applications from the field of software verification: a library for property-based testing in the style of SmallCheck, and the embedding of a basic model checker inside our framework, which we use to verify the correctness of common concurrency algorithms.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords Agda, testing, properties, model checking, proof, automation, critical section, concurrency

1. Introduction

Computers have been used to establish proofs for decades. For example, the original proof of the four colour theorem (Appel et al. 1977) and the Kepler conjecture (Hales 2005) relied on a computer program for exhaustive case analysis. Closer to home, proofs *about programs* are very commonly established automatically, using everything ranging from model checkers, which establish properties about programs by examining their state space, to test frameworks, which produce counterexamples to hypothetical specifications.

Such computerised proof-searches are opaque. Their result consists typically of a single bit, without context: either the search succeeded or it failed. What this means in terms of proof is left up to a human to decide. Recently, for the aforementioned mathematical theorems, a great deal of work has been done to formalise these proof-searches inside a mechanical theorem prover (Gonthier 2008; Hales 2006), in order to recover the provenance of that single bit. In other words, they formally verify the proof-search program to ensure that the bit it produces means what we think it means.

Dependently typed languages hold a lot of promise in this area. By unifying the language of programs and the language of proofs, such proof-search programs can produce not merely a yes-or-no answer, but actual *evidence* to support the proposition they are attempting to automatically prove. The type system already ensures that this evidence constitutes a correct proof, and so separate verification is unnecessary[1].

In this paper, we describe a general framework in Agda for describing *evidence-producing* proof-search programs, and use it to develop a combinator language in the style of property-based testing libraries such as SmallCheck (Runciman et al. 2008) and QuickCheck (Claessen and Hughes 2000). To demonstrate the versatility of our approach, we shall also prove some classic verification properties about concurrent programs, by defining a simple model checker within our framework, and using it for exhaustive state space analysis.

TyDe'16, September 18, 2016, Nara, Japan
© 2016 ACM. 978-1-4503-4435-7/16/09...$15.00
http://dx.doi.org/10.1145/2976022.2976030

2. Decision Procedures

The simplest example of a proof-search procedure in Agda is the humble Dec type.

```
data Dec (P : Set) : Set where
  yes : P   → Dec P
  no  : ¬ P → Dec P
```

A program given the type Dec P amounts to an evidence-producing *decision procedure* for the proposition P. Dec, or a type much like it, is commonly used as a replacement for `Bool` in a dependently-typed setting.

Any *complete* proof-search procedure can be encoded as a Dec. For example, equality of natural numbers is fully decidable:

```
_≐_ : (m : ℕ) → (n : ℕ) → Dec (m ≡ n)
zero  ≐ zero   = yes refl
suc m ≐ suc n  with m ≐ n
... | yes p    = yes (cong suc p)
... | no prf   = no (prf ∘ cong pred)
zero  ≐ suc n  = no λ()
suc m ≐ zero   = no λ()
```

A decider for ordering of natural numbers can be defined along the same lines:

```
_≤?_ : (m : ℕ) → (n : ℕ) → Dec (m ≤ n)
```

Or a decision procedure that, drawing from the standard library, checks if a number is the greatest common divisor of two others:

```
gcd? : (m n d : ℕ) → Dec (GCD m n d)
```

These deciders are useful not just for programming, but also as a simple proof automation tool. We can use each one as a miniature

[1] Assuming, of course, that the type checker is correct

tactic, simply by using dependent types to extract the evidence from the value, whichever way it goes:

```
DecType : ∀{P} → Dec P → Set
DecType {P} (yes _) = P
DecType {P} (no  _) = ¬ P
dec-evidence : ∀{P} → (d : Dec P) → DecType d
dec-evidence (yes p) = p
dec-evidence (no  p) = p
```

A simple example is ordering proofs for naturals, where the proof object grows linearly with the left-hand number. By using our previously-defined decider, we can get Agda to compute this large proof object for us:

```
ex₁ : 135 ≤ 341
ex₁ = dec-evidence (135 ≤? 341)
```

A more interesting example is a proof that 6 and 35 are coprime:

```
ex₂ : GCD 6 35 1
ex₂ = dec-evidence (gcd? 6 35 1)
```

Problems begin to set in, however, when we wish to write proof-search procedures for propositions that have no complete proof-search algorithm. For example, there is no decider to determine if a (potentially infinite) colist has finite length:

```
postulate halt : ∀{A : Set}{xs : Colist A} → Dec (Finite xs)
- Impossible.
```

This is due to a simple reduction to the halting problem: Let xs be a colist containing each successive configuration of an arbitrary Turing Machine as it executes. Then, deciding if xs is finite is equivalent to deciding the halting problem, and therefore such a decider cannot exist.

3. Half a Decider

While predicates such as Finite are not decidable, there are cases where a proof can be found automatically (i.e. when the list is finite). So, a proof search procedure is really *half* of a decider, where failure of the proof search *does not* indicate that the proposition is false — merely that a proof cannot be found. We call these procedures *hemideciders* (as opposed to *semideciders*, which will always find a proof if it exists but may loop forever otherwise, a *hemidecider* always terminates but is never obligated to find a proof).

We can define hemideciders by taking the Dec type and removing the negative evidence:

```
data HDec (P : Set) : Set where
  success : P →  HDec P
  failed  :      HDec P
```

With this definition, success e corresponds to the yes case of Dec, where evidence is provided in favour of the proposition being tested. A result of failed simply means that such evidence was not found, *not* that it does not exist.

We could write similar proof-reflection functions as for Dec for this new definition, as follows:

```
HDecType : ∀ {P} → HDec P → Set
HDecType {P} (success _) = P
HDecType {P} (failed)    = ⊤

evidence : ∀ {P} → (p : HDec P) → HDecType p
evidence (success p) = p
evidence (failed)    = tt
```

However this representation precludes a powerful performance optimisation. Note that the definition of HDecType merely checks whether a proof was found — it discards the evidence itself.

When type checking a use of evidence the type checker must run the proof search to either success or failed to determine the result of HDecType. Importantly, the HDec only needs to be evaluated far enough to discover the constructor (success or failed) and no further. This means that we should make determining which constructor is returned as efficient as possible.

Consider the case where we compose multiple hemideciders monadically (as in Section 3.1), where all constituant hemideciders must find a proof in order for their composition to find a proof. In this case we must traverse the search space and build up a proof object (which may be very large), at every step checking that all the substeps returned success. We cannot determine the final result until after we have created the entire proof object.

We solve this problem by separating the single bit that tells us if a proof was found from the procedure that constructs the evidence:

```
record HDec (P : Set) : Set where
  constructor hd
  field found : Bool
  field proof : T found → P
```

This definition includes a single boolean, found, that signifies that a proof was found, and a function proof that produces evidence for the given proposition, given a witness that the boolean found is true[2].

Now, in order to determine if the proof was found, we merely need to examine the found field, and only construct the proof object when needed with proof.

```
HDecType : ∀ {P} → HDec P → Set
HDecType {P} (hd true  pr) = P
HDecType {P} (hd false pr) = ⊤
evidence : ∀ {P} → (p : HDec P) → HDecType p
evidence (hd true  p) = p tt
evidence (hd false _) = tt
```

Of course, we can continue to use the stronger Dec as a HDec by throwing away the refutation:

```
fromDec : ∀ {A : Set} → Dec A → HDec A
fromDec (yes p) = hd true (const p)
fromDec (no  p) = hd false ⊥-elim
```

Just as QuickCheck includes a `Testable` type class to automatically convert from `Bool` (among other things) to `Property`, we define a record type and use Agda's *instance arguments* mechanism (Devriese and Piessens 2011) to similarly convert Dec to HDec automatically:

```
record IsSearch (t : Set → Set) : Set₁ where
  constructor is-hdec
  field toHDec : (∀ {p} → t p → HDec p)

instance
  HDecIsSearch = is-hdec id
  DecIsSearch  = is-hdec fromDec

open IsSearch {{ ... }}
```

[2] In Agda, T is an indexed type that is inhabited iff its index is true

Here, the open keyword is used to bring the function toHDec in scope, where the specific implementation is chosen by automatic search through all bindings declared in an instance block.

