abcBridge

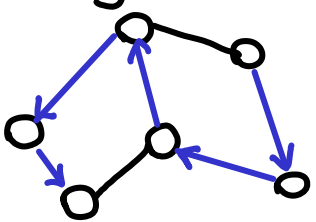
Functional Interfaces for AIGs & SAT Solving

Edward Z. Yang
Galois, Aug 24, 2010

This is a presentation about abcBridge, a set of Haskell bindings for ABC that I wrote over the course of my summer internship at Galois.

Many, many NP problems...

Traveling Salesman



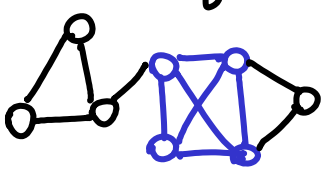
Knapsack

$$\max \sum_{i=1}^{n} v_i x_i \text{, s.t. } \sum_{i=1}^{n} \omega_i x_i \leq W$$

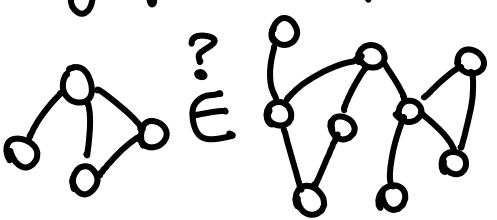
SAT

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge \cdots$$

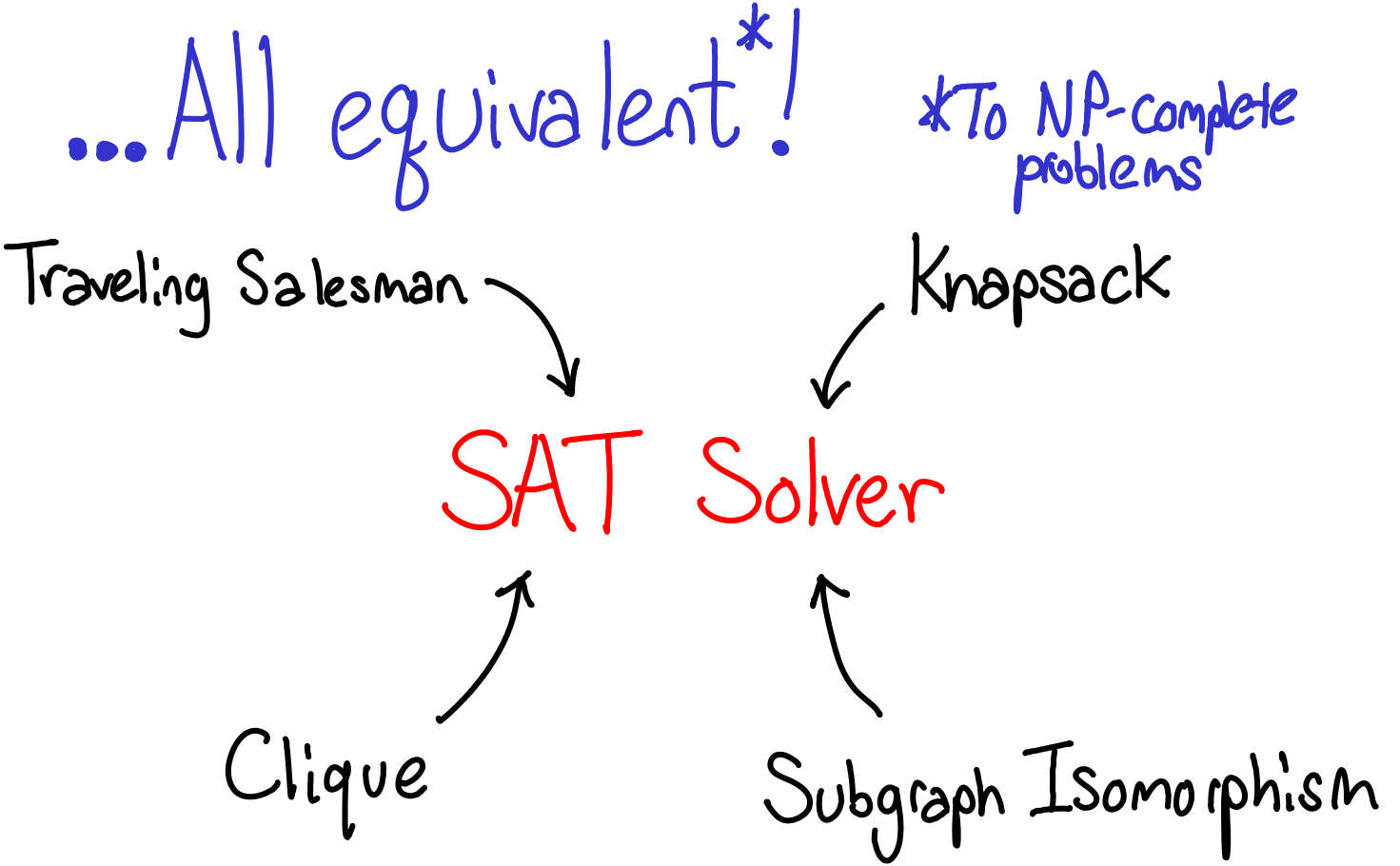
Clique



Subgraph Isomorphism

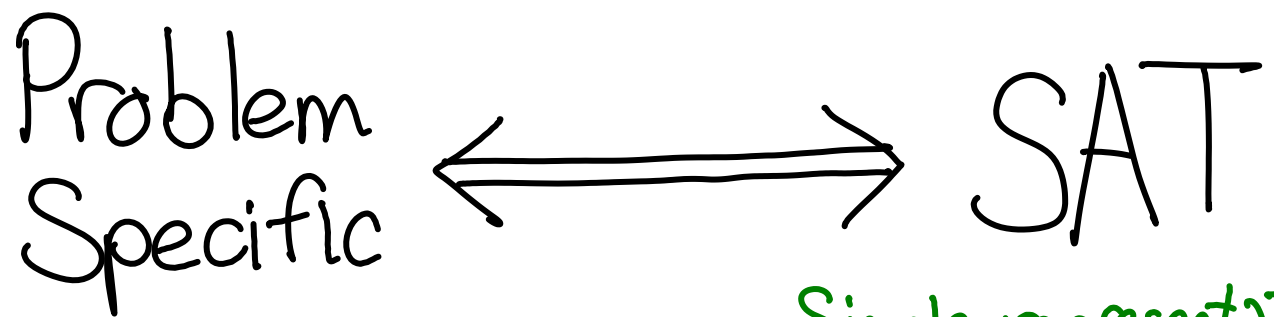


There are many, many NP problems in this world that occur in many, many practical situations. One might think that we'd have to devote endless hours to writing algorithms for each and every one of them...


...All equivalent*!
*To NP-complete problems
Traveling Salesman
Knapsack
SAT Solver
Clique
Subgraph Isomorphism

...but in fact, they are all equivalent to a class of problems called NP-complete. This means that we can algorithmically translate a representation of any problem (say, "Traveling Salesman") into another problem (canonically, "Boolean Satisfiability") and then ask a solver for that problem to solve the problem for us. Tricks that we implement in one problem domain can help us out in other places.

