Type-level Property Based Testing

Thomas Ekström Hansen
Edwin Brady
teh6@st-andrews.ac.uk
ecb10@st-andrews.ac.uk
School of Computer Science, University of St Andrews
St Andrews, Fife, United Kingdom

Abstract

We present an automated framework for solidifying the cohesion between software specifications, their dependently typed models, and implementation at compile time. Model Checking and type checking are currently separate techniques for automatically verifying the correctness of programs. Using Property Based Testing (PBT), Indexed State Monads (ISMs), and dependent types, we are able to model several interesting systems and network protocols, have the type checker verify that our implementation behaves as specified, and test that our model matches the specification's semantics; a step towards combining model and type checking.

1 Introduction/Motivation

Stateful computer systems are ubiquitous. Embedded devices, computer networks and banking systems all involve states and transitions between different states. We can formally verify programs using tools like Spin [30] or Uppaal [5], however these tools rarely scale to real-world code bases due to the State Explosion Problem (although progress is steadily being made) [22–24, 26, 39]. There are also several testing tools and methodologies one can employ, such as Test-Driven Development — à la JUnit — and Fuzz-testing [29, 40], which typically target imperative programming languages.

For functional languages, Property Based Testing (PBT) using QuickCheck [19] has proven widely successful [2, 18, 20, 31, 32] and been ported to other languages, both functional and imperative [6, 27, 35]. Test-driven methods are well understood, but the tests are only as good as the cases which the programmer can think of. On the flip side, model-checking tools require a secondary implementation of the program in the form of a formal model, which opens the verification process up to errors in translation leading to a semantic mismatch between the running code and the verified model [2]).

Dependently Typed languages like Agda [8] and Idris [9, 12] can be used for "correct by construction" programming, where typically an Embedded Domain Specific Language (eDSL) is constructed to ensure the program is valid by definition [7, 13, 14, 16]. However, well-typed programs can go wrong. Occasionally, this is due to bugs in the type checker [17] or due to problems with how programmers use the language-provided escape hatches [36], but there is also a third, arguably more likely, case: what if the types themselves are subtly incorrect? One could imagine a program requiring that a certain number remain positive but either of familiarity or by accident, the programmer gives the type `Int` instead of `Nat`. This is unlikely to be caught by the type checker, tests, or implementations, as the programmers are likely to carry this assumption in their minds, thereby avoiding including it in both tests and implementation error-checks. Nevertheless, the code modelling the specification has now introduced subtly different permitted states. How can we be sure that the DSL does not accidentally permit an incorrect state or transition?

1.1 Contributions

We make the following contributions:

- An implementation of QuickCheck for use with dependent types at compile time.
- A framework for simultaneously specifying, implementing, and testing a stateful model of an Automated Teller Machine (ATM), using QuickCheck to increase confidence in the correctness of all 3 parts.
- We demonstrate the power of the framework by generalising it to stateful programs, both finite and infinite, and evaluate it by implementing an example of a network protocol.

In doing so, we aim to increase confidence in the correctness of the types we use to model stateful systems, using type level testing to help us understand the behaviour of state machines.

2 QuickCheck in Idris2

QuickCheck is a Property Based Testing tool introduced by Claessen and Hughes in 2000 [19]. Although initially written for Haskell, it has been successfully ported to many programming languages including Isabelle [6], Erlang [20], Coq [27], and Java [35].

Since types in Idris2 are first-class, we can use our QuickCheck implementation at the type level. Thus the test suite can be run at compile-time, acting on the implementation itself — rather than a test environment — and then being erased for the compiled program.

2.1 Regular QuickCheck in Idris2

Most of QuickCheck ports directly to Idris2 from the original Haskell implementation, with some minor modifications.

```
data Gen : Type -> Type where
  MkGen :  (Int -> PRNGState -> a) -> Gen a
```

Idris2 has no `newtype` so we have to wrap it in a regular datatype; the implementation is otherwise identical. The `PRNGState`-type represents the state type of a Pseudorandom Number Generator (PRNG), which is essential for allowing QuickCheck to generate example instances. As in Haskell, `Gen` is an instance of the `Monad` interface. It requires the passing two *independent* PRNGs [19], which is easily achieved if the provided PRNG is *splittable* — that two seemingly independent PRNGs may be derived from an initial one [15].

```
(>>=) (MkGen g1) c = MkGen (\n, r0 =>
  let (r1, r2) = split r0
      (MkGen g2) = c (g1 n r1)
  in g2 n r2)
```

Statistically sound splits challenging to achieve and have been the source of bugs in the Haskell implementation [21] Schaatun [37] concluded that amongst many existing allegedly splittable PRNGs, only the one presented by Claessen and Pałka [21] was sound. As such, we do not make any cryptographic promises about the PRNG used in our implementation: we use a Linear Congruence Generator for its ease of implementation, using multipliers from [38]. This is sufficient for demonstrating our approach.

Another difference between Idris2 and Haskell is that `(->)` is not a type constructor in Idris2, but rather a binder. Therefore, we manually wrap functions in a data type alongside an eliminator:

```
data Fn : Type -> Type -> Type where
  MkFn : (f : a -> b) -> Fn a b

apply : Fn a b -> a -> b
apply (MkFn f) = f
```

We can now implement *promoting* functions of type `a -> Gen b` to a generator of functions from `a` to `b`, as well as implement the `Arbitrary` interface for functions, given we know how to (1) modify a generator given some specific instance of the domain's type, and (2) generate arbitrary instances of the codomain's type:

```
promote : (a -> Gen b) -> Gen (Fn a b)
  promote f = MkGen (\n, r => MkFn
    (\x => let (MkGen gb) = f x in gb n r))

Arbitrary a => Arbitrary b =>
Arbitrary (Fn a b) where
  arbitrary = promote (`coarbitrary` arbitrary)

  coarbitrary fn gen = arbitrary >>=
    ((`coarbitrary` gen) . (apply fn))
```

2.2 QuickCheck with Dependent Types

Using QuickCheck with dependent types presents new challenges. Consider, for example, the `Vect` datatype: dependently typed lists with length. QuickCheck's bread and butter is generators and the `Arbitrary` interface. Provided we know how to generate the type of the elements, we should be able to generate a vector of them. However, the following definition fails:

```
Arbitrary t => Arbitrary (Vect n t) where
  arbitrary = do
    arbN <- arbitrary
    v <- genN arbN arbitrary
    pure v
    where
      genN : (n : Nat) -> Gen t -> Gen (Vect n t)
      genN Z _ = []
      genN (S k) g = do
        x  <- g
        xs <- genN k g
        pure (x :: xs)
```

The type checker has no way of unifying `n` with `arbN`; it needs to know that the two are the same number, and it does not know that there is a link between the two. Indeed, it *cannot* know because it is unknown at type checking time, what length of vector will be generated! One solution is to write `Arbitrary` instances only for the sizes of vectors we are interested in:

```
Arbitrary t => Arbitrary (Vect 3 t) where
  arbitrary = do
    v0 <- arbitrary
    v1 <- arbitrary
    v2 <- arbitrary
    pure [v0, v1, v2]
```