3.1 Tactical Applicatives

As our HDec type is similar to Maybe, it admits similar Monad, Applicative and Alternative instances[3]. These instances give us a basic framework to combine proof-search procedures, reminiscent of *tacticals* from LCF or HOL-style theorem provers (Slind and Norrish 2008).

Trying Many Alternative Approaches To give an instance of Alternative, we must make use of the following lemma from the standard library, which takes a witness of disjunction and produces a disjunction of witnesses:

```
∨-T : ∀{a b} → T (a ∨ b) → T a ⊎ T b
```

The Alternative instance consists of a proof-search procedure that always fails; and a function combines two hemideciders for the same proposition by trying one then the other, similarly to the ORELSE tactical in HOL.

```
∅ : ∀{X} → HDec X
∅ = hd false ⊥-elim

_|_ : ∀{X} → HDec X → HDec X → HDec X
_|_ {X} (hd f₁ a) (hd f₂ b) = hd (f₁ ∨ f₂) (choose ∘ ∨-T)
  where
    choose : T f₁ ⊎ T f₂ → X
    choose (inj₁ t₁) = a t₁
    choose (inj₂ t₂) = b t₂
```

Explicit Chaining with Monad The Monad instance allows one to chain proof-searches together, each one building on the results of the previous, similarly to the THEN tactical in HOL. If any of the individual searches fail, the whole search fails:

```
_»=_ : ∀{P Q}
  → HDec P
  → (P → HDec Q)
  → HDec Q
_»=_ (hd true  p) f = f (p tt)
_»=_ (hd false p) f = hd false ⊥-elim

return : ∀{P} → P → HDec P
return = hd true ∘ const
```

The explicit dependency in the Monad bind function requires us to construct the evidence of the left hand proposition (by calling the function p), before executing the right-hand proof-search procedure. As constructing this evidence is costly, substantial performance improvements can be made by relying on the Applicative instance wherever possible, as applicative functors remove this dependency.

Efficient Composition with Applicative The popular Applicative idiom (McBride and Paterson 2008) gives us a way to apply a proof-search procedure for each subgoal in a rule, similarly to the THENL tactical in HOL. Crucially, and unlike Monad, it allows us to compose hemideciders without constructing any proof objects. Analogously to Alternative, we use a lemma from the standard library, this time regarding conjunction:

```
∧-T : ∀{a b} → T (a ∧ b) → T a × T b
```

The definition of ⊛ then proceeds smoothly. Note that unlike the Monad instance, we never apply the expensive evidence-producing function a:

```
_⊛_ : ∀{A B}
  → HDec (A → B)
  → HDec A → HDec B
_⊛_ {A}{B} (hd f₁ a) (hd f₂ b)
  = hd (f₁ ∧ f₂) (conclude ∘ ∧-T)
  where
    conclude : T f₁ × T f₂ → B
    conclude (t₁ , t₂) = a t₁ (b t₂)
```

The Applicative instance allows us to use the *idiom brackets* notation of McBride and Paterson (2008), as implemented in Agda 2.5.1. Idiom brackets allow expressions of the form $(f \langle\$\rangle a \circledast b \circledast g\ c)$ to be written more succinctly as $(\!|f\ a\ b\ (g\ c)|\!)$.

Putting Our Instances to Work With this framework for defining proof-searches, we can write some combinators for common logical connectives, like conjunction and disjunction. Conjunction relies on the Applicative instance, as both conjuncts must be proved:

```
_and_ : ∀{A B}{hdec₁ hdec₂}
  → {| P₁ : IsSearch hdec₁ |} → hdec₁ A
  → {| P₂ : IsSearch hdec₂ |} → hdec₂ B
  → HDec (A × B)
a and b = (| toHDec a , toHDec b |)
```

Disjunction conversely relies on the Alternative instance, as only one of the hemideciders must find a proof:

```
_or_ : ∀{A B}{hdec₁ hdec₂}
  → {| P₁ : IsSearch hdec₁ |} → hdec₁ A
  → {| P₂ : IsSearch hdec₂ |} → hdec₂ B
  → HDec (A ⊎ B)
a or b = (| inj₁ (toHDec a) |) | (| inj₂ (toHDec b) |)
```

To define an implication connective, we search for a refutation of the antecedent or a proof of the conclusion, exploiting the familiar fact that $(\neg a \vee b) \to (a \to b)$:

```
impl : ∀{A B : Set}{hdec₁ hdec₂}
  → {| P₁ : IsSearch hdec₁ |} → hdec₁ (¬ A)
  → {| P₂ : IsSearch hdec₂ |} → hdec₂ B
  → HDec (A → B)
impl a b = (| contr (toHDec a) |) | (| const (toHDec b) |)
  where
    contr : ∀{A B : Set} → ¬ A → A → B
    contr ¬a a = ⊥-elim (¬a a)
```

We can even make our hemideciders recursive, to search through data structures. For example, the Any and All types state that a given property is true for any or all elements, respectively, of a given list. The combinators for these types are quite straightforward. For any, we search for a proof of the property for the head, and recurse into the tail if the search fails.

```
any : ∀ {X : Set}{p : X → Set}{hdec}
  → {| P : IsSearch hdec |}
```

[3] As the standard library lacks an Alternative module, we use the equivalent RawMonadPlus module instead.

```
    → (xs : List X)
    → ((x : X) → hdec (p x))
    → HDec (Any p xs)
any [] f       = ∅
any (x :: xs) f = ⦇ here (toHDec (f x)) ⦈
                | ⦇ there (any xs f)    ⦈
```

For the definition of all, we use the Applicative instance rather than the Alternative, and search for a proof of the property for each element of the list:

```
all : ∀ {X : Set}{p : X → Set}{hdec}
    → ⦃ P : IsSearch hdec ⦄
    → (xs : List X)
    → ((x : X) → hdec (p x))
    → HDec (All p xs)
all [] f       = pure []
all (x :: xs) f = ⦇ toHDec (f x) :: all xs f ⦈
```

4. Property Based Testing

Our proof search procedures are particularly suited to existence proofs: finding a solution to an equation, or a counterexample to a conjecture, or even the presence of a bug in a program. Property-based testing libraries such as QuickCheck (Claessen and Hughes 2000) and SmallCheck (Runciman et al. 2008) are examples of such existence searches already in widespread use. More recently, Paraskevopoulou et al. (2015) designed a *verified* a property based testing library in Coq, by associating a semantics to checkers and showing that the tests do indeed check the properties we hope they do. Our typed approach allows for a much more direct encoding of properties (see Section 4.1), as the semantics of the checker show up in the checker's type.

Unlike Paraskevopoulou et al. (2015), which focuses on pseudo-random candidate generation, we merely define a generator to be a coinductive stream of test candidates. Exactly *how* these candidates are generated is left up to the producer of the stream. Bulwahn (2012) examines the relative performance of the various generation mechanisms in a theorem-proving context. They found that exhaustive enumeration found slightly more counterexamples to common theorems than selecting inputs randomly, so we shall stick to exhaustive enumerators for this development.

```
Gen = Stream
```

In turn, Stream is defined in the Agda standard library as:

```
data Stream (A : Set) : Set where
  _::_ : (x : A) (xs : ∞ (Stream A)) → Stream A
```

In Agda idiom, the type $\infty\ P$ refers to a *suspended computation* with result type P. Such thunks can be created with the ♯ operator, and forced into values with the ♭ operator. Agda's termination checker works a double shift as a productivity checker for corecursion — just as it checks that all recursion is structural, it ensures that all corecursion is guarded.

Many useful functions for working with streams are available from the standard library also: map, head, tail, take, and so on. Of particular use is iterate, as it allows for a natural way to define enumerators for types. For example, the enumerator for all natural numbers:

```
instance nats₀ = S.iterate suc 0
```

To avoid confusing these functions with the equivalent List functions, we have qualified them in this development by the letter S. Similarly, for the equivalent functions for length-indexed vectors (called in Agda Vec), we use the qualifier V.