Tension



Build it yourself

Complex Representation

Better Heuristics

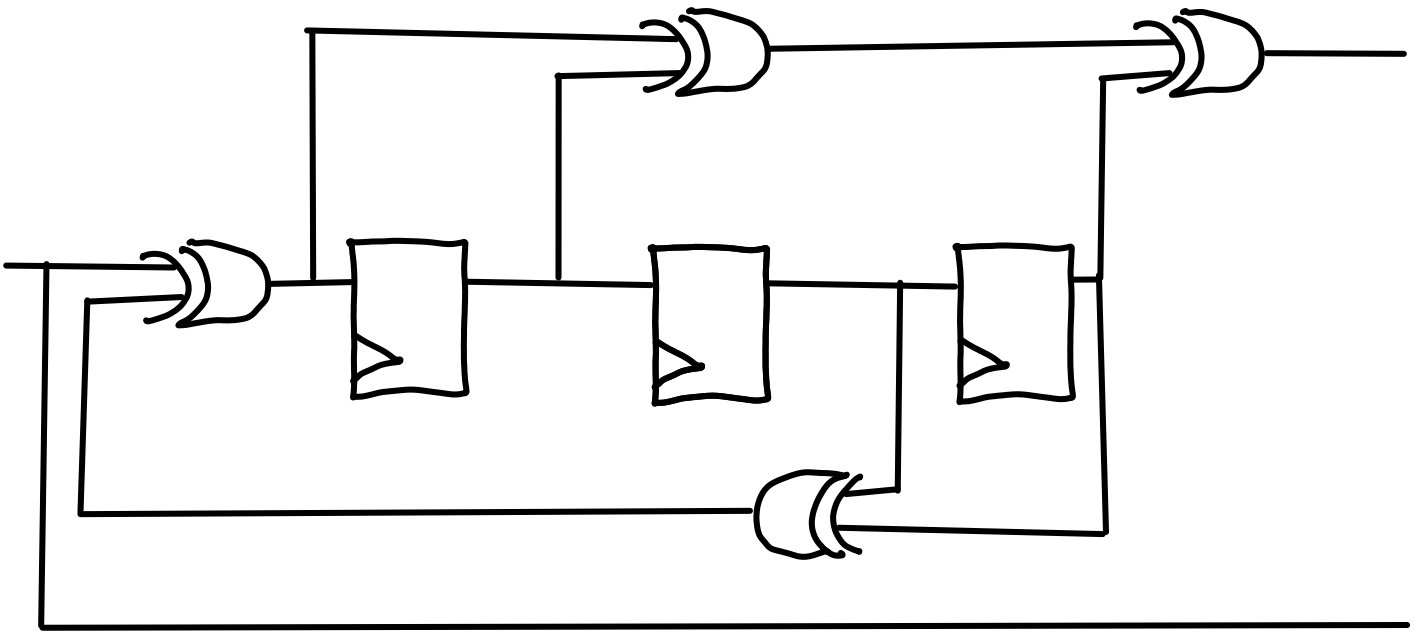
Simple representation

Off-the shelf solvers

Loss of high-level structure

SAT was the first NP-complete problem to be proved, well, NP-complete, and lends itself well to our computers, so it has been one of the most thoroughly studied of any NP-complete problems. There are SAT solving competitions, and behind almost any program that solves NP problems you will find a SAT solver to do the dirty work. However, when we translate any given problem into a list of boolean clauses, we are effectively throwing away the high level structure of the original problem instance, which may have allowed us to apply smarter heuristics and get the answer faster. Thus, there is a tension between the simple SAT representation, and the more complex problem-specific representation.

Problem Domain = Logic Circuits



½ RATE CONVOLUTIONAL ENCODER

ABC and abcBridge look at one specific problem domain: the domain of logic circuits. Here, we see a simple convolutional encoder composed of elementary logic gates (XORs) and three latches, which constitute the state of the encoder.

Target Audience

our reason!