However, this is not very useful: we need an implementation for every length, and should we want to generate an arbitrary length we would have to reduce our arbitrary length to only the lengths which we have defined generators for, defeating the point of `Arbitrary` interface. Instead, we can use dependent pairs, where the second element's type depends on the value of the first, written using `(**)`. This allows us to indicate to the type checker that although we may not know the exact length of the vector at type checking time, *when* we know the length, we also know it is related to a concrete instance of `Vect`:

```
someVect : (n : Nat ** Vect n Nat)
someVect = (3 ** [1, 2, 3])
```

Idris2 has `failing` blocks which compile if and only if they contain a term raising an error. Passing in a string requires it to be part of the error message, with the compiler rejecting the block if it fails with a different message. We can thus confirm it is an error to provide a mismatching length and `Vect`:

```
failing "Mismatch between: 1 and 0."
  sizeMismatch : (n : Nat ** Vect n Nat)
  sizeMismatch = (0 ** [3])
```

And we can ask Idris2 to infer the first element's value:

```
inferLength : (n : Nat ** Vect n Nat)
inferLength = (_ ** [1, 2, 3])
```

Dependent pairs allow us to define a general `Arbitrary` instance for `Vect`:

```
Arbitrary t => Arbitrary (n : Nat ** Vect n t) where
  arbitrary = do
    nElems <- arbitrary
    vect   <- genN nElems
    pure (_ ** vect)
    where
      genN : Arbitrary a => (m : Nat) -> Gen (Vect m a)
      genN Z     = pure []
      genN (S k) = do
        x  <- arbitrary
        xs <- genN k
        pure (x :: xs)
```

3 Example: ATM

We now consider an example Finite State Machine (FSM) modelling the behaviour of an Automated Teller Machine (ATM). This example is used as a showcase of how dependent types can help with correct stateful programming [11]. However, as we shall shortly see, while dependent types go a long way towards helping us be confident our program is correct, they are not enough on their own.

3.1 The ATM state machine

The ATM state machine consists of three states:

- **Ready** — The starting state of the ATM, representing the machine being ready for operation.
- **CardInserted** — When a card is present in the ATM, pending authorisation.
- **Session** — An authorised session whereby the user can dispense an amount of money.

Figure 1 illustrates the following transitions:

- *Insert* — Inserting a bank card. This action is only valid when the ATM is in the **Ready** state and results in the machine changing to the **CardInserted** state.
- *Dispense* — Dispensing a given amount of money. This is only valid when the card has been authenticated, i.e. the machine is in a **Session**. Since a user may want to dispense multiple amounts of money, *Dispense* keeps the machine in its **Session** state.
- *CheckPIN* — Verifying that the given PIN authenticates the card. This is only valid when the ATM has a card in it. This transition is unique in that it leads to different states depending on the result of checking the PIN: *Incorrect* causes the machine to stay in the **CardInserted** state, whereas *Correct* moves the machine to the **Sesssion** state.
- *Eject* — At any point, the user may choose to eject their card. This takes the machine back to the **Ready** state.

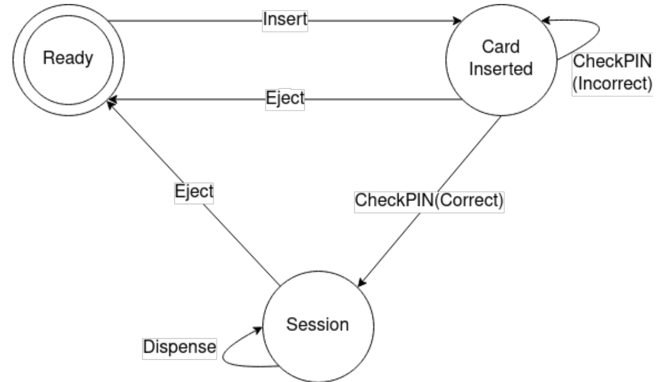

Figure 1. Diagram of the ATM state machine

3.2 Modelling the ATM in Idris2

We can model this state machine in Idris2 by declaring a new data type for the *CheckPIN* results, with constructors for each option, as well as a data type for the states, with a constructor per state:

```
data ATMState = Ready | CardInserted | Session

data PINok = Correct | Incorrect
```

Next, we model the function describing the dependent state transition for *CheckPIN*. This is a function from the result type, `PINok`, to the type of the states, `ATMState`:

```
ChkPINfn : PINok -> ATMState
ChkPINfn Correct = Session
ChkPINfn Incorrect = CardInserted
```

With the states, PIN results, and dependent transition modelled, we can now model the transitions themselves. As described in Brady's Book [11], this is where dependent types really get a chance to shine for modelling and programming these stateful systems: we index our operations by their result type, their starting state, and their state transition functions. This allows us to use the type declaration to state what the result type of our program should be, its starting state, and its end state, and having the type checker verify that we keep our promise and reach the end state via lawful transitions. Furthermore, we supply a bind operator for do-notation. The transition function, for most states, is a `const` function, as they only move from one state to the next. However, for *CheckPIN*, the state function is more interesting. We refer to the complete model of operations as an Indexed State Monad (ISM):

```
data ATM :  (t : Type) -> ATMState
         -> (t -> ATMState) -> Type where
```

```
  Insert : ATM () Ready (const CardInserted)
  CheckPIN : (pin : Int) -> ATM PINok CardInserted
                                ChkPINfn
  Dispense : (amt : Nat) -> ATM () Session
                                (const Session)
  Eject : ATM () st (const Ready)
  Pure : (x : t) -> ATM t (stFn x) stFn
  (>>=) : ATM a s1 s2f -> ((x : a) -> ATM b (s2f x)
          s3f) -> ATM b s1 s3f
```

We can now use the model to write stateful programs which are guaranteed to conform to the model:

```
testProg : ATM () Ready (const Ready)
testProg = do
  Insert
  Correct <- CheckPIN 1234
    | Incorrect => ?handle_incorrect
  Dispense 42
  Eject
```

And programs which attempt to misbehave are rejected:

```
failing "Mismatch between: Session and CardInserted."
  badProg : ATM () Ready (const Ready)
  badProg = do Insert ; Dispense 42
```

This setup looks to be correct: we have our dependently typed model which describes the desired semantics; we can use it to program with, expressing state invariants which must be obeyed and which are automatically verified; and whilst writing the program, the type checker keeps track of the state for us. This is a very strong position compared to languages without such typed modelling capabilities. However, while it may seem correct, there is a mistake in this model of the ATM: the amount of PIN retries is unlimited. The `ChkPINfn` only takes a `PINok` result, neither it nor the **CardInserted**-state keeps track of how many times the user has tried to enter a PIN. The following program, while not terminating, is completely valid as far as the type checker knows:

```
covering
loopProg : ATM () Ready (const Ready)
loopProg = do
    Insert
    let pin = 4321
    loop pin
  where
    loop : Int -> ATM () CardInserted (const Ready)
    loop p = do
      Incorrect <- CheckPIN p
        | Correct => ?omitted
      loop p
```

Unfortunately, *this error is not caught by the type checker.* Even totality checking does not help: it is a terminating computation, for example, to iterate over all 10,000 PINs and withdrawing all the money on finding the correct one. This illustrates a tricky situation: as type-driven programmers, we are inclined to believe that expressive types mean the type checker will catch our mistakes. Nevertheless, subtle errors may occur in our modelling, and there is no way to automatically catch these unless the programmer tries to write non-obvious programs. Who type checks the types?