4.1 Properties

We define a Property to be a proof search procedure parameterised by a search depth:

```
Property : Set → Set
Property P = (depth : ℕ) → HDec P
```

The `Property` type used in property-based testing libraries would be more accurately represented as a search for a counterexample ($\mathbb{N} \to$ HDec ($\neg P$)) than our definition above. In an intuitionistic logic such as Agda's type theory, however, the additional negation can prove rather inconvenient. A pessimistic search for a counterexample precludes the use of these techniques for positive propositions, such as constructive existence proofs. As we do not want our framework to be restricted merely to counterexamples, we reframe the language to be more optimistic: a search for truth, rather than a statement to refute.

Similarly to IsSearch above, we define a "type class" to allow us to treat HDec, Dec and Property uniformly.

```
record IsProp (t : Set → Set) : Set₁ where
    constructor is-prop
    field toProp : (∀ {p} → t p → Property p)

instance
  IsSearchIsProp : ∀{S} → ⦃ P : IsSearch S ⦄ → IsProp S
  IsSearchIsProp  = is-prop (const ∘ toHDec)
  PropertyIsProp  = is-prop id

open IsProp ⦃ ... ⦄
```

To actually check a property, we simply supply the search depth, and the proof is returned to us, if found:

```
PropType : ∀{P} → ℕ → Property P → Set
PropType d p = HDecType (p d)

check : ∀{P} → (d : ℕ) → (p : Property P) → PropType d p
check d p = evidence (p d)
```

4.2 Existence Search

Now we have all the ingredients in place to define a property combinator for existential quantifiers, by testing every element of the given generator until a proof is found or a depth limit is reached. We use our existing any combinator to search through the first d candidates from the stream.

```
exists : ∀ {X}{p : X → Set}{prop}
    → ⦃ g : Gen X ⦄
    → ⦃ P : IsProp prop ⦄
    → ((x : X) → prop (p x))
    → Property (∃ p)
exists {X}{p} ⦃ s ⦄ f d
  = let xs = V.toList $ S.take d s
    in weaken ⟨$⟩ any xs (λ x → toProp (f x) d )
  where
    weaken : ∀{ls} → Any p ls → ∃ p
```

```
weaken (here  x) = _ , x
weaken (there y) = weaken y
```

Now we can try some more interesting applications. For example, it may seem (at first glance) that the following conjecture is true:

$$\gcd(i^2 + 7, (i+1)^2 + 7) = 1$$

Certainly, for small values of i, it seems true enough. Let us put our framework to work, and see if it can find a counterexample:

```
ex₃ : Property (∃ λ i → ¬ GCD (i ² + 7) ((i + 1)² + 7) 1)
ex₃ = exists λ i → ¬? (gcd? (i ² + 7) ((i + 1)² + 7) 1)
```

Indeed, evaluating `check 20 ex₃` gives us a counterexample: $(14, \langle \textit{large proof term} \rangle)$, which indicates that our conjecture is falsified when $i = 14$.

We can even extract the proof from the property, as we did with Dec, and save ourselves the trouble of figuring out how to prove it by hand:

```
lemma : ∃ λ i → ¬ GCD (i ² + 7) ((i + 1)² + 7) 1
lemma = check 20 ex₃
```

4.3 Some Standard Stream Functions

Curiously, the following very useful functions were missing from the standard library's Stream module, where they perhaps belong. Firstly, a sequence function, to convert a vector of streams to a stream of vectors:

```
readMulti : ∀{A}{l}
  → Vec (Stream A) l → (Vec A l × Vec (Stream A) l)
readMulti []       = [] , []
readMulti (x :: v) with readMulti v
... | as , ss      = (S.head x :: as) , (S.tail x :: ss)

sequence : ∀{A}{l}
  → Vec (Stream A) l → Stream (Vec A l)
sequence vs with readMulti vs
... | as , ss = as :: ♯ sequence ss
```

Secondly, a concat function, which takes a stream of (non-empty) lists and yields each element of each list individually:

```
concat : ∀ {A} → Stream (List⁺ A) → Stream A
concat ((x :: [])       :: xss) = x :: ♯ concat (♭ xss)
concat ((x :: y :: xs)  :: xss) = x :: ♯ concat ((y :: xs) :: xss)
```

Finally, a function to fairly merge a group of streams into a single stream:

```
multiplex : ∀{A}{l} → Vec (Stream A) (suc l) → Stream A
multiplex vs = concat (S.map fromVec (sequence vs))
```

4.4 Generating Regular Types

Now that we have defined a combinator language for Property, and plugged some of the holes in the standard library's Stream module, we now turn to the remaining unconquered territory that property-based testing has charted for us. We need a language to build generators for common data types.

Sum types are rather stress-free, accomplished easily by fairly merging the two streams:

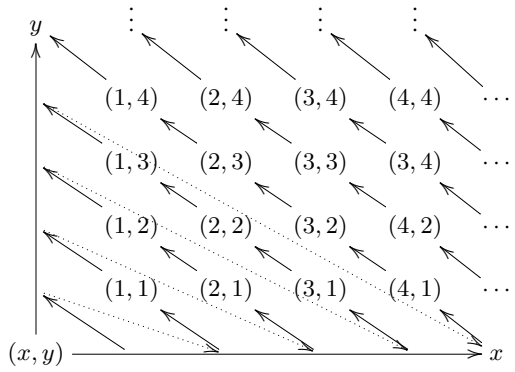


Figure 1. Cantor's zig-zag traversal, for $\mathbb{N} \times \mathbb{N}$

```
instance
  sums : ∀{A B} ⦃ a : Gen A ⦄ ⦃ b : Gen B ⦄ → Gen (A ⊎ B)
  sums ⦃ a ⦄ ⦃ b ⦄ = multiplex
    ( S.map inj₁ a :: S.map inj₂ b :: [])
```

Products, or their more general dependent counterparts, Σ-types, are slightly more intricate. It is easy to construct a two-dimensional plane of products from the two generators:

```
Σ-plane : ∀{A}{B : A → Set}
  → Gen A → ((x : A) → Gen (B x))
  → Gen (Gen (Σ A B))
Σ-plane as bs = S.map (λ a → S.map (_ ,_ a) (bs a)) as
```

But then there are an infinite number of possible lines one could draw through this plane to produce a single stream. We could draw a straight diagonal, which is easy, but this does not preserve the exhaustivity of the original generators.

Instead, we borrow a trick originally from Georg Cantor (Ewald 1996), and proceed in a "zig-zag" fashion, covering the whole plane (See Figure 1).

The nth diagonal component of the traversal can be expressed as all coordinates where the sum of the x and y components is n. Therefore, we can simply generate each diagonal component and concatenate them to produce our line.

To store each "line" in the plane, we use Agda's non-empty List type List⁺, which convinces Agda that our traversal is always productive.

```
zig : ∀ {A} → List⁺ (Gen A)
  → Gen (Gen A) → Gen A
zag : ∀ {A} → List⁺ A → List⁺ (Gen A)
  → Gen (Gen A) → Gen A
zig xs ls  = zag (S.head ⟨$⟩ xs) (S.tail ⟨$⟩ xs) ls

zag (x :: [])       zs (l :: ls)  = x :: ♯ zig (l :: toList zs) (♭ ls)
zag (x :: y :: xs)  zs ls         = x :: ♯ zag (y :: xs) zs ls
```

This traversal allows us to define a function to flatten a plane into a line:

```
flatten : ∀ {A} → Gen (Gen A) → Gen A
flatten (l :: ls) = zig (l :: []) (♭ ls)
```

Now, using this function, we can define a generator for Σ-types, and their specialisation to familiar product types:

```
_also_ : ∀{A}{B : A → Set}
       → Gen A → ((x : A) → Gen (B x)) → Gen (Σ A B)
as also bs = flatten (Σ-plane as bs)

instance
  pairs : ∀{A B}⦃ a : Gen A ⦄ ⦃ b : Gen B ⦄ → Gen (A × B)
  pairs ⦃ a ⦄ ⦃ b ⦄ = a also (λ x → b)
```

A nice example of this generator is searching for solutions to diophantine equations, such as Pythagoras' theorem:

```
triad : Property _
triad = exists λ ⁅ (z , x , y) →
  ¬? (x ≐ 0) and ¬? (y ≐ 0) and ((x ² + y ²) ≐ z ²) ⁆
```

Checking triad not only gives the expected triad $(5, 4, 3)$, but also provides a proof[4] that these numbers satisfy the equation $x^2 + y^2 = z^2$.