Hardware circuit designers

Cryptographic Implementors (Crypto!!)

Programmers who know logic circuits and have NP problems* to solve!

*easily translated to logic circuits

Logic circuits are of paramount importance to hardware circuit designers. However, at Galois we've had great success applying converting cryptographic algorithms into logic circuits and then verifying them with what are traditional electrical engineers. The logic circuit domain is one closely associated with SAT, but probably far more well-known to programmers who have their own NP problems to solve.

BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP

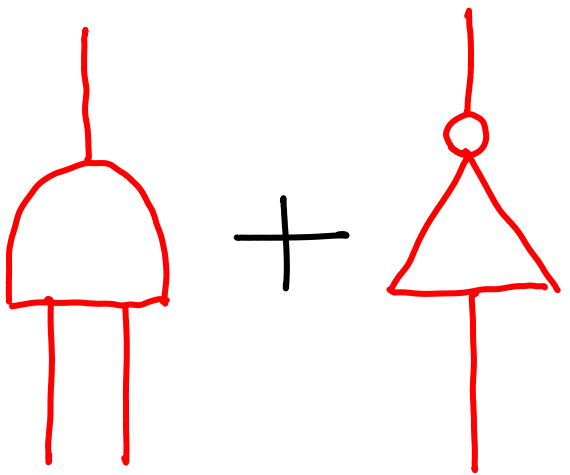
ABC: A System for Sequential Synthesis and Verification

http://www.eecs.berkeley.edu/~alanmi/abc/

Enough background, so what is ABC? ABC is a "System for Sequential Synthesis and Verification." I've highlighted "Sequential" and "Verification": "Sequential" means that ABC has support for handling circuits with latches, i.e. state. I've highlighted "Verification" because verification that two circuits are equivalent is precisely where our SAT solver will come in handy. ABC is public domain software from the Berkeley Logic Synthesis and Verification Group, and is primarily developed by Alan Mischenko.

Internal Representation = AIG

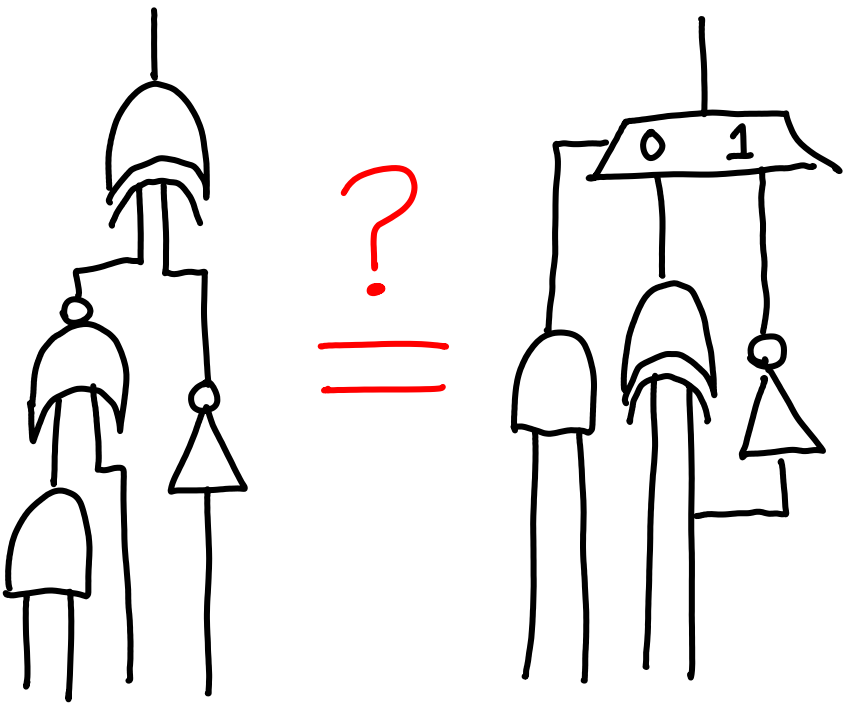
UNIVERSAL
SIMPLE
UNIFORM
FRAIGable



AND - INVERTER
GRAPH (+ LATCHES)

Like most systems, ABC is internally backed by a SAT solver. However, for its domain specific algorithms, it adopts a different representation than the usual conjunctive-normal form (a highly regular representation that uses only ANDs and ORs). ABC uses AIGs: And-Inverter graphs: graphs that contain only AND gates and NOT gates (you may recall that NAND gates are universal: you can simulate any logical formula with them.) AIGs are a simple representation that can be easily translated into SAT. It is also an incredibly uniform representation (after all, there is only one type of multiple fan-in node to worry about), which makes implementing algorithms for it easier. Finally, and this is the secret sauce that makes ABC so much better at finding satisfying instances than pure SAT solvers, is that this representation is amenable to the functional reduction of AIGs--known as fraiging--in which functionally equivalent segments of a circuit are coalesced. When you ask ABC to verify two circuits, it will spend some time fraiging before passing it to the SAT solver.