3.3 A framework for ATM simulation

To gain confidence in our specification, we could try modelling it in a formal verification tool or model checker, but this does not solve the root of the problem: our models themselves can be wrong, and so translating them into different tools gives more places for introducing errors, or worse, different errors in each model. We would instead like to generate example instances of each part of our model, pass these through our state transitions, and specify properties which, provided well-typed inputs, the model obeys. Ideally, this should be done in the same development environment as the model and implementation, thus eliminating the risk of translation mishaps. With some work, we can achieve this with QuickCheck.

In section 2.2 we saw how dependent pairs allow us to generate arbitrary dependent types. This means we could, hypothetically, declare the following `Arbitrary` instance:

```
Arbitrary (resT : Type ** nsFn : resT -> ATMState
           ** ATM resT st stFn)
```

If we know the result type, state function's type, and some starting state, we have all the necessary information to construct a concrete instance of the `ATM` type. However, such an instance has a couple of problems:

1. It would only generate a single operation at a time, with no obvious way to trace which operations were taken when. Related to this is the issue of how to generate instances of the binding and sequencing operators, which each require a specific pair of operations to work correctly.
2. To advance to the next state, we need an instance of the result type `resT`. However, we only know the *type* of results the operation returns, we do not know *which* instance of that type it returned. We could make up a value, but then we would fix a parameter that we want to test.

3.3.1 Separating operations from programming logic.

To address the first problem, we split the operations and the sequencing into separate types:

```
  data ATMOp :  (t : Type) -> ATMState
             -> (t -> ATMState) -> Type where
    Insert : ATMOp () Ready (const CardInserted)
    CheckPIN : (pin : Int) -> ATMOp PINok CardInserted
                                ChkPINfn
    Dispense : (amt : Nat) -> ATMOp () Session
                                (const Session)
    Eject : ATMOp () st (const Ready)

  data ATM :  (t : Type) -> ATMState
           -> (t -> ATMState) -> Type where
```

```
Op : ATMOp t st nsFn -> ATM t st nsFn
Pure : (x : t) -> ATM t (nsFn x) nsFn
(>>=) : ATM a s1 s2f -> ((x : a) -> ATM b (s2f x)
          s3f) -> ATM b s1 s3f
```

This separation allows us to access the next-state function directly from the ATMOp type. We already specify it as part of the type, so given a concrete ATMOp we should have access to its next-state function. The only caveat is that we need to be operating at the type level. Idris2 erases runtime-irrelevant proofs and types by default, meaning we are only allowed to access them in parts of the program that are themselves erased, that is, those defined at quantity 0 in QTT [3, 12, 34].

```
0 nextState :  (st : ATMState) -> ATMOp t st nsFn
            -> (res : t) -> ATMState
nextState st _ res = nsFn res
```

On its own, this function is no more useful than the Arbitrary instance from earlier, but as we shall demonstrate, extracting the state transition function allows us to establish clear links between the model, tests, and implementation.

3.3.2 Tracing operations and programs.

We now consider the second issue: needing to keep track of the result type *along with* a concrete instance of the result type, in a form we can control and test. To do this, we store an operation along with its result in a record type, OpRes:

```
record OpRes (resT : Type) (currSt : ATMState)
             (nsFn : resT -> ATMState) where
  constructor MkOpRes
  op : ATMOp resT currSt nsFn
  res : resT
  {auto rShow : Show resT}
```

The auto-implicit rShow is so that QuickCheck can print counterexamples if necessary. Given OpRes, all that remains is to trace how a state and operations are chained. We first consider each individual step, where an OpRes and its resulting ATMState are stored in the same record, and then create a type which, for a given bound, stores the trace:

```
record TraceStep where
  constructor MkTS
  opRes : OpRes rT aSt aStFn
  resSt : ATMState

data ATMTrace : ATMState -> Nat -> Type where
  MkATMTrace : (initSt : ATMState) -> {bound : Nat}
             -> (trace : Vect bound TraceStep)
             -> ATMTrace initSt bound
```

With this framework in place, we are finally ready to write meaningful Arbitrary instances for testing our ATM model.

3.4 Arbitrary OpRes

In order to generate arbitrary traces, we first need to be able to generate arbitrary operation-result pairs. Generating an OpRes requires knowing what the current state is, as we need it to determine the set of valid operations from it. As with generating Vect instances, we use dependent pairs to capture the chain of things we need to know about the type of the operation. Once we know the concrete resT type, we know the type of our nsFn, which means we know the type of our OpRes:

```
(resT : Type ** nsFn : Type -> ATMState
  ** OpRes resT st nsFn)
```

For the concrete implementation, we can now pattern-match on the implicitly given st. This restricts which operations are available, as only some are valid in a given state, allowing us to return an operation chosen randomly from the set of compatible operations. We can also assign weights to the individual elements via QuickCheck's frequency function, thereby controlling how often certain operations are picked compared to others. Our Arbitrary instance for OpRes thus becomes:

```
{currSt : ATMState} ->
Arbitrary (resT : _ ** nsFn : resT -> ATMState
            ** OpRes resT currSt nsFn) where
  arbitrary {currSt=Ready} =
    pure (_ ** _ ** MkOpRes Insert ())

  arbitrary {currSt=CardInserted} = do
    let arbPIN = 0
    let correct = (_ ** _ ** MkOpRes
                    (CheckPIN arbPIN)
                    Correct)
    let incorrect = (_ ** _ ** MkOpRes
                      (CheckPIN arbPIN)
                      Incorrect)
    let eject = (_ ** _ ** MkOpRes Eject ())
    frequency $ [ (1, pure correct)
                , (4, pure incorrect)
                , (1, pure eject) ]

  arbitrary {currSt=Session} = do
    arbAmount <- arbitrary
    let dispense = (_ ** _ ** MkOpRes
                    (Dispense arbAmount)
                    ())
    let eject = (_ ** _ ** MkOpRes Eject ())
    oneof $ map pure [dispense, eject]
```

3.5 Arbitrary ATMTrace

To chain steps together, we make an instance of Arbitrary ATMTrace. Recall that traces are bounded by their depth, therefore we pattern-matching on the depth.