4.5 Lists

As our regular types are composed of sums and products, these building blocks can be used to make generators for many common data types. Using products, we can define a generator for vectors of a fixed length:

```
vecs : ∀{A} n ⦃ g : Gen A ⦄ → Gen (Vec A n)
vecs zero          = S.repeat []
vecs (suc n) ⦃ g ⦄ = S.map (λ ⁅ (a , b) → a :: b ⁆) $
  pairs ⦃ g ⦄ ⦃ vecs n ⦃ g ⦄ ⦄
```

Furthermore, we can simply generate vectors of *every length*, and traverse them once again in a zig-zag fashion, to give us a generator for lists:

```
lists : ∀{A} → ⦃ g : Gen A ⦄ → Gen (List A)
lists = flatten $
  S.map (λ n → S.map V.toList (vecs n)) nats₀
```

5. Model Checking

One of the biggest sources of surprising counterexamples is in the verification of programs with large amounts of nondeterminism, such as critical section solutions used in concurrency.

The most common automated technique for verifying such programs is *model checking*, where the program is typically modelled by some automaton, such as a Büchi Automaton (Thomas 1990), a Kripke structure (Kripke 1963), or other transition diagrams, such as those used by Floyd (Floyd 1967). The state space of this automaton is searched for violations of the desired correctness conditions. If no violations are found, and the state space is exhaustively searched, the program is correct.

In this section, we shall embed a basic model checker for a subset of CTL (Clarke et al. 1986) within Agda, and then put our existing hemidecisions framework to work, verifying the correctness of common critical section solutions.

We intentionally do not make use of more sophisticated model checking techniques, such as LTL model checking using Büchi Automata, as bridging the distance between what these checkers *do* and what they *prove* requires a fair bit of non-trivial reasoning. While these methods are not incompatible with our framework, designing an evidence-producing algorithm for such model checking is much more involved.

[4] In this case, the proof term is merely refl

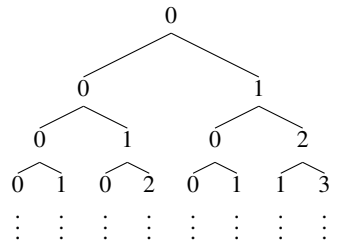


Figure 2. The computation tree of **HiHorse** || **LoRoad**

5.1 Transition Diagrams

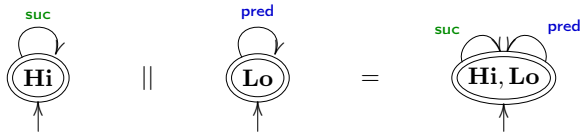
A transition diagram is essentially a nondeterministic automaton, with a set of state labels L, an initial state label I, some shared state Σ and a transition relation $\delta : (L \times \Sigma) \times (L \times \Sigma)$, which we model as a function from an initial state $L \times \Sigma$ to a list of successor states.

```
record Diagram (L : Set)(Σ : Set) : Set₁ where
  constructor td
  field δ : L × Σ → List (L × Σ)
        I : L
```

The parallel composition of two diagrams represents a nondeterministic interleaving of every transition in either of the original processes. That is, parallel composition is simply the product of the two automata. An individual process may interfere with the behaviour of the other, as they both manipulate the same shared state Σ.

```
_‖_ : ∀{L₁ L₂}{Σ}
    → Diagram L₁ Σ → Diagram L₂ Σ
    → Diagram (L₁ × L₂) Σ
(td δ₁ i₁) ‖ (td δ₂ i₂) = td δ (i₁ , i₂)
where
  δ = (λ ⁅ ((ℓ₁ , ℓ₂) , σ) →
    map (λ ⁅ (ℓ₁′ , σ′) → (ℓ₁′ , ℓ₂ ) , σ′ ⁆) (δ₁ (ℓ₁ , σ)) ++
    map (λ ⁅ (ℓ₂′ , σ′) → (ℓ₁  , ℓ₂′) , σ′ ⁆) (δ₂ (ℓ₂ , σ)) ⁆)
```

An illustrative example is the composition of the two processes **HiHorse** and **LoRoad**, who share a single number between them. **HiHorse** will always increase the number, but **LoRoad** always decreases it.



Note that the parallel composition repeatedly makes a nondeterministic choice between increasing and decreasing the number, reflecting the nondeterministic interleaving of the two original processes.

```
HiHorse LoRoad : Diagram ⊤ ℕ
HiHorse = let δ = λ ⁅ (ℓ , σ) → [ (ℓ , suc σ )]⁆
          in  td δ tt
LoRoad  = let δ = λ ⁅ (ℓ , σ) → [ (ℓ , pred σ )]⁆
          in  td δ tt
```

$$
\begin{array}{lllll}
\text{guards} & g & : & \Sigma \to \mathbb{B} \\
\text{formulae} & \phi, \psi & ::= & \langle g \rangle \mid \mathbf{Completed} \\
& & \mid & \phi \wedge \psi \mid \top \\
& & \mid & \mathbf{A}(\phi\ \mathbf{U}\ \psi) \\
& & \mid & \mathbf{E}(\phi\ \mathbf{U}\ \psi) \\
& & \mid & \mathbf{AF}\ \phi \mid \mathbf{EF}\ \phi \\
& & \mid & \mathbf{AG}\ \phi \mid \mathbf{EG}\ \phi
\end{array}
$$

Figure 3. Grammar of our CTL subset

5.2 Computation Tree Logic

Having found a suitable method to define processes, we now wish to perform analysis on their behaviour. We will state our program properties in a subset of the Computation Tree Logic of Clarke et al. (1986), which includes a series of modal operators to describe properties of processes' *computation trees*. This tree is the (potentially infinitary) unfolding of the transition diagram, and takes the form of a corecursive rose tree:

```
module CTL(L Σ : Set) where

data CT : Set where
  At : (L × Σ) → ∞ (List CT) → CT
```

Each path through the tree is a valid trace of the original program. For example, the computation tree of **HiHorse** || **LoRoad** is a binary tree where all paths extend infinitely (See Figure 2). Given a value for the starting shared state, we construct the computation tree of a given diagram as follows:

```
model : Diagram L Σ → Σ → CT
model (td δ I) σ = follow (I , σ)
  where
    follow  : (L × Σ) → CT
    followAll : List (L × Σ) → List CT
    follow  σ          = At σ (♯ followAll (δ σ))
    followAll (σ :: σs) = follow σ :: followAll σs
    followAll []       = []
```

The syntax of our CTL fragment is given in Figure 3. In Agda, our model checker will examine the computation tree exhaustively up to a finite depth. A formula therefore is defined as a property of a computation tree with a depth limit:

```
Formula = (depth : ℕ) → (tree : CT) → Set
```

We say a tree satisfies a formula (written $t \models \phi$) if there exists a depth d_0 for which the property holds for all depths $d \geq d_0$.

```
data _⊨_ (m : CT)(φ : Formula) : Set where
  satisfies : ∀ d₀ → (∀ { d } → d₀ ≤′ d → φ d m) → m ⊨ φ
```

In our development, we are only concerned with formulae that are not falsifiable by increasing the search depth. We call this property *depth-invariance*.

```
record DepthInv (φ : Formula) : Set where
  constructor di
  field proof : (∀{n}{m} → φ n m → φ (suc n) m)
```

If a depth invariant formula holds for a particular depth, it follows that the tree satisfies the formula in general:

```
di-⊨ : ∀{n}{φ}{m} ⦃ d : DepthInv φ ⦄
  → φ n m → m ⊨ φ
di-⊨ {n}{φ}{m} ⦃ di d ⦄ p = satisfies n (λ q → di-≤ p q)
    where
  di-≤ : ∀{n n′} → φ n m → n ≤′ n′ → φ n′ m
  di-≤ p ≤′-refl      = p
  di-≤ p (≤′-step l)  = d (di-≤ p l)
```

We make the depth-invariance constraint an instance argument as it can be automatically proven by using syntax-directed rules, which we will prove along with each formula.