Verification = Circuit Equivalence

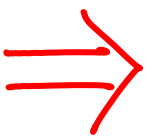


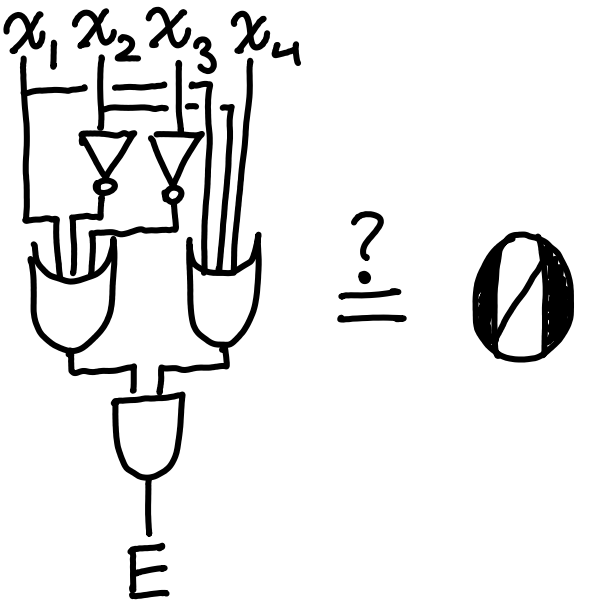
What does it mean to verify a circuit? If we are optimizing a circuit, we would like to make sure each optimized version is equivalent to the original version. So verification is equivalence checking.

Circuit Non-equivalence is NP-hard

$\exists x_1, x_2, x_3, x_4 . E = \text{true}$

$E = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_4)$





Boolean Formula (SAT)

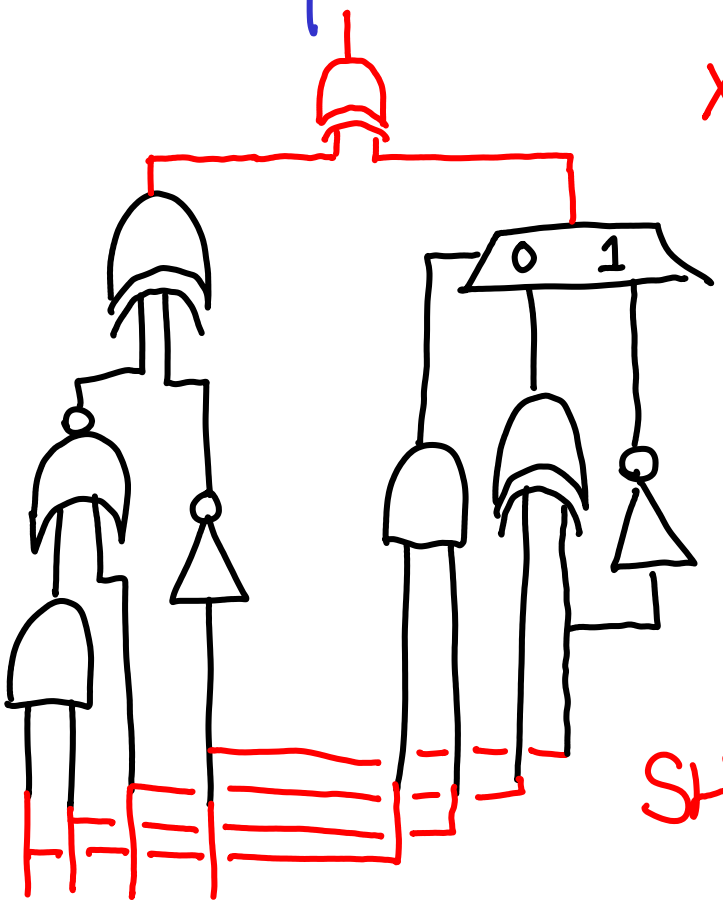
Logic Circuit (Non-equivalence)

Circuit equivalence is NP-hard: we can easily translate instances of SAT into it. Just treat the set of boolean clauses as a logic circuit, and check if it's equivalent to the zero circuit: the circuit that always outputs False. It will be inequivalent if there is a satisfying instance.

Circuit Non-equivalence is NP-complete

XOR the outputs

MITER the two circuits



SHARE the inputs

Circuit equivalence is NP-complete: we can do the translation in the other direction (which is what ABC does.) To do this, we compute the miter of two circuits, which gives us a single circuit with a single output that outputs True if the two circuits give different outputs.

What abcBridge does...

SYNTHESIS (building the network)

&

VERIFICATION (checking equivalence)

in Haskell!

So what does abcBridge do? It gives you "Synthesis": a way of building the logic network, and "Verification", a way to check that networks are equivalent. And it lets you do this all in our favorite programming language, Haskell!

A quick taste of verification:

```
import Data.ABC
import qualified Data.ABC.AIG as AIG

main = withABC $ do
    n1 <- AIG.readAiger "n1.aig"
    n2 <- AIG.readAiger "n2.aig"
    print (AIG.cec n1 n2)
```

IO

combinational equivalence

The combinational verification interface is extremely simple: we have functions for reading AIGs from a binary file format called AIGER, and we can simply pass them to a pure function 'cec' which will test if the two circuits are equivalent.

```
$ runghc readTest.hs
Pass ()   # equivalent
$ runghc readTest2.hs
Fail [True, False, True, True]
# not equivalent : counter-example
$ runghc readTest3.hs
Intermediate ()  # no idea! (solving failed)
```

The output of this function is tristate: we can either Pass (in the case of equivalence), Fail with a counterexample (in the case of non-equivalence), or fail, letting the user know we ran out of resources before we could figure out one way or another.

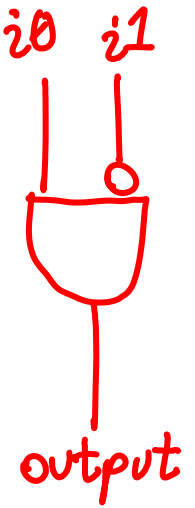
Simple synthesis...

```
main = withABC $ do
        n1 ← AIG.readAiger "n1.aig"
        n2 ← AIG.readAiger "n2.aig"
        let m= AIG.miter n1 n2
        AIG.writeAiger "m.aig" m
```

We can synthesize new networks without needing gates: the 'miter' function computes the miter of two networks, which we can then save to a file for further inspection.