3.5.1 If $depth = 0$:

the remaining trace must be empty as the depth bound has been reached.

```
arbitrary {depth=0} = pure $ MkATMTrace iSt []
```

3.5.2 If $depth = (S\ b)$:

the trace depth has not been reached and we need to generate at least one more trace step.

```
arbitrary {depth=(S b)} = do
  ?arbitrary_trace_rhs
```

The first step is to generate an arbitrary OpRes, which we can now do thanks to the implementation from section 3.4. We may not know the OpRes's result type or state function but we can capture this in type variables:

```
arbitrary {depth=(S b)} = do
  opRes <- the (Gen (resT ** fn **
                      OpRes resT iSt fn)) arbitrary
  ?arbitrary_trace_rhs
```

Asking the compiler for the type of the hole gives us:

```
  iSt : ATMState
  b : Nat
  opRes : (resT : Type ** (fn : resT -> ATMState **
            OpRes resT iSt fn))
------------------------------
arbitrary_trace_rhs : Gen (ATMTrace iSt (S b))
```

The generated dependent pair containing our new OpRes is not too useful as a single variable. However, we can split out its components via pattern-matching.

```
arbitrary {depth=(S b)} = do
  (resT ** nsFn ** (MkOpRes op res)) <- the
                                  (Gen {-...-})
  ?arbitrary_trace_rhs
```

This gives us access to several critical pieces of data:

- op — The operation itself, so we can log what operation led where.
- res — The result of the operation. In order to continue constructing our trace, we need to know what state we moved to, which requires applying the next-state function to a concrete result; this is exactly what is given here.
- nsFn — The next-state function *as written directly in the type*. It is worth re-emphasising this: we are guaranteed to use the same transition function as our model/specification, because we are extracting it from the type which uses it! We can access this because the entire process is happening at type checking time, so we can use elements which will be erased at run time.

And we can confirm this by taking a look at the updated information for our hole:

```
  iSt : ATMState
  b : Nat
  resT : Type
  nsFn : resT -> ATMState
  res : resT
  op : ATMOp resT iSt nsFn
------------------------------
arbitrary_trace_rhs : Gen (ATMTrace iSt (S b))
```

Applying nsFn to the generated res gives us the first state of our trace:

```
arbitrary {depth=(S b)} = do
  (resT ** nsFn ** (MkOpRes op res)) <- the
                                  (Gen {-...-})
  let fstTraceSt = nsFn res
  ?arbitrary_trace_rhs
```

Which we again confirm by looking at our new supporting information:

```
  iSt : ATMState
  b : Nat
  resT : Type
  nsFn : resT -> ATMState
  res : resT
  op : ATMOp resT iSt nsFn
  fstTraceSt : ATMState
------------------------------
arbitrary_trace_rhs : Gen (ATMTrace iSt (S b))
```

The first trace state can only have been obtained by following the specification's semantics because the function we applied to generate it was the exact function specified in the original type! We now construct the trace by storing the operation and its resulting state and recursively generating the rest of the trace:

```
arbitrary {depth=(S b)} = do
  (resT ** nsFn ** (MkOpRes op res)) <- the
                                  (Gen {-...-})
  let fstTraceSt = nsFn res
  let atmTrace =
    (MkTS (MkOpRes op res) fstTraceSt) ::
      !(trace b fstTraceSt)
  pure $ MkATMTrace iSt atmTrace
```

Above, we use a helper function, trace, because MkATMTrace expects a Vect of exactly bound elements. We could extract this from a recursive call to arbitrary, generating a new ATMTrace and then pattern-matching on its constructor to extract the Vect, however we prefer this solution of generating the Vect in-place using a helper function. Its definition is almost verbatim that of the abitrary instance:

```
trace :  (steps : Nat)
      -> (st : ATMState)
      -> Gen (Vect steps TraceStep)
trace 0 _ = pure []
trace (S k) st = do
  (_ ** nsFn ** opR@(MkOpRes _ res)) <- the
                                  (Gen {-...-})
  let nextState = nsFn res
  pure $ (MkTS opR nextState) ::
         !(trace k nextState)
```

The as-pattern captures the entire OpRes, meaning we can omit the operation being bound to a variable as we never use it in the body of the function; and the bang-notation is shorthand for extracting the result of a monadic computation [10].

3.6 QuickCheck-ing the type-level ATM

We have our specification, modelled as an ISM, with datatypes for generating sample execution traces, so we are now in a position to specify properties for QuickCheck to verify. To start, we check that when we are in the **Ready** state, we always end up in **CardInserted** after a single operation.

```
0 PROP_readyInsert : Fn (ATMTrace Ready 1) Bool
PROP_readyInsert = MkFn
  (\case (MkATMTrace _ (
           (MkTS _ CardInserted) :: []))
             => True
         (MkATMTrace _ _) => False)
```

Notice that our property is given at quantity 0, so that it is compile time only. To QuickCheck this, we wrap the default quickCheck function in a type-level one, which tests the given property, specifying whether the test should be considered passed if QuickCheck exhausted the arguments.

```
0 QuickCheck :  Testable t
             => (allowExhaust : Bool)
             -> (prop : t)
             -> Bool
QuickCheck allowExhaust prop =
  Maybe.fromMaybe allowExhaust $
    (quickCheck prop).pass
```

Using the Idris2 built-in data type `Data.So`, which is inhabited if and only if its argument evaluates to `True`, we can now ask the compiler to ensure the property holds:

```
0 RI_OK : So (QuickCheck False PROP_readyInsert)
RI_OK = Oh
```

As the file loads successfully, we can be confident that our model is sound with respect to the specified property. Next, we specify a property which we hope QuickCheck will find does not hold: that the ATM eventually gets back to an available state within reasonable time.

```
0 PROP_eventuallyReady : Fn (ATMTrace Ready 10) Bool
PROP_eventuallyReady = MkFn
  (\case (MkATMTrace _ trace)
    => elem Ready (map (.resSt) trace))

0 ER_OK : So (QuickCheck False PROP_eventuallyReady)
ER_OK = Oh
```

Trying to load the file with this property gives:

```
-- Error: While processing right hand side of
--          EventuallyReady_OK. When unifying:
    So True
-- and:
    So (QuickCheck False PROP_eventuallyReady)
-- Mismatch between: True and False
```

QuickCheck returns `False`, indicating that our property is failing. Inspecting the reason by running QuickCheck on the property at the Idris2 REPL reveals the cause of the issue:

```
MkQCRes (Just False) <log> """
Falsifiable, after 4 tests:
Starting @ Ready:
[ (<ATMOp 'Insert ~ ()'>, CardInserted)
, (<ATMOp 'CheckPIN 0 ~ Incorrect'>, CardInserted)
, (<ATMOp 'CheckPIN 0 ~ Incorrect'>, CardInserted)
, (<ATMOp 'CheckPIN 0 ~ Incorrect'>, CardInserted)
, (<ATMOp 'CheckPIN 0 ~ Incorrect'>, CardInserted)
-- <etc>
]"""
```

This is the loop that was indeed wrong in the initial specification. Our setup has constructed sample programs which use the same semantics as our model and implementation, and discovered unintended behaviour in the model itself.

Remark: The first test is technically incorrect: the ISM, as it is specified, allows for the user to attempt to *Eject* the card in the **Ready** state, a no-op. However, our generator for `OpRes` never includes this option. This is an inherent shortcoming with QuickCheck— it is no silver bullet to incomplete data generators.