The simplest example of a formula is the trivially true formula $\top$. As this formula is never falsifiable, it is easily depth-invariant:

```
data True : Formula where
  tt : ∀{n}{m} → True n m
instance
  True-di : DepthInv True
  True-di = di (const tt)
```

The formula $\langle g \rangle$ is true when the current state satisfies the guard g.

```
data ⟨_⟩ (g : ⦃ σ : Σ ⦄ → ⦃ ℓ : L ⦄ → Set) : Formula where
  here : ∀{σ}{ℓ}{ms}{d} →
    g ⦃ σ ⦄ ⦃ ℓ ⦄ → ⟨ g ⟩ d (At (ℓ , σ) ms)
```

We make the parameters to the guard instance arguments to allow guards about state projections to be written more succinctly, as in Section 7.1.

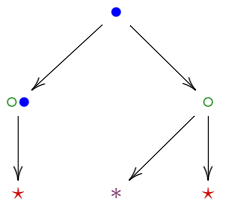
Once again depth-invariance is easily established as this formula ignores the depth:

```
instance
  ⟨⟩-di : ∀{p : ⦃ σ : Σ ⦄ ⦃ ℓ : L ⦄ → Set}
    → DepthInv ⟨ p ⟩
  ⟨⟩-di {p} = di proof
    where
      proof : ∀{n}{m} → ⟨ p ⟩ n m → ⟨ p ⟩ (suc n) m
      proof (here x) = here x
```

Conjunction is defined predictably, requiring a proof of both conjuncts, and is depth-invariant if the two conjuncts are depth-invariant.

```
data _∧′_ (φ ψ : Formula ) : Formula where
    _,_ : ∀{n}{m} → φ n m → ψ n m → (φ ∧′ ψ) n m
instance
  ∧′-di : ∀{φ ψ} ⦃ p : DepthInv φ ⦄ ⦃ q : DepthInv ψ ⦄
    → DepthInv (φ ∧′ ψ)
  ∧′-di ⦃ di p ⦄ ⦃ di q ⦄ = di λ { (a , b) → p a , q b }
```

To illustrate the semantics of the temporal operators, we use the following example:



The temporal operator $\mathbf{A}(\phi\ \mathbf{U}\ \psi)$ is true when, for all **A**ll paths in the tree, ϕ is true **U**ntil ψ is true. For example, the tree above satisfies $\mathbf{A}(\bullet\ \mathbf{U}\ \circ)$. In Agda we define it using the standard library's All data type to recursively apply the formula to each successor state.

```
data A[_U_] (φ ψ : Formula) : Formula where
  here  : ∀{t}{n} → ψ n t → A[ φ U ψ ] (suc n) t
  there : ∀{σ}{ms}{n}
    → φ n (At σ ms)
    → All (A[ φ U ψ ] n) (♭ ms)
    → A[ φ U ψ ] (suc n) (At σ ms)
```

The depth-invariance proof for this operator is a straightforward induction:

```
instance
  A-di : ∀{φ ψ} ⦃ p : DepthInv φ ⦄ ⦃ q : DepthInv ψ ⦄
    → DepthInv A[ φ U ψ ]
  A-di {φ}{ψ} ⦃ di p ⦄ ⦃ di q ⦄ = di prf
    where
    prf : ∀{d}{t} → A[ φ U ψ ] d t → A[ φ U ψ ] (suc d) t
    prf (here x)      = here  (q x)
    prf (there x xs)  = there (p x) (All.map prf xs)
```

Similarly, $\mathbf{E}(\phi\ \mathbf{U}\ \psi)$ is true if there **E**xists a path in the tree where ϕ is true **U**ntil ψ is true. For example, the tree above satisfies $\mathbf{E}(\bullet\ \mathbf{U}\ \star)$, on the left branch. In Agda, the definition is similar to $\mathbf{A}(\phi\ \mathbf{U}\ \psi)$, except All is replaced by Any.

```
data E[_U_] (φ ψ : Formula) : Formula where
  here  : ∀{t}{n} → ψ n t → E[ φ U ψ ] (suc n) t
  there : ∀{σ}{ms}{n}
    → φ n (At σ ms)
    → Any (E[ φ U ψ ] n) (♭ ms)
    → E[ φ U ψ ] (suc n) (At σ ms)
instance
  E-di : ∀{φ ψ} ⦃ p : DepthInv φ ⦄ ⦃ q : DepthInv ψ ⦄
    → DepthInv E[ φ U ψ ]
  E-di {φ}{ψ} ⦃ di p ⦄ ⦃ di q ⦄ = di prf
    where
    prf : ∀{d}{t} → E[ φ U ψ ] d t → E[ φ U ψ ] (suc d) t
    prf (here x)      = here  (q x)
    prf (there x xs)  = there (p x) (Any.map prf xs)
```

Using these formulae as building blocks, we can define the **F**inally operators, which state that a formula will be eventually satisfied, either in all paths (**AF**), or a single path (**EF**).

```
AF EF : Formula → Formula
AF φ = A[ True U φ ]
EF φ = E[ True U φ ]
```

In the example tree above, **AF** $\langle\circ\rangle$ is true, as all paths include a $\circ$-state; and **EF** $\langle*\rangle$ is true as there exists a path which includes a $*$-state.

Unfortunately for us, the **G**lobally operators, which make assertions for *every* state in a path, cannot be defined straightforwardly in terms of our existing operators. In fact, it is impossible for our model checking approach to find a proof for these operators in general: If a path extends infinitely, examining the path up to a finite depth does not help us to prove the property for the entire path. Model checking algorithms exist which handle infinite paths, but they all work on the automata directly rather than the computation tree. Instead, we shall examine only the case where paths are *finite* (i.e. where the program is terminating). This allows us to exhaustively analyse all behaviours merely by examining the computation tree. We introduce a new sort of formula, **Completed**, which states that the current state has no successor states.

```
data Completed : Formula where
  completed : ∀{σ}{n}{ms}
    → ♭ ms ≡ []
    → Completed n (At σ ms)
instance
  Completed-di : DepthInv Completed
  Completed-di = di prf
    where
    prf : ∀{d}{t} → Completed d t → Completed (suc d) t
    prf (completed p) = completed p
```

With this we can define a stronger form of the **G**lobally operators than the traditional CTL, as we require that the relevant paths are finite:

```
AG EG : Formula → Formula
AG  φ = A[  φ U φ ∧′ Completed ]
EG  φ = E[  φ U φ ∧′ Completed ]
```

5.3 Proof-Search for CTL

We can now put our proof-search framework to use, writing HDec combinators for each of our CTL operators. A *model-checking* procedure for a given formula is defined as a HDec for the formula, parameterised by a computation tree and a depth limit:

```
MC : Formula → Set
MC φ = (t : CT)(d : ℕ) → HDec (φ d t)
```

The proof-search combinator for $\langle g\rangle$-formulae is called now. It lifts a HDec on states into a MC procedure:

```
now : ∀{ g : ⦇ σ : Σ ⦈ → ⦇ ℓ : L ⦈ → Set }{hdec}
  → ⦇ P : IsSearch hdec ⦈
  → (⦇ σ : Σ ⦈ → ⦇ ℓ : L ⦈ → hdec (g ⦇ σ ⦈ ⦇ ℓ ⦈) )
  → MC ⟨ g ⟩
now p (At (ℓ , σ) ms) _ = (| here (toHDec (p ⦇ σ ⦈ ⦇ ℓ ⦈))) |)
```

Another simple combinator is completed? which checks if a state has any successors.

```
completed? : MC Completed
completed? (At σ ms) _ = completed ⟨$⟩ empty? (♭ ms)
  where
    empty? : ∀{X}(n : List X) → HDec (n ≡ [])
    empty? []        = return refl
    empty? (_ :: _)  = ∅
```

Conjunction of MC procedures uses the Applicative instance just as our combinator for HDec does:

```
_and′_ : ∀ {φ ψ} → MC φ → MC ψ → MC (φ ∧′ ψ)
_and′_ a b m n = (| a m n , b m n |)
```

