SMT

Combining languages and SMT solvers

an EDSL study

Don Stewart | WG 2.8 | March 2011

galois

The Productivity Challenge in Software

- As John Hughes pointed out yesterday, we must climb a productivity cliff in software construction if we are to build the multi-million line codebases industry demands, when labor is expensive
- The "FP" approach:
  - Automate code generation: Embedded DSLs
  - Automate verification and validation: automated testing + solvers + provers
- This talk:
  - Making high powered SMT solvers easier to program by mortal programmers (i.e. without an FM background)
  - How? Program them with an EDSL of course!

Background: SAT solvers

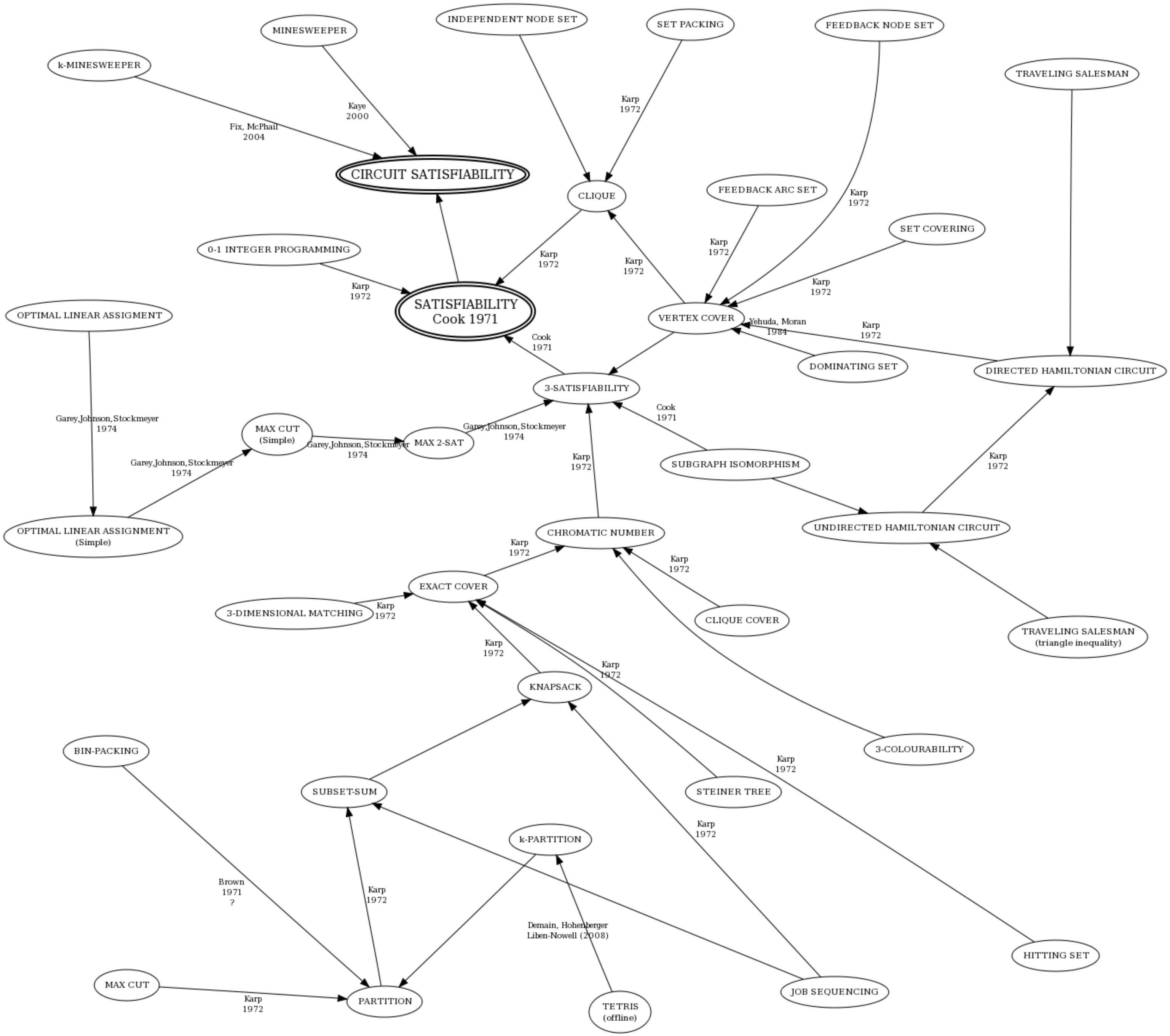
- SAT: finding if a formula has a model
  - Find an assignment to Boolean variables to make proposition true
  - The classic NP-complete problem
  - And now routinely fast in practice
- 10^300 states are commonly handled these days
- Can handle booleans, vectors of booleans, and encodings to bounded types
- Reals, recursive data types ... too hard
- Simple interface (actually returns a Bool, but you then ask for the evidence...):
  - `sat :: Proposition → Maybe Model`