3.7 Fixing the ATM

To fix the model, we index the **CardInserted**-state by the number of retries available, and update the next-state function to take this number into account. This limits the number of permitted PIN attempts.

```
data ATMState = Ready | CardInserted Nat | Session
```

There is a minor technical issue to solve along the way: since the number of retries is stored something of type `ATMState`, this means our updated next-state function takes a generic `ATMState` despite us only being interested in the **CardInserted** case. To resolve this, we define a predicate which requires the state to be **CardInserted** and use it in our updated `ChkPINfn`:

```
data IsCI : ATMState -> Type where
  ItIsCI : IsCI (CardInserted (S k))

ChkPINfn :  (ciSt : ATMState)
         -> {auto ford : IsCI ciSt}
         -> PINok -> ATMState
ChkPINfn (CardInserted (S k)) {ford=ItIsCI}
          Correct = Session
ChkPINfn (CardInserted (S 0)) {ford=ItIsCI}
          Incorrect = Ready
ChkPINfn (CardInserted (S (S k))) {ford=ItIsCI}
          Incorrect = CardInserted (S k)
```

As can be seen above, by pattern-matching on the predicate, Idris2 can infer that the seemingly generic `ciSt` argument is actually restricted to one specific instance of the type. This means our updated `ChkPINfn` now neatly slots in to the fixed `ATMOp` structure:

```
data ATMOp :  (t : Type) -> ATMState
           -> (t -> ATMState) -> Type where
  Insert : ATMOp () Ready (const (CardInserted 3))
  CheckPIN :  (pin : Int)
           -> ATMOp PINok
                    (CardInserted (S tries))
                    (ChkPINfn
                      (CardInserted (S tries)))
  Dispense :  (amt : Nat)
           -> ATMOp () Session (const Session)
  Eject : ATMOp () st (const Ready)
```

The file reloads successfully, meaning the type-level property test ER_OK passed. And if we retest the property at the Idris2 REPL, we get:

```
MkQCRes (Just True) <log> "OK, passed 100 tests"
```

We are now no longer able to introduce a loop in our implementation, as the fourth attempt involves 0 remaining retries, which forces us back into **Ready** thanks to the updated ChkPINfn.

```
failing "Mismatch between: CardInserted (S ?tries)
         and Ready."
  noLoop : ATM () Ready (const Ready)
  noLoop = do
    Op Insert
    Incorrect <- Op $ CheckPIN 1234
      | Correct => ?noLoop_rhs_1
    Incorrect <- Op $ CheckPIN 1243
      | Correct => ?noLoop_rhs_2
    Incorrect <- Op $ CheckPIN 1432
      | Correct => ?noLoop_rhs_3
    Incorrect <- Op $ CheckPIN 4231
      | Correct => ?noLoop_rhs_4
    ?noLoop_rhs
```

This highlights the power of our new approach: an error in the specification can be automatically found and, once fixed, the new model is automatically threaded through to both the type checker — verifying all implementations — and the sample program generation. This greatly increases our confidence that the model is well-behaved, meaningfully tested, and correctly implemented.

4 Generalising

The ATM example required significant effort to to type-check, program, and property test. If this were required for every state model, the approach would be tedious to adopt. Instead, it would be convenient to only have to specify the model and its transitions, and get the rest "for free".

4.1 Generic operations and programs

To generalise the data types from the previous section, we need to extract the common factor and index over it. As briefly discussed in section 3.3.1, the programming part of our approach is largely already generalised. To reuse the code with a different system, we need to define the new states and transitions (or operations): This gives us our indices: the state – st : Type – and the type of the operations – op:

```
op : forall st . (t' : Type) -> st -> (t' -> st) -> Type
```

In order to make the Prog type generic, we index it over the type of valid operations for the states:

```
data Prog :  {0 stT : _}
          -> (opT :  (t' : _) -> stT -> (t' -> stT)
                  -> Type)
          -> (t : Type) -> (from : stT)
          -> (to : t -> stT) -> Type where
    Pure : (x : t) -> Prog opT t (stFn x) stFn
    Op :  {0 opT :  (t' : _) -> stT -> (t' -> stT)
                 -> Type}
       -> opT t st stFn -> Prog opT t st stFn
    (>>=) : Prog opT resT1 st1 stFn1
         -> ((x : resT1)
              -> Prog opT resT2 (stFn1 x) stFn2)
         -> Prog opT resT2 st1 stFn2
```

This gives us a generic way to describe a program producing a result of some type, starting in a given state, and ending in a state depending on the result. Note that the program's return type and the operations' return types may differ; each operation can return different things, which may be different from the return type of the whole program. Using this generalised version, anything described in the shape of the op-type automatically gains support for do-notation as well as the type checker verifying that the program only changes states in accordance with the specification.

4.2 Generic traces

Taking the same approach as with programs, we can index the infrastructure required for the trace generation by the type of operations to make it generic. The first part is OpRes – capturing the type of an operation and the type of result it produced, along with the state it happened in and the function describing how to process the result to change state. We also need a Show instance, to show counterexamples:

```
record OpRes {0 stT : _}
             (opT :  (t' : _) -> stT -> (t' -> stT)
                  -> Type)
             (resT : Type) (currSt : stT)
             (0 nsFn : resT -> stT) where
  constructor MkOpRes
  op : opT resT currSt nsFn
  res : resT
  {auto opShow : Show (opT resT currSt nsFn)}
  {auto rShow : Show resT}
```

Both TraceStep and Trace follow the same pattern:

```
record TraceStep (opT :  (t' : _) -> stT
                      -> (t' -> stT) -> Type) where
  constructor MkTS
  {0 stepRT : _}
  {0 stepSt : stT}
  {0 stepFn : stepRT -> stT}
  opRes : OpRes opT stepRT stepSt stepFn
  resSt : stT
  {auto showStT : Show stT}

data Trace :  (opT :  (t' : _) -> stT
                   -> (t' -> stT) -> Type)
           -> stT -> Nat -> Type where
  MkTrace : Show stT => (initSt : stT) -> {bound : Nat}
         -> (trace : Vect bound (TraceStep opT))
         -> Trace opT initSt bound
```

4.3 The Traceable interface

To generate traces, we need to know which operations are valid given a current state. In the specific instance, we could define this directly as part of the Arbitrary implementation. In the general case, however, it is not possible to know ahead of time which operations are allowed in what state. We therefore introduce the Traceable interface. An operation is *traceable* if for some given current state, we can produce a list of generators producing valid transitions away from that state, paired with a result they might have produced, i.e. generators of dependent pairs for various OpRes types. To allow the user to specify likeliness of the different outcomes, they must also provide the weights of each generator.

```
interface Traceable (0 opT : (t' : _) -> stT
                     -> (t' -> stT) -> Type) where
  options : (st : stT) -> List (Nat, Gen
              (resT : Type ** nsFn : resT -> stT
                ** OpRes opT resT st nsFn))
```

For cases where the implementations are straightforward, in order to alleviate the development overhead of using Gen, we provide some helper functions to make programming more ergonomic:

- singular — When there is only one option, this allows the user to state that option without any of the weighting or Gen-wrapping.
- choice — For lists of options. This wraps each option in a pure call and weights them evenly.
- weighted — For lists of options where the weighting of choices matter. It takes a list of weightings and a list of options, the lengths of which must be the same, wraps the options in pure and zips them up with their respective frequencies.

```
singular :  {0 stT : _}
         -> {0 opT : (t' : _) -> stT -> (t' -> stT)
              -> Type} -> {st : stT}
         -> (resT : Type ** nsFn : resT -> stT **
             OpRes opT resT st nsFn)
         -> List (Nat, Gen (resT : Type **
                           nsFn : resT -> stT **
                           OpRes opT resT st nsFn))
singular option = [(1, pure option)]

choice :  {0 stT : _}
       -> {0 opT : (t' : _) -> stT -> (t' -> stT)
            -> Type} -> {st : stT}
       -> List (resT : Type **
                nsFn : resT -> stT **
                OpRes opT resT st nsFn)
       -> List (Nat, Gen (resT : Type **
                         nsFn : resT -> stT **
                         OpRes opT resT st nsFn))
choice xs = map (MkPair 1) (pure <$> xs)

weighted :  {0 stT : _}
         -> {0 opT : (t' : _) -> stT -> (t' -> stT)
              -> Type} -> {st : stT}
         -> (weights : Vect nOptions Nat)
         -> (opts : Vect nOptions
                    (resT : Type **
                     nsFn : resT -> stT **
                     OpRes opT resT st nsFn))
         -> List (Nat, Gen (resT : Type **
                           nsFn : resT -> stT **
                           OpRes opT resT st nsFn))
weighted weights opts = toList $
  zip weights (pure <$> opts)
```

For more complex examples, such as combinations of simple and composed generators, the user must specify the options explicitly. However, the option of being able to express more complicated generators whilst still using our framework shows the strength of the approach: both complicated models and test generators can be implemented in the same file; indeed, one can use the full power of higher-order functions to simplify some things, for example map (MkPair 1) works for converting a list of complex generators into a choice of generators.

4.4 Arbitrary for generic ISMs

With the supporting data structures and records generalised, we can implement a version of Arbitrary which will work for *any* ISM that implements Traceable. The approach is the same as used in section 3.4 and section 3.5, except with everything indexed by the type of the permitted operations.

```
{0 stT : _} -> {0 opT : _} -> {st : stT} ->
Traceable opT =>
Arbitrary (resT : Type ** nsFn : resT -> stT **
           OpRes opT resT st nsFn) where
  where
  arbitrary {st} = frequency $ options st

{0 stT : _} -> {iSt : stT} -> {bound : Nat} ->
{opT :  (t' : Type) -> stT -> (t' -> stT) -> Type} ->
Show stT =>
Traceable opT =>
Arbitrary (resT ** nsFnT **
           OpRes opT resT iSt nsFnT) =>
Arbitrary (Trace opT iSt bound)
  where
  arbitrary {bound = 0} =
    pure $ MkTrace iSt []
  arbitrary {bound = (S k)} = do
    (_ ** nsFn ** opRes@(MkOpRes op res)) <-
      the (Gen (rT ** fnT **
                OpRes opT rT iSt fnT)) arbitrary
    let fstTraceSt = nsFn res
    let traceHead = MkTS opRes fstTraceSt
    traceTail <- trace k fstTraceSt
    pure (MkTrace iSt (traceHead :: traceTail))
    where
      trace :  (steps : Nat) -> (st : stT)
            -> Gen (Vect steps (TraceStep opT))
```

```
trace 0 _ = pure []
trace (S j) st = do
  (_ ** stFn ** opR@(MkOpRes op res)) <-
    the (Gen (x ** y **
              OpRes opT x st y)) arbitrary
  let nextSt = stFn res
  pure $ (MkTS opR nextSt) ::
          !(trace j nextSt)
```

This completes the generalisation, allowing us to model, verify, implement, and test any specification as long as the states, transitions, and options from each state are given.

4.5 Evaluation: The ARQ Protocol

We evaluate our generalisation by implementing a different system, the Automatic Repeat Request (ARQ) protocol. The ARQ protocol works by sending a single packet containing some data and a packet number, and then waiting for an acknowledgement of the packet number before advancing to sending the next packet [33]. We chose ARQ because it is simple enough to be understandable, while also presenting a some interesting challenges: there is an external second party involved, whose behaviour we cannot know; and, due to packet numbering, there are potentially infinite states.

4.5.1 The states of ARQ. Naïvely, the protocol only has two states: **Ready** and **Waiting**. However, the semantics of ARQ introduce a third state, **Acked**. When we receive an acknowledgement — an `Ack` — for a certain packet, we need to check that it acknowledges the sequence number we sent and retry if the `Ack` was for another packet (potentially due to data corruption on the return trip). Checking the acknowledged sequence number can then either require us to retransmit the same packet or, if everything is fine, to proceed to sending the next packet in the sequence.

```
data ARQState = Ready Nat | Waiting Nat
              | Acked Nat Nat
```

Each state takes the current sequence number of the packet being transmitted, with **Acked** additionally taking the acknowledged sequence number so that we can verify it.

We define a simple packet record, along with a data type for capturing the possible outcomes of waiting on an acknowledgement.

```
record Pkt where
  constructor MkPkt
  pl : Bits8
  sn : Nat

data WaitRes = Ack Nat | Timeout
```

Note that this captures the fact that we cannot know how the other side will reply, if at all. We are not trying to simulate timed automata to model the exact timeouts required. Instead, we model the possible outcomes and test that our protocol is well-behaved under these scenarios.

4.5.2 The `Next` state function, uses the same predicate-based technique we saw for `ChkPINfn` to ensure we are in a **Waiting** state, and transitions to **Acked** if an `Ack`-reply was received — keeping track of both the packet number and the reply number — or immediately back to **Ready** if the reply never came, forcing us to retry sending the same packet.

```
data IsWaiting : ARQState -> Type where
  ItIsWaiting : IsWaiting (Waiting n)

Next : (st : ARQState) -> {auto ford : IsWaiting st}
    -> WaitRes -> ARQState
Next (Waiting n) {ford=ItIsWaiting} (Ack a) =
  Acked n a
Next (Waiting n) {ford=ItIsWaiting} Timeout =
  Ready n
```

4.5.3 Moving on to ARQ operations, *Send* takes a packet to send and ensures the state types keeps track of the current packet number, and *Wait* proceeds with a wait result. The more interesting transitions, *Proceed* and *Retry*, take a proof that the acknowledged number and the packet number are equal or that they cannot be equal, respectively. This adds some overhead to programming with the operations, but we chose to include this as it nicely show how dependent types integrate with our new approach:

```
data ARQOp : (t : _) -> ARQState -> (t -> ARQState)
          -> Type where
  Send : (pkt : Pkt) -> ARQOp () (Ready pkt.sn)
                          (const $ Waiting (pkt.sn))
  Wait : ARQOp WaitRes (Waiting n) (Next (Waiting n))
  Proceed : (ok : a === n)
    -> ARQOp () (Acked n a) (const $ Ready (S n))
  Retry : (Not (a === n))
    -> ARQOp () (Acked n a) (const $ Ready n)
```

4.5.4 Finally, we need a `Traceable` instance. When in the **Ready** state, we have access to the sequence number we are meant to be sending, and so we can construct an arbitrary packet and *Send* it (we use the placeholder payload of 255 rather than `arbitrary` for brevity). Once we have received an `Ack` and are in the **Acked** state, we need to check whether the two numbers are equal. If they do, the only thing we can do is to advance to sending the next packet. If they cannot be equal, the only thing we can do is to retry sending the packet. This may sound like we have no control over the frequency of accepted versus rejected acknowledgements, however we *can* control this by simulating an unreliable network from the **Waiting** state: 20% of the time we do not get a reply, timing out instead; 5% of the time we get an `arbitrary` acknowledgement; and the remaining 75% of the time we successfully transmit and get a valid acknowledgement back:

```
Traceable ARQOp where
  options (Ready k) = singular
    (_ ** _ ** MkOpRes (Send (MkPkt 255 k)) ())

  options (Waiting k) =
```

```
    [ (4, pure (_ ** _ ** MkOpRes Wait Timeout))
    , (1, do pure
        (_ ** _ ** MkOpRes Wait (Ack !arbitrary)))
    , (15, pure (_ ** _ ** MkOpRes Wait (Ack k)))
    ]

  options (Acked n a) = case decEq a n of
    (Yes prf) =>
      singular (_ ** _ ** MkOpRes (Proceed prf) ())
    (No contra) =>
      singular (_ ** _ ** MkOpRes (Retry contra) ())
```

4.5.5 This is all we need. We have now defined everything the programmer needs to define to use our new approach. Thanks to our generalisation, we can now plug our new stateful model into Prog and immediately get access to do-notation and type-level state transition verification:

```
sendN : (n : Nat)
  -> Prog ARQOp () (Ready n) (const $ Ready (S n))
sendN n = do
  Op $ Send (MkPkt 255 n)
  (Ack a) <- Op Wait
    | Timeout => sendN n
  case decEq a n of
       (Yes prf) => Op $ Proceed prf
       (No contra) => do
         Op $ Retry contra
         sendN n

prog : Prog ARQOp () (Ready 0) (const $ Ready 3)
prog = do
  sendN 0 ; sendN 1 ; sendN 2

failing "Mismatch between: 1 and 0"
  bad : Prog ARQOp () (Ready 0) (const $ Ready 2)
  bad = do sendN 1
```

Additionally, although the program above may run forever, we can increase our confidence that it will not. Traceable allows us to use type-level QuickCheck, meaning we can write a property and check it at compile-time:

```
0 PROP_sendThreeOK : Fn (Trace ARQOp (Ready 0) 20)
                        Bool
PROP_sendThreeOK = MkFn (\case (MkTrace _ trace) =>
  elem (Ready 3) $ (.resSt) <$> trace)

0 QC_sendThreeOK : So (QuickCheck False
                       PROP_sendThreeOK)
QC_sendThreeOK = Oh
```

The trace is to a depth of 20 because it takes at least 3 transitions to reliably send a single packet. Since there are no reported mismatches between True and False on file loading, we know that the property holds. While we have not *proven* that our program is guaranteed to terminate, we have increased our confidence that it does, without having to leave the language or modelling framework we are already using, and with a guarantee that the types, program, and test all use the same model and rules.

5 Evaluation & Further Work

The types, state transitions, and Traceable implementation for ARQ come to around 50 lines of code, in contrast to the nearly 80 we had to write just for the supports for the ATM in section 3.3.2. These are tricky lines of code — weaving the state in the types and making sure the trace generation follows the correct sequence — so having them in a generalised form saves us from the risk of incorrectly reimplementing them, in addition to also saving us a lot of tedious work.

It could be argued that most DSLs are simple enough to manually reason about. However, as we have seen, seemingly trivial DSLs like the one for the ATM are easy to get wrong and this mistake can remain undetected for years. For more involved use cases like network protocols, resource management, or concurrency, where the host type system is leveraged to provide certain desirable guarantees for the target domain [7, 13, 14, 16, 25, 28], the risk of the DSL accidentally introducing subtle inconsistencies and unintended behaviour, is likely to be much higher, weakening the goal of eliminating certain bugs by using a stricter host language. Combined with the ubiquity of stateful systems, we therefore believe that our approach is worthwhile and intend to model and test more advanced protocols in the future.

Throughout the paper, all the traces have had seemingly magic numbers as their bound. The numbers were determined partially on reasoning: it is possible to get a good estimate of the depth needed by taking the number of transitions in the ISM, deciding on an upper bound for when the system should be in the desired state, and multiplying this by the number of transitions (or the number of states) in the system. Should the properties fail, they can be examined to either reveal a valid error case, possibly a fault in the generators, or a false positive caused by the model (as happened in [32]). For true positives the depth can be doubled until they pass, at which point a binary search can be used to find the smallest valid bound. We believe this to be part of the confidence building: the programmers can increase the bound until they are confident in their typed models, or they can decide that the current bound is sufficient. In our experience, this is not a hindrance, as the properties fail quickly, thereby quickly finding the initial upper bound.

The type-level PBT for the ARQ model takes around 3.5 seconds on a reasonably modern laptop[1]. While this may seem slow, it is worth remembering that the type checker is doing a lot of work. It is solving interface constraints, unifying values, running a PRNG, and doing equality checks for non-trivial types at least 100 times. The Idris2 type checker is currently the main bottleneck to our approach and presents

[1]x86_64 Intel Core i7-8750H @ 2.20GHz, boosting to ~4.08GHz, with 32GiB of SODIMM DDR4 RAM @ 2667MT/s

many interesting research questions in terms of how to speed up the elaboration process, when to expand and inline certain functions and datatypes, and how much information to keep around in the type checker and elaborator.

In the future, we plan to examine and implement increasingly more advanced systems. ARQ with Sliding Window [33] would be a nice extension to the ARQ example, as it improves the throughput of the protocol and presents some new challenges for the state function: how do we best model the packet window's movements? Pick and Place machines used for automatic placement of surface mounted components [1], file systems [18], and protocols with crash-stop failures [4], are all stateful systems which will present interesting modelling challenges as well as provide us with more performance and usability data.

All the code used in this paper is publicly available at: https://github.com/CodingCellist/tyde-24-code

6 Conclusion

We successfully implemented QuickCheck in Idris2 and demonstrated how it can be extended to generate arbitrary instances of dependent types. We then highlighted how dependent types allow us to model stateful systems, that these models are tricky to get right, and how we can use QuickCheck at the type-level to automatically detect this and help us fix it. Finally, we generalised the type-level framework to support any stateful system, and demonstrated its usefulness by modelling, implementing, and testing a network protocol. Our approach is not a proof system, however it should help prototype specifications and models faster, gaining confidence that the current system is sound, before potentially choosing to model check or to formalise and prove it. We believe that our generalisation is low-friction enough for wider adoption and are excited to see what this may lead to.

References

[1] Gokulnath A R, Chandrakumar S, and Sudhakar T D. 2018. Open Source Automated SMD Pick and Place Machine. *Procedia Computer Science* 133 (2018), 872–878. https://doi.org/10.1016/j.procs.2018.07.107

[2] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR Software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Graz, Austria, 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

[3] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Oxford United Kingdom, 56–65. https://doi.org/10.1145/3209108.3209189

[4] Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. 2023. Designing Asynchronous Multiparty Protocols with Crash-Stop Failures. https://doi.org/10.48550/arXiv.2305.06238 arXiv:2305.06238 [cs]

[5] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems III*, G. Goos, J. Hartmanis, J. Van Leeuwen, Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag (Eds.). Vol. 1066. Springer Berlin Heidelberg, Berlin, Heidelberg, 232–243. https://doi.org/10.1007/BFb0020949

[6] S. Berghofer and T. Nipkow. 2004. Random Testing in Isabelle/HOL. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.* IEEE, Beijing, China, 230–239. https://doi.org/10.1109/SEFM.2004.1347524

[7] Saleem Bhatti, Edwin Brady, Kevin Hammond, and James McKinna. 2009. Domain Specific Languages (DSLs) for Network Protocols (Position Paper). In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, Montreal, Quebec, Canada, 208–213. https://doi.org/10.1109/ICDCSW.2009.64

[8] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Vol. 5674. Springer Berlin Heidelberg, Berlin, Heidelberg, 73–78. https://doi.org/10.1007/978-3-642-03359-9_6

[9] Edwin Brady. 2013. Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 5 (Sept. 2013), 552–593. https://doi.org/10.1017/S095679681300018X

[10] Edwin Brady. 2015. Resource-Dependent Algebraic Effects. In *Trends in Functional Programming (Lecture Notes in Computer Science, Vol. 8843)*, Jurriaan Hage and Jay McCarthy (Eds.). Springer International Publishing, Cham, 18–33. https://doi.org/10.1007/978-3-319-14675-1_2

[11] Edwin Brady. 2017. *Type-Driven Development with Idris*. Manning Publications Co, Shelter Island, NY.

[12] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. *arXiv* (April 2021). arXiv:2104.00480 https://arxiv.org/abs/2104.00480

[13] Edwin Brady and Kevin Hammond. 2010. Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols. *Fundamenta Informaticae* 102, 2 (2010), 145–176. https://doi.org/10.3233/FI-2010-303

[14] Edwin Brady and Kevin Hammond. 2012. Resource-Safe Systems Programming with Embedded Domain Specific Languages. In *Practical Aspects of Declarative Languages*, Claudio Russo and Neng-Fa Zhou (Eds.). Vol. 7149. Springer Berlin Heidelberg, Berlin, Heidelberg, 242–257. https://doi.org/10.1007/978-3-642-27694-1_18

[15] F. Warren Burton and Rex L. Page. 1992. Distributed Random Number Generation. *Journal of Functional Programming* 2, 2 (1992), 203–212. https://doi.org/10.1017/S0956796800000320

[16] David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for Certified Multiparty Computation: From Mechanised Metatheory to Certified Multiparty Processes. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 237–251. https://doi.org/10.1145/3453483.3454041

[17] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 1–30. https://doi.org/10.1145/3485500

[18] Zilin Chen, Christine Rizkallah, Liam O'Connor, Partha Susarla, Gerwin Klein, Gernot Heiser, and Gabriele Keller. 2022. Property-Based Testing: Climbing the Stairway to Verification. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022)*. Association for Computing Machinery, Auckland New Zealand, 84–97. https://doi.org/10.1145/3567512.3567520

[19] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[20] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. Finding Race Conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 149–160. https://doi.org/10.1145/1596550.1596574

[21] Koen Claessen and Michał H. Pałka. 2013. Splittable Pseudorandom Number Generators Using Cryptographic Hashing. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/2503778.2503784

[22] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2001. Progress on the State Explosion Problem in Model Checking. In *Informatics*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Reinhard Wilhelm (Eds.). Vol. 2000. Springer Berlin Heidelberg, Berlin, Heidelberg, 176–194. https://doi.org/10.1007/3-540-44577-3_12

[23] Edmund M Clarke. 2008. Model Checking – My 27-Year Quest to Overcome the State Explosion Problem. (2008), 1. https://doi.org/10.1007/978-3-540-89439-1_13

[24] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification*, Bertrand Meyer and Martin Nordio (Eds.). Vol. 7682. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–30. https://doi.org/10.1007/978-3-642-35746-6_1

[25] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. 2004. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky (Eds.). Vol. 3016. Springer Berlin Heidelberg, Berlin, Heidelberg, 51–72. https://doi.org/10.1007/978-3-540-25935-0_4

[26] S. Demri, F. Laroussinie, and Ph. Schnoebelen. 2006. A Parametric Analysis of the State-Explosion Problem in Model Checking. *J. Comput. System Sci.* 72, 4 (June 2006), 547–575. https://doi.org/10.1016/j.jcss.2005.11.003

[27] Maxime Dénès, Cătălin Hriţcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-Based Testing for Coq. http://prosecco.gforge.inria.fr/personal/hritcu/talks/coq6_submission_4.pdf

[28] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2018. Pi-Ware: Hardware Description and Verification in Agda. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:27. https://doi.org/10.4230/LIPIcs.TYPES.2015.9

[29] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[30] G.J. Holzmann. 1997. The Model Checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[31] John Hughes. 2007. QuickCheck Testing for Fun and Profit. In *Practical Aspects of Declarative Languages*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Michael Hanus (Eds.), Vol. 4354. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–32. https://doi.org/10.1007/978-3-540-69611-7_1

[32] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Vol. 9600. Springer International Publishing, Cham, 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

[33] Shu Lin and Daniel J. Costello. 1983. *Error Control Coding: Fundamentals and Applications* (nachdr. ed.). Prentice-Hall, Englewood Cliffs, NJ.

[34] Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Vol. 9600. Springer International Publishing, Cham, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

[35] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[36] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, London UK, 763–779. https://doi.org/10.1145/3385412.3386036

[37] Hans Georg Schaathun. 2015. Evaluation of Splittable Pseudo-Random Generators. *Journal of Functional Programming* 25 (2015), e6. https://doi.org/10.1017/S095679681500012X

[38] Guy Steele and Sebastiano Vigna. 2021. Computationally Easy, Spectrally Good Multipliers for Congruential Pseudorandom Number Generators. https://doi.org/10.48550/arXiv.2001.05304 arXiv:2001.05304 [cs]

[39] Antti Valmari. 1998. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Wolfgang Reisig, and Grzegorz Rozenberg (Eds.). Vol. 1491. Springer Berlin Heidelberg, Berlin, Heidelberg, 429–528. https://doi.org/10.1007/3-540-65306-6_21

[40] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294. https://doi.org/10.1145/1993316.1993532

Preprint — do not distribute.