Our proof-search procedures for $\mathbf{A}(\phi\ \mathbf{U}\ \psi)$ and $\mathbf{E}(\phi\ \mathbf{U}\ \psi)$ first attempt to find a proof of ψ at the current state, and, if that fails, they will search for a proof for ϕ then recursively descend the successor states, requiring that all (for **A**) or any (for **E**) of the successor states in turn satisfy $\mathbf{A}(\phi\ \mathbf{U}\ \psi)$.

```
a-u : ∀{φ ψ} → MC φ → MC ψ → MC A[ φ U ψ ]
a-u _  _  _  zero  = ∅
a-u p₁ p₂ t@(At σ ms) (suc n) = (| here (p₂ t n) |)
                              | (| there (p₁ t n) rest |)
```

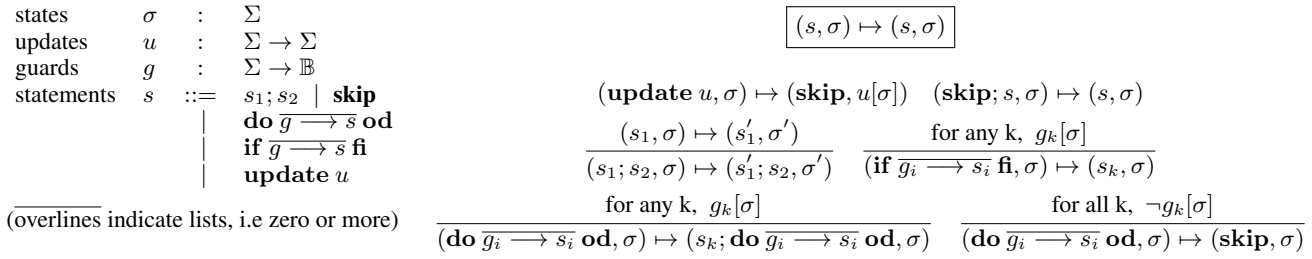


Figure 4. The Guarded Command Language: Syntax and Semantics

```
  where rest = all (♭ ms) λ m → a-u p1 p2 m n

e-u : ∀{φ ψ} → MC φ → MC ψ → MC E[ φ U ψ ]
e-u _ _ zero = ∅
e-u p1 p2 t@(At σ ms) (suc n) = ⟪ here (p2 t n) ⟫
                              | ⟪ there (p1 t n) rest ⟫
  where rest = any (♭ ms) λ m → e-u p1 p2 m n
```

Proof-searches for our derived **F** and **G** operators are easy to define using these combinators:

```
ef : ∀{φ} → MC φ → MC (EF φ)
af : ∀{φ} → MC φ → MC (AF φ)
eg : ∀{φ} → MC φ → MC (EG φ)
ag : ∀{φ} → MC φ → MC (AG φ)

ef p = e-u (λ _ _ → return tt) p
af p = a-u (λ _ _ → return tt) p
eg p = e-u p (p and′ completed?)
ag p = a-u p (p and′ completed?)
```

We now have all the combinators we need to do some simple model checking. For example, we could prove that there exists a path in our combined process **HiHorse** || **LoRoad** where the state reaches the number 10:

```
tree = model (HiHorse || LoRoad) 0

reaches10 : HDec _
reaches10 = ef (now (λ ⟪ σ ⟫ → σ ≐ 10)) tree 20
```

Evaluating reaches10 produces a proof, but only for a particular depth (namely 20). We can generalise this result to a general proof of this property by using our depth-invariance results for each combinator. As this can be deduced in a syntax-directed way, Agda's instance arguments feature is able to find the depth-invariance proof for our formula automatically.

```
proof : tree ⊨ EF ⟨ (λ ⟪ σ ⟫ → σ ≡ 10) ⟩
proof = di-⊨ (evidence reaches10)
```

This example demonstrates how easily the proofs produced by our framework can be used inside manual proof.

6. The Guarded Command Language

Technically, we have all the tools we need to analyse sophisticated programs, but encoding these programs directly as transition diagrams is cumbersome, error-prone, and difficult. Instead, we shall embed within Agda a miniature imperative language in the style of Dijkstra's Guarded Command Language (Dijkstra 1970), originally used as an example of predicate transformer semantics, but later used as a basis for modelling languages such as Promela (Holzmann 1997). The syntax and structural operational semantics of the Guarded Command Language are presented in traditional format in Figure 4. Note that nondeterminism is introduced by both **if** and **do**, as any branch with a valid guard may be chosen. Furthermore, there is the possibility of deadlock (a *stuck state*) if all of the guards in an **if** statement are false — progress is not guaranteed. A program is considered to have terminated if it evaluates to a single **skip**.

Encoding the syntax in Agda is a relatively straightforward translation from the formal definition.

```
module GCL(Σ : Set) where
mutual
  data GCL : Set where
    if_fi    : List Guard → GCL
    _⨾_      : GCL → GCL → GCL
    do_od    : List Guard → GCL
    update   : (Σ → Σ) → GCL
    skip     : GCL
  Pred = (⟪ σ : Σ ⟫ → Bool)
  data Guard : Set where
    _⟶_ : Pred → GCL → Guard
```

We also define a decider to determine whether a program successfully terminated (i.e. equal to **skip**):

```
skip? : (ℓ : GCL) → Dec (ℓ ≡ skip)
skip? if x fi       = no λ ()
skip? (σ ⨾ σ1)      = no λ ()
skip? do x od       = no λ ()
skip? (update x)    = no λ ()
skip? skip          = yes refl
```

For the operational semantics, we once again rephrase the relation as a function from a state to a list of successor states:

```
ops : (GCL × Σ) → List (GCL × Σ)
ops (skip , σ)      = []
ops (update u , σ)  = [ skip , u σ ]
ops (skip ⨾ y , σ)  = [ y , σ ]
ops (x ⨾ y , σ)     = map (λ { (x , σ') → (x ⨾ y) , σ'}) $
                      ops (x , σ)
ops (if xs fi , σ)  = map (λ { (g ⟶ x) → x , σ }) $
                      filter (λ { (g ⟶ x) → g ⟪ σ ⟫ }) xs
ops (do xs od , σ)
  with map (λ { (g ⟶ x) → (x ⨾ do xs od) , σ})
    $ filter (λ { (g ⟶ x) → g ⟪ σ ⟫ }) xs
```

```
... | []   = [ skip , σ ]
... | ys   = ys
```

Then we can convert any GCL program into a Diagram simply by using the program itself for local state labels, and the operational semantics as the transition relation:

```
⟦ _ ⟧ : GCL → Diagram GCL Σ
⟦ p ⟧ = td ops p
```

6.1 Derived Control Structures

Traditional loops are all special cases of **do**. The venerable **while** loop, for example, is just a **do** loop with a single guard:

```
while : Pred → GCL → GCL
while g x = do g ⟶ x :: [] od
```

In the same vein, **if** with a single guard is an **await** statement, which halts progress unless a certain condition is met.

```
await : Pred → GCL
await g = if g ⟶ skip :: [] fi
```

The traditional deterministic **if** ... **then** ... **else** ... structure is just an **if** statement with mutually exclusive and total guards:

```
if_then_else_ : Pred → GCL → GCL → GCL
if g then x else y = if g ⟶ x :: not g ⟶ y :: [] fi
```

By embedding this expressive language inside Agda, encoding existing algorithms as models for our checker is much more straightforward.

7. Critical Section Solutions

One of the most common example use-cases for model checkers is concurrency algorithms, particularly so-called "critical section" solutions. A *critical section* is a portion of a process execution that must not be intefered with by any other process. Solving this problem requires some finesse, as interference is common in any concurrent program with shared state. The most common solutions ensure *synchronisation* of all processes such that, at any time, at most one process is executing its critical section. This property is called *mutual exclusion*. There are a number of algorithms that solve this problem, each with varying benefits and drawbacks. Two other important desiderata for a critical section solutions are:

Starvation freedom If a process desires entry into its critical section, it will eventually gain entry.

Absence of deadlock The parallel composition of all processes cannot reach a stuck state.