Complex synthesis...

```
import Data.ABC.AIG (AIG)
network :: AIG
network = AIG.runNT $ do
        i0 ← AIG.input
        i1 ← AIG.input
        n ← AIG.and i0 (AIG.not i1)
        AIG.output n
```



NT "Network Transform" Monad

If we want to build up our network gate-by-gate, we need to bust out a little more machinery. Network creation is monadic: we create inputs, create gates and assign outputs inside the NT monad, standing for "network transformation." We run the NT monad using the "runNT" function.

Primitives

n ~ Phantom type (like ST s)

```
one :: Node n
zero :: Node n
and :: Node n → Node n → NT n (Node n)
not :: Node n → Node n
input :: NT n (Node n)
output :: Node n → NT n ()

latch :: Node n → Bool → NT n (Node n)
```

Here are the primitive operations you can run inside the NT monad. Notice that the type of node is 'Node n', not 'Node': the 'n' is a phantom type that ensures that we don't mix up nodes from one network with another.

Running the monad

```
runNT :: (∀n. NT n ()) → AIG
withNT :: AIG → (∀n. NT n ()) → AIG
```

Phantom type prevents Node mix-ups

```
runNT $ do
    n1 ← input
    let foo = runNT $ do
            output n1  # type error!
```

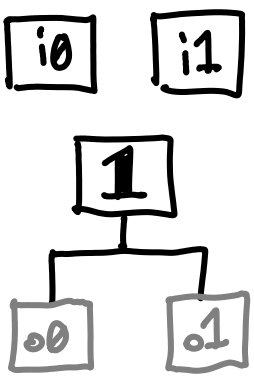
To actually get this assurance, we use rank-2 types in our functions that run the monads. Here we see 'runNT' with it's type signature: in this case, we don't care about preventing nodes from leaking out of scope, since our functions don't let you smuggle them out.

The NT monad captures mutable state

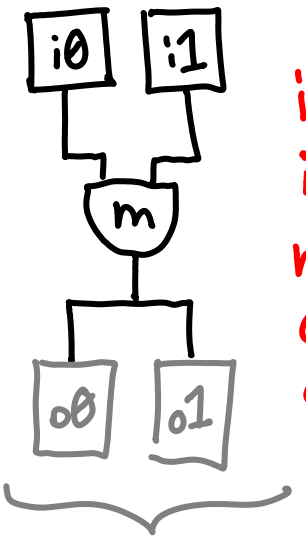
```
replace :: Node n → Node n → NT n ()
replaceOutput :: Int → Node n → NT n ()
deleteInput :: Int → NT n ()
deleteOutput :: Int → NT n ()

setLatchDriver :: Node n → Node n → NT n ()
```

Since the NT monad is actually all mutation under the hood, abcBridge also offers some functions for modifying a network as you build it. These come in handy if your modifying an already existing network, or sometimes they let you do operations that would not normally be possible, for example, making a latch feed itself.

replace v. replaceOutput

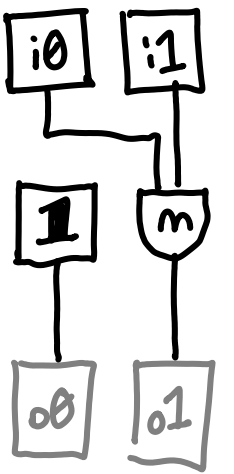


replace m one



i0 ← input
i1 ← output
m ← and i0 i1
output m
output m

NOT IN SCOPE (EVER)



replaceOutput 0 one

Here's one small detail about the difference between replace and replaceOutput. Any node that is replaced may feed arbitrarily many other nodes, in which case all of those nodes are replaced. replaceOutput lets us tweak the output value of just one node.

NQ "Network Query" Monad

```
queryNQ :: AIG -> (∀n . NQ n a) -> a
askNQ :: NQ n AIG

cone :: Node n -> NQ n AIG
run :: [Bool] -> NQ n [Bool]
```

Where does Node n come from?

After we've built up networks, we can run them through the equivalence checker, or we can make more fine-tuned queries on them. The network query monad lets us do things like calculate the logic cone of a node (the subset of the network that directly influences that node) or run the network. Of course, in order for this to be useful, we need a way to introduce nodes into the NQ monad (remember that the phantom types prevent us from mixing nodes from NT and NQ.)

NetworkMonad: NT & NQ

NetworkMonad m AIG n ⇒ (NTn OR NQ n)

```
getInput :: Int → m AIG n (Node n)
getOutputFanin :: Int → m AIG n (Node n)

numInputs :: m AIG n Int
numOutputs :: m AIG n Int
numLatches :: m AIG n Int
```