SMT solvers: SAT for programmers

- Booleans are (not so) great, but...
- SMT solvers add pluggable theories to SAT
  - Equality
  - Linear arithmetic, real arithmetic, ...
  - Extensional (applicative) arrays
  - Sized bit vectors
  - Data types (sums, products, record types, recursive types)
  - Lambdas...
- Use clever theory combination theories (combining two decidable theories usually yields a decidable theory)
- Solve for types of variables that are useful to programmers

SMT Solvers: Big Hammers for Solving

- Declarative programming:
  - Describe a problem as a SAT problem, ask the solver for solutions
  - Massively efficient search in your programming language?
  - Hover on the limits of decidability
- Make NP-complete problems look like nails
  - Verifying large sets of constraints and pre/post-conditions satisfy a property – e.g. demo by Jean-Christophe on Tuesday
  - Generating test cases from models
  - Solving scheduling problems – Eaton's garbage truck controllers, Galois' flight hardware monitors
  - Equivalence checking: Cryptol's VHDL ↔ src checker
  - Easier things: Loco game solvers? Checking business compliance rules


MINESWEEPER
INDEPENDENT NODE SET
SET PACKING
FEEDBACK NODE SET
k-MINESWEEPER
TRAVELING SALESMAN
Kaye
2000
Karp
1972
Fix, McPhail
2004
CIRCUIT SATISFIABILITY
CLIQUE
FEEDBACK ARC SET
Karp
1972
SET COVERING
0-1 INTEGER PROGRAMMING
Karp
1972
Karp
1972
Karp
1972
Karp
1972
SATISFIABILITY
Cook 1971
VERTEX COVER
Karp
1972
OPTIMAL LINEAR ASSIGMENT
Yehuda, Moran
1984
Karp
1972
Cook
1971
DOMINATING SET
DIRECTED HAMILTONIAN CIRCUIT
3-SATISFIABILITY
Cook
1971
Garey,Johnson,Stockmeyer
1974
MAX CUT
(Simple)
Garey,Johnson,Stockmeyer
1974
MAX 2-SAT
Garey,Johnson,Stockmeyer
1974
Karp
1972
SUBGRAPH ISOMORPHISM
Karp
1972
Garey,Johnson,Stockmeyer
1974
OPTIMAL LINEAR ASSIGNMENT
(Simple)
CHROMATIC NUMBER
UNDIRECTED HAMILTONIAN CIRCUIT
Karp
1972
Karp
1972
EXACT COVER
3-DIMENSIONAL MATCHING
Karp
1972
CLIQUE COVER
TRAVELING SALESMAN
(triangle inequality)
Karp
1972
Karp
1972
KNAPSACK
BIN-PACKING
Karp
1972
3-COLOURABILITY
SUBSET-SUM
STEINER TREE
k-PARTITION
Karp
1972
Brown
1971
?
Karp
1972
Demain, Hohenberger
Liben-Nowell (2008)
HITTING SET
MAX CUT
JOB SEQUENCING
Karp
1972
PARTITION
TETRIS
(offline)

SMT solvers as a programming paradigm

- Not programming in a functional style anymore:
  - Take a problem you want to solve
  - Identify a set of variables that could represent a solution
  - Write down all the constraints on those variables that a solution must satisfy
  - Add any other facts you know about
  - Blast it with the solver.
- Feels like a meta-programming game (recursing over a structure of a problem AST, yielding constraints)

Aside: Insights into a healthy technology community

|galois|

- Yearly shootouts (SMT-COMP) as part of conferences
- Results published, prizes awarded
- Shared, large suite of benchmarks
- Many different problem divisions – based on theory type – so easy to concentrate on innovation in one area at a time
- A common input language – SMT-LIB – with solvers also accepting their own custom languages

  "SMT-LIB currently contains 93,480 benchmarks (totalling 16.2 GB) in 325 families over 22 logics"

- *Thought: Modify the ICFP contest to directly improve the state of the FP ecosystem...*
- *Thought 2: Use Fritz's regex to enumerate solver ASTs for those guys...*

So... how do we get that power into our programming languages?

- SMT-LIB input source (at least, Yices version):

```
(define p::bool)
(define q::bool)
(define r::bool)
(assert (=> (and (=> p q) (=> q r)) (=> p r)))
```

- Programmers don't want to switch tools, so write in an existing notation they understand: their language:

```
\p q r -> (p → q && q → r) → (p → r)
```

Goal: make the later notation work. <quick demo>

Challenges for the embedding: funky SMT type systems

- As Jean-Christophe mentioned on Monday, SMT solvers seem to have many-sorted (ad hoc-ish) type systems
- Some dependently typed features, known holes and unsoundness – ill-typed programs accepted by type checkers, and fail with runtime assertions or segfaults
- Need to embed a model of the solver's type system accurately into a sound logic (host language's type system)

EDSLs SMT Design Goals