These properties are simple enough to be checked automatically by a model checker, but are difficult to show by hand, requiring at least a quadratic number of proof obligations relative to the total number of transitions in each process (Owicki and Gries 1976). Furthermore, it is very easy to mistakenly write a critical section solution that appears correct, but in fact does not satisfy one of these properties. While it is possible to express large parallel compositions in our framework, we will restrict ourselves to the simpler two-process case. As additional processes are added, the number of transitions in the overall automata grows exponentially. As our model checker is rather primitive, this very quickly consumes all of Agda's resources, resulting in `stack overflows` and very hot computers.

7.1 Peterson's Algorithm

Perhaps one of the simplest critical section solutions is Peterson's algorithm (Peterson 1981). The shared state consists of three variables, intent_1, intent_2 and turn. We also keep some *ghost* variables inCS_1 and inCS_2, to make specifying properties such as mutual exclusion easier in future.

```
record State : Set where
field
  intent₁ intent₂ : Bool
  turn    : ℕ
  inCS₁ inCS₂ : Bool

open State {{ ... }}
```

If intent_n is true, then process n desires entry into its critical section. Each critical section is modeled as a single **skip** where the inCS variable is set.

```
CS₁ CS₂ : GCL
CS₁ =
  update (λ σ → record σ { inCS₁ = true  }) ·
  skip ·
  update (λ σ → record σ { inCS₁ = false })
CS₂ =
  update (λ σ → record σ { inCS₂ = true  }) ·
  skip ·
  update (λ σ → record σ { inCS₂ = false })
```

In order to reuse our equality decider $\_\doteq\_$ for boolean equality checks, we shall define some notation to weaken a Dec into a Bool.

```
⌊_⌋ : {A : Set} → Dec A → Bool
⌊ yes _ ⌋ = true
⌊ no  _ ⌋ = false
```

The code for Peterson's algorithm is given in our embedded command language in Figure 5.

The process $\mathsf{petersons}_1$ is granted access to its critical section either if $\mathsf{petersons}_2$ does not desire entry, or if $\mathsf{petersons}_2$ has already given priority to $\mathsf{petersons}_1$ by setting turn to 0.

Now, the diagram we wish to analyse is the parallel composition of the two processes:

```
petersons = ⟦ petersons₁ ⟧ ‖ ⟦ petersons₂ ⟧
```

First, we need to define the properties we wish to verify, using a decider for the standard library's T operator, which makes a Bool condition into a proposition:

```
T? : (b : Bool) → Dec (T b)
T? true  = yes tt
T? false = no id
```

Mutual exclusion states that both inCS variables cannot be simultaneously set.

```
Mutex = AG ⟨ T (not (inCS₁ ∧ inCS₂)) ⟩

mutex? : MC Mutex
mutex? = ag (now (T? _))
```

Starvation freedom is specified as the conjunction of each process reaching their critical section:

```
petersons₁ : GCL
petersons₁ =
  update (λ σ → record σ { intent₁ = true }) ·
  update (λ σ → record σ { turn = 1 }) ·
  await (not intent₂ ∨ ⌊ turn ≐ 0 ⌋) ·
  CS₁ ·
  update (λ σ → record σ {intent₁ = false })
```

```
petersons₂ : GCL
petersons₂ =
  update (λ σ → record σ { intent₂ = true }) ·
  update (λ σ → record σ { turn = 0 }) ·
  await (not intent₁ ∨ ⌊ turn ≐ 1 ⌋) ·
  CS₂ ·
  update (λ σ → record σ {intent₂ = false })
```

Figure 5. Peterson's algorithm in our embedded GCL

```
SF = AF ⟨ T inCS₁ ⟩ ∧′ AF ⟨ T inCS₂ ⟩

sf? : MC SF
sf? = af (now (T? _)) and′ af (now (T? _))
```

Deadlock freedom is implied by the stronger requirement that all possible traces lead to successful termination (i.e. all final states are skip).

```
Termination = AF ⟨ allSkip ⟩
  where
    allSkip : {‖ ℓ : GCL × GCL ‖} → Set
    allSkip {‖ ℓ = (a , b) ‖} = a ≡ skip × b ≡ skip

termination? : MC Termination
termination? = af (now (hd term? sound))
  where
    term? : {‖ ℓ : GCL × GCL ‖} → Bool
    term? {‖ a , b ‖} = ⌊ skip? a ⌋ ∧ ⌊ skip? b ⌋

    sound : {‖ ℓ : GCL × GCL ‖} → T term? → _
    sound {‖ a , b ‖} _  with skip? a | skip? b
    sound {‖ a , b ‖} _  | yes p      | yes q = p , q
    sound {‖ a , b ‖} () | yes _      | no _
    sound {‖ a , b ‖} () | no  _      | _
```

Now we can define a Property which expresses all of our desired correctness conditions and will, when evaluated, produce a proof that Peterson's algorithm is a correct solution to the critical section problem.

```
init : State
init = record
  { inCS₁  = false ; inCS₂ = false
  ; intent₁ = false ; intent₂ = false
  ; turn = 0
  }
tree = model petersons init

petersons-search : Property _
petersons-search = exists $
  (mutex? and′ sf? and′ termination?) tree
```

Once again this proof can be used directly with our depth-invariance proofs to generalise the correctness result to arbitrary depths:

```
petersons-correct : tree ⊨ Mutex ∧′ SF ∧′ Termination
petersons-correct
  = di-⊨ (proj₂ (check 1000 petersons-search))
```

7.2 Dekker's Algorithm and a Note on Fairness

Another critical section solution common in the literature is Dekker's algorithm (Dijkstra 2002), the full source of which is given in Figure 6. The key difference between the two algorithms is that Dekker's algorithm relies on *busy waiting*: rather than using the turn variable to force the other process into its critical section, here the processes may be scheduled so that a process continually spins in a loop, waiting for the other process to clear its intent variable. Solving this problem requires us to constrain our model of parallel composition to include some notion of *fairness* in scheduling. That is, if an individual process continually *can* make a transition, it will eventually be allowed to do so by the scheduler. Without this notion, a process may spin unendingly in a loop, waiting for access to its critical section, and in doing so prevent another process from giving it that access. Dekker's algorithm depends on fairness in order to be free of starvation. As our current model is not necessarily fair, it includes these nonterminating, access-starving executions. And, because of our bounded model-checking approach, we cannot explore the traces of such a model exhaustively. Sadly, this means that we can't prove the strong correctness properties we did for Peterson's algorithm. Instead, our framework is reduced to a mere sanity checker to catch mistakes:

```
dekkers = ⟦ dekkers₁ ⟧ ‖ ⟦ dekkers₂ ⟧

dekkers-check : HDec _
dekkers-check = ef (now (T? (inCS₁ ∧ inCS₂)))
  (model dekkers init) 100
```

This fails to find a proof, because we are searching for a violation of mutual exclusion and our implementation of Dekker's algorithm is correct, but there is no way to prove correctness — our framework merely failed to find a violation, it could not prove that no violations existed.

Dekker's algorithm could be properly verified by using other model checking techniques, such as LTL model checking using Büchi automata in the style of SPIN (Holzmann 1997). As fairness can be expressed as an LTL formula, it is quite straightforward to integrate fairness into the model. Sadly, as our bounded approach can only view a fixed k-unfolding of a transition system, it is unclear how fairness can be integrated into our model checker.

8. Related Work

Our basic framework bears some similarity to the *tacticals* of LCF-style theorem provers such as HOL (Slind and Norrish 2008) and Isabelle (Nipkow et al. 2002). Tactic languages such as LTac (The Coq development team 2004) and MTac (Ziliani et al. 2013) also include similar features. The Coq extension `ssreflect`

```
dekkers₁ : GCL
dekkers₁ =
  update (λ σ → record σ { intent₁ = true }) ·
  while intent₂ (
    if ⌊ turn ≟ 0 ⌋ ⟶ skip
    :: ⌊ turn ≟ 1 ⌋ ⟶ (
      update (λ σ → record σ { intent₁ = false }) ·
      await (⌊ (turn ≟ 0) ⌋) ·
      update (λ σ → record σ { intent₁ = true }))
    :: []
    fi) ·
  CS₁ ·
  update (λ σ → record σ { turn = 1 }) ·
  update (λ σ → record σ { intent₁ = false})
```

```
dekkers₂ : GCL
dekkers₂ =
  update (λ σ → record σ { intent₂ = true }) ·
  while (intent₁) (
    if ⌊ turn ≟ 1 ⌋ ⟶ skip
    :: ⌊ turn ≟ 0 ⌋ ⟶ (
      update (λ σ → record σ { intent₂ = false }) ·
      await ⌊ turn ≟ 1 ⌋ ·
      update (λ σ → record σ { intent₂ = true }))
    :: []
    fi) ·
  CS₂ ·
  update (λ σ → record σ { turn = 0 }) ·
  update (λ σ → record σ { intent₂ = false})
```

Figure 6. Dekker's algorithm in our embedded GCL

(Whiteside et al. 2012) includes several computation-focused tactics for proof-search, and was used in the proof of the four colour theorem (Gonthier 2008). All of these developments, however, rely on metaprogramming using an external tactic or proof language. Our approach embeds the language used to describe proof-search inside the object language, and therefore does not require any special prover support.