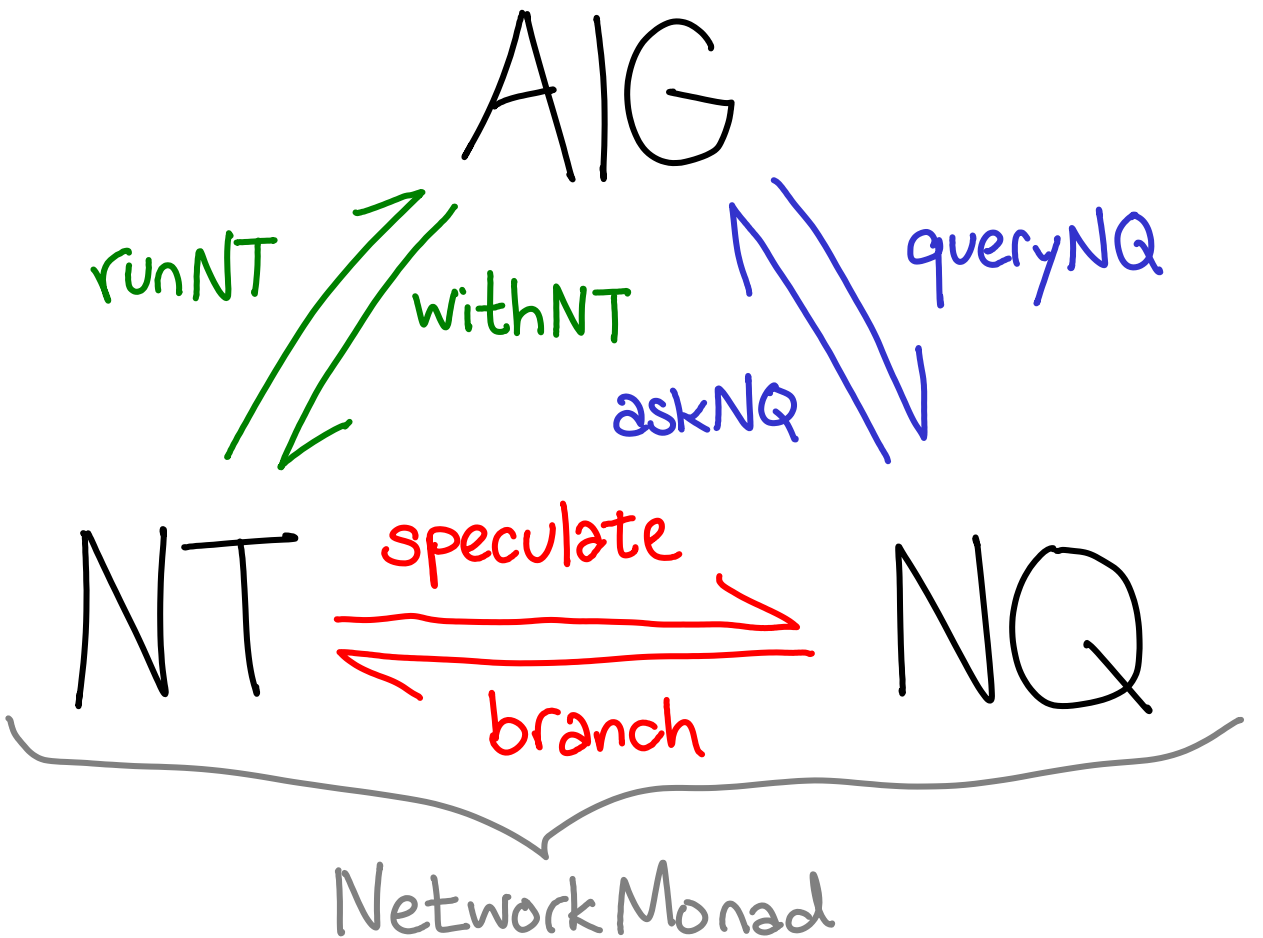
To this effect, we have a NetworkMonad typeclass, which offers functions that work in both NT and NQ.

NT ⟷ NQ

```
type Dup n n2

branch :: (∀n2. NT (Dup n n2) ()) → NQ n AIG
speculate :: (∀n2. NQ (Dup n n2) a) → NT n a

translate :: Node n → m AIG (Dup n n2) (Node (Dup n n2))
```

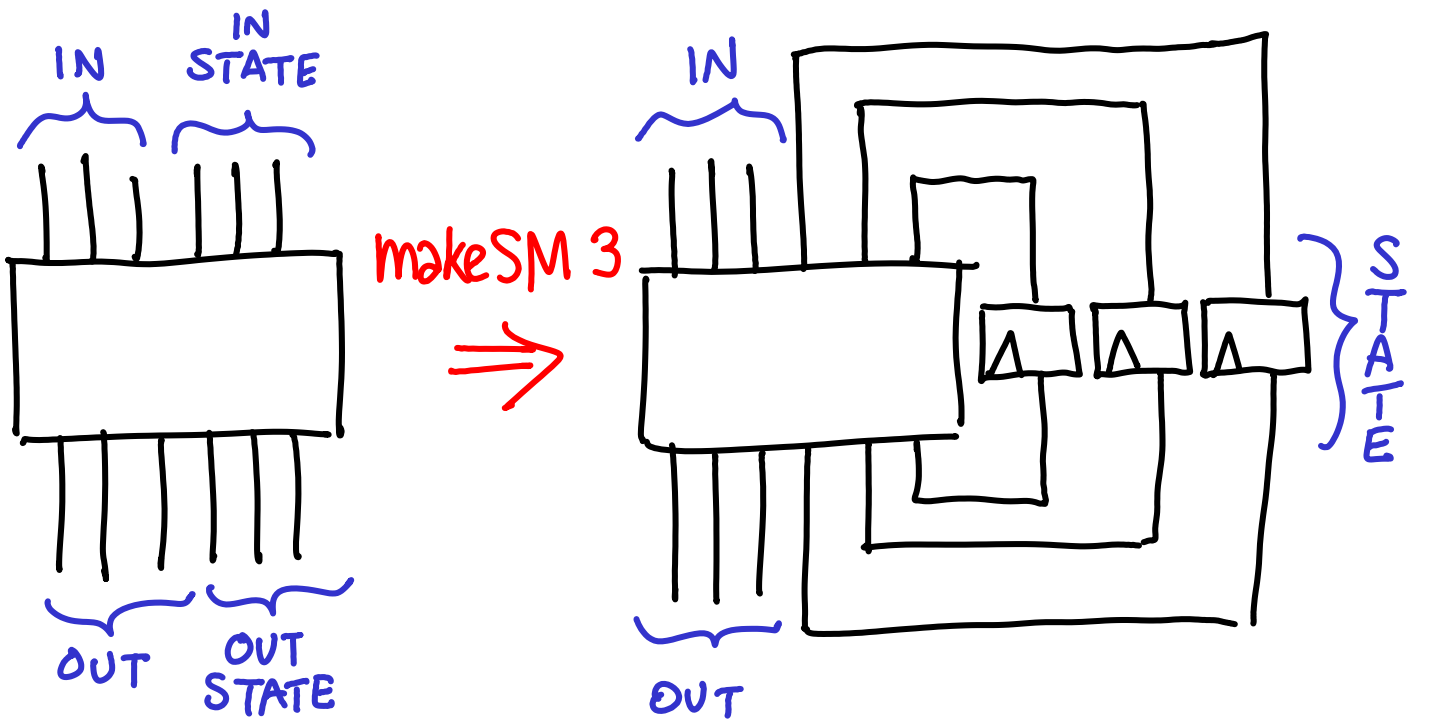
↑ MORE ON THIS LATER

There might be circumstances when you are building a network in the NT monad, and decide, "I want to take a look at what I have so far" with some operation in the NQ monad. In this case, you can use 'speculate' to drop you into the NQ monad with the current partially formed network. You can use 'branch' to go in the other direction. The types are a little complicated, and I will discuss them in depth later.


AIG
runNT
withNT
queryNQ
askNQ
NT
speculate
branch
NQ
Network Monad

Here is the flow between NT, NQ and AIG.

Extended Example: makeSM



Here is an extended example of how you might use the API. Consider a purely combinational circuit that simulates state through a number of input and output ports. We'd like to hook those ports up to latches, so our circuit is intrinsically stateful.

Extended Example: makeSM

```
makeSM n = do
    inCount ← numInputs; outCount ← numInputs
    let inState@(li, ri) = ( inCount-n, inCount-1)
        outState@(lo, ro) = (outCount-n, outCount-1)
    latches ← forM [li...ri] $ \o →
        outFanin ← getOutputFanin o
        latch outFanin False
    forM_ (zip [lo...ro] latches) $ \(i,l) →
        inputNode ← getInput i
        replace inputNode l
    deleteOutputRange outState; deleteInputRange inState
```

Here's the code that does it. The first three lines calculate the ranges of the the input and output state ports, the next three lines generate the latches and hook them up to the state output ports, the further three lines replace the state input ports with the outputs of the latches, and then we delete the (now unused) input and output state ports.


Questions?

Questions?

Part 2

Fitting an Imperative Peg in a Functional Hole

Part two of my presentation dives into the internals of abcBridge. ABC is a very imperative, shared state library, but I've managed to package it up in something that smells like Haskell. There were a lot of things to take care of to make this happen.

Imperative Complications

- → Manual Memory Management
- → Shared state
- → Hidden Invariants
- → static inline
- → Interruptibility

I'm going to talk about five particular issues that came up while writing abcBridge. The first four are examples of issues that the library can hide or statically typecheck on the user's side; the last is one case where my valiant efforts failed to shield the user.

Some conventions...

```
data Abc_Ntk_t_
```

TRAILING UNDERSCORE

```
type Abc_Ntk_t = Ptr Abc_Ntk_t_
```

C Identifier → HS Identifier

Abc_NtkDelete ⇒ abcNtkDelete

&Abc_NtkDelete ⇒ p_abcNtkDelete

First, some conventions. We indicate the raw struct in Haskell types by appending a trailing underscore, whereas a pointer to the type doesn't have the trailing underscore. If we have an identifier in C for a function, we translate it into a Haskell identifier by lower-casing the first character and removing the underscores. Function pointers are prefixed with 'p_'.

Manual Memory Management

```
Abc_Ntk_t n = Abc_NtkAlloc();
```

POINTER TYPE

```
:

Abc_NtkDelete(n);
```

C code is all about manual memory management: if you allocate it,
you have to free it. Gritty and prone to memory leaks and double frees.

Garbage Collection

```
data AIG = AIG (ForeignPtr Abc_Ntk_t_)

make :: Abc_Ntk_t -> IO AIG
make p = do
    fp <- newForeignPtr_ p
    addForeignPtrFinalizer p_abcNtkDelete fp
    return (AIG fp)
```

TRAILING UNDERSCORE

Since Haskell has a garbage collector, it would be a shame if we didn't use it! We can do this by converting pointers to structures that we normally would have had to manually track into foreign pointers, and give them a finalizer that frees them when all references go away.

What if ABC wants to free it?

```
int Abc_NtkIvyProve( Abc_Ntk_t ** ppNtk )
{ Abc_Ntk_t *pNtk, *pNtkTemp = *ppNtk;
  ...
      pNtk = Abc_NtkIvyAfter(pNtkTmp);
      Abc_NtkDelete(pNtkTemp);
  ...
  *ppNtk = pNtk;
  return RetValue; }
```

Now, once we've put a pointer into a foreign pointer, it's there forever: we can't GHC to stop memory managing it (with some caveats.) So what do you do if you need to pass the original pointer into a function that destroys it and gives you a replacement? We'd need to update every copy of the foreign pointer floating around in memory.

ForeignPtr to a Ptr

```
data AIG = AIG (ForeignPtr Abc_Ntk_t)

make :: Abc_Ntk_t -> IO AIG
make p = do
    fpp <- mallocForeignPtr
    withForeignPtr fpp $ \fp ->
        poke fp p
    addForeignPtrFinalizer p_abcNtkPtrDelete fpp
    return (AIG fpp)
```

This is difficult, so instead, we make the foreign pointer point to a pointer that points to the struct. We can then poke the inner pointer when we need to change over, and this change will be reflected in all of the foreign pointers (since they are pointing to the memory location you just changed.) If you are writing a concurrent application, use an MVar instead.

Shared State

Monad captures state transformation

```
runNT f = unsafePerformIO
            (performNT new f)
```

Type system critical to safety

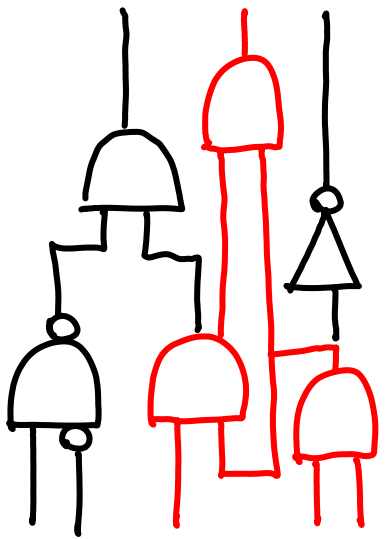
abcBridge uses unsafePerformIO under the hood to modify (hopefully) local state. The type system ensures that the local state never leaks out.

With unsafePerformIO comes responsility

```
data AIG=AIG {
      aigPtr :: ForeignPtr Abc_Ntk_t
    , aigTempLock :: MVar ()
    , aigTravLock :: MVar ()
    }
```

However, while monads may be a good API for initially building the data structure, when we stop mutating our structure we'd like to put away the monad and use a pure interface. Unfortunately, our C code may do more mutation to hidden shared state.

aigTravLock for Traversal



NOT TRAVERSED

TRAVERSED

```
struct Abc_Obj_t_ {
    int TravId;
```

```
cone x =
  ...
  takeMVar (aigTravLock n)
  pr ← abcNtkCreateCone
       a pNode "po0" True
  putMVar (aigTravLock n) ()
```

USE THE SOURCE

Here is one example: ABC's depth-first traversal to calculate the logic cone of a node. As ABC walks the tree, it marks the TravId field in nodes to avoid having to retraverse nodes that it has already seen. This means that only one traversal can happen at a time, and so we must take out a lock on the field.

Hidden Invariants

```
Abc_Obj_t *pObj = ...;
int id = pObj->Id;
Abc_Ntk_t *pNew = Abc_NtkDup(
                    pObj->pNtk);
pObjNew = Abc_NtkObj(pNew, id);
assert(pObjNew == pObj->pCopy)
```

Faking a persistent interface means that we need to copy a data structure whenever we would like to mutate it some more after exiting the monad. In this case, you have to very careful about what invariants you assume about the copying process. Early in the project, I assumed that the IDs of nodes was stable across a copy: this was not actually the case! The correct thing to do was to use the pCopy field to get the new version of the node.

Type system to the rescue

```
branch :: (∀n2. NT (Dup n n2) ()) → NQ n AIG
```

$$\forall n, n2.\ n \neq \text{Dup } n\ n2$$

```
translate :: Node n → NT (Dup n n2) (Node (Dup n n2))

n ← getInput 0
alt ← branch $ do
    n' ← translate n
    ...
```

Since our in-Haskell representation of nodes was just an integer ID, with no pointer to the owner network, users need to explicitly translate an ID from an old object to a new object. We can enforce this in the type system by assigning a new phantom type variable that depends on the old network's phantom type (using Dup.) Our translation function then only permits translating between networks whose phantom types line up in this manner.

static inline

```
static inline int Abc_NtkObjNum
```

```
abcNtkObjNum = liftM cIntConv.
                    {#get Abc_Ntk_t->nObjs#}
```

OR

```
int OurAbc_NtkObjNum(Abc_Ntk_t* p) {
        return Abc_NtkObjNum(p); }
```

Haskell's emphasis on static checking means that if you subvert the static type checker, it's very easy to cause a segfault. This includes writing low-level code that dereferences pointers. However, one instance where you might be tempted to write C-style code in Haskell is when handling static inline functions, which cannot be imported via the FFI without creating a wrapper function around it, which means that it can't be inlined. The temptation is high...

Don't reimplement functions...

...even if they're shorter

bindings-dsl had the Right™ idea

Bugs with No Stack Trace

```
(gdb) bt
#0 0x0804de37 in s26i_info ()
#1 0x00000000 in ?? ()
```

...but I've concluded that it's not worth it! Inline functions can quickly reach a complexity that make comparison with a Haskell transcription non-trivial, and when you get it wrong, you get segfaults with no stack trace (if FFI code segfaults, you do get a stack trace of all the C code.) bindings-dsl had the right idea: add convenience macros for making the process of wrapping functions less painful.

Interruptibility

```
$ ./signal-test
Installing our signal handler...
Checking for equivalence...
^C^C^C
^Z
$pkill signal-test
Terminated
```

The last thing to say is interruptibility. Pure Haskell code is very interruptible. FFI code is very *not* interruptible, due to limitations in the RTS and also due to the fact that most C code is not designed to be interrupted. So if you installed your own signal handler, you should uninstall it when you call a function backed by the FFI, even if it has been made pure by the virtue of an unsafePerformIO.

Questions?

Questions?


Thanks!

Thanks for listening!