Property based testing has been used in theorem provers before, such as in Isabelle (Bulwahn 2012) and even in Agda's ancestor Alfa (Dybjer et al. 2003). Dybjer et al. (2004) even went on to integrate a model checker with Alfa. In each instance, however, these tools are external plugins to the proof assistant, and are used merely to assist manual proving by finding counterexamples to goals, not as a mechanism for proof-search. More recently, Paraskevopoulou et al. (2015) implemented and verified a property based testing library in Coq based on pseudorandom candidate generation. Unlike our approach, they do not enrich the types of properties or checkers with their semantic information, but instead verify each combinator in a more traditional, post-hoc fashion. Other work focuses on using theorem provers to generate test data suitable for particular properties (Brucker and Wolff 2013; Carlier et al. 2012). It would be interesting to integrate these approaches with our test framework to generate data more intelligently.

Much of the proof automation work in Agda thus far has been focused on implementing traditional tactic methods, such as ring solvers (Jedynak 2015), presburger arithmetic solvers (Allais 2015), or first order proof search (Kokke and Swierstra 2014). These methods typically operate over a *deep embedding*, where the proof problem is reified into a concrete data type, either manually or by Agda's reflection mechanism (Van Der Walt and Swierstra 2013), and the proof search manipulates these concrete terms. We have described a more general framework for proof-search based instead on *shallow embeddings*, where the proposition under test is directly represented in the framework.

9. Future Work

Dependent types are the key for *certifying* proof-search. Now, instead of a test framework simply reporting a counterexample, a test framework can report a counterexample *along with the reason it doesn't work*, with that evidence verified by the type system. Instead of a model checker spitting out a message barely better than "`I think I'm done!`" and terminating, it can emit a proof of the properties it was supposed to verify. These proofs can then be combined with manual reasoning — we can use a model checker as a tactic!

There are a number of potential research directions that can be taken in this vein:

- Exploring the world of model checking, and integrating more sophisticated techniques, such as LTL model checking or BDD-based model checking, into Agda using our framework. This would allow algorithms such as Dekker's algorithm to be verified.
- Making use of Agda's reflection mechanism to automate proof work. For our development, it may be possible to generate the HDec corresponding to a goal automatically using this feature, by defining a deeply-embedded representation of propositions under test.
- Improving Agda's type-checker evaluation performance, to make larger-scale developments using computation in the type checker possible. Currently, every example in this paper checks in less than a minute, with the exception of petersons-correct, which takes three minutes on a 2013 MacBook Pro, but getting performance to this point required significant engineering effort.

It is our hope that a wide variety of proof automation tools can be integrated into Agda in this way. We strongly believe that this kind of proof automation should be further investigated.

All of the code developed in this paper, and more besides, is available as an open-source Agda library on GitHub:

`https://github.com/liamoc/me-em`

Acknowledgments

I would like to thank Edward Lee for his mathematical intuition; Ulf Norell for excellent performance suggestions; Ben Lippmeier, Gabi Keller, and various anonymous reviewers for their useful feedback.

References

G. Allais. Presburger arithmetic solver for agda. `https://github.com/gallais/agda-presburger`, 2015.

K. Appel, W. Haken, et al. Every planar map is four colorable. part i: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013. ISSN 1433-299X. doi: 10.1007/s00165-012-0222-y. URL `http://dx.doi.org/10.1007/s00165-012-0222-y`.

L. Bulwahn. The new quickcheck for isabelle. In *Certified Programs and Proofs*, pages 92–108. Springer, 2012.

M. Carlier, C. Dubois, and A. Gotlieb. A first step in the design of a formally verified constraint-based testing tool: Focaltest. *Tests and Proofs: 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 – June 1, 2012. Proceedings*, pages 35–50, 2012. doi: 10.1007/978-3-642-30473-6_5.

K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montréal, Canada, September 18-21, 2000*, number 9, page 268. Pearson Education, 2000.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986. ISSN 0164-0925. doi: 10.1145/5397.5399. URL `http://doi.acm.org/10.1145/5397.5399`.

D. Devriese and F. Piessens. On the bright side of type classes: Instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 143–155, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034796. URL `http://doi.acm.org/10.1145/2034773.2034796`.

E. W. Dijkstra. Report on a conference sponsored by the NATO Science Committee, Rome. In *Software Engineering Techniques*, Apr. 1970.

E. W. Dijkstra. Cooperating sequential processes. In P. Hansen, editor, *The Origin of Concurrent Programming*, pages 65–138. Springer New York, 2002. ISBN 978-1-4419-2986-0. doi: 10.1007/978-1-4757-3472-0_2.

P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, TPHOLS '03, pages 188–203, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45130-3. doi: 10.1007/10930755_12. URL `http://dx.doi.org/10.1007/10930755_12`.

P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology*, 46(15):1011–1025, 2004. doi: 10.1016/j.infsof.2004.07.002. URL `http://dx.doi.org/10.1016/j.infsof.2004.07.002`.

W. B. e. Ewald. From immanuel kant to david hilbert: A source book in the foundations of mathematics. Oxford University Press, 1996.

R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

G. Gonthier. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics*, pages 333–333. Springer, 2008.

T. C. Hales. A proof of the kepler conjecture. *Annals of mathematics*, pages 1065–1185, 2005.

T. C. Hales. Introduction to the flyspeck project. In T. Coquand, H. Lombardi, and M.-F. Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2006/432`.

G. J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

W. Jedynak. Ring solver for agda. `https://github.com/wjzz/Agda-reflection-for-semiring-solver`, 2015.

P. Kokke and W. Swierstra. Auto in Agda: programming proof search. Submitted to ICFP 2014, `http://www.staff.science.uu.nl/~swier004/Publications/AutoInAgda.pdf`, 2014.

S. A. Kripke. Semantical considerations on modal logic. In *Acta Philosophica Fennica*, volume 16, pages 83–94, 1963.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of functional programming*, 18(01):1–13, 2008.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.

Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational Property-Based Testing. In *ITP 2015 - 6th conference on Interactive Theorem Proving*, volume 9236 of *Lecture Notes in Computer Science*, Nanjing, China, Aug. 2015. Springer. doi: 10.1007/978-3-319-22102-1_22. URL `https://hal.inria.fr/hal-01162898`.

G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 37–48, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: 10.1145/1411286.1411292.

K. Slind and M. Norrish. A brief overview of hol4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 28–32, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-71065-3. doi: 10.1007/978-3-540-71067-7_6. URL `http://dx.doi.org/10.1007/978-3-540-71067-7_6`.

W. Thomas. Handbook of theoretical computer science (vol. b). chapter Automata on Infinite Objects, pages 133–191. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-444-88074-7.

P. Van Der Walt and W. Swierstra. Engineering proof by reflection in agda. In *Implementation and Application of Functional Languages*, pages 157–173. Springer, 2013.

I. Whiteside, D. Aspinall, and G. Grov. An essence of ssreflect. In *Proceedings of the 11th International Conference on Intelligent Computer Mathematics*, CICM'12, pages 186–201, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31373-8. doi: 10.1007/978-3-642-31374-5_13. URL `http://dx.doi.org/10.1007/978-3-642-31374-5_13`.

B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 87–100, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500579. URL `http://doi.acm.org/10.1145/2500365.2500579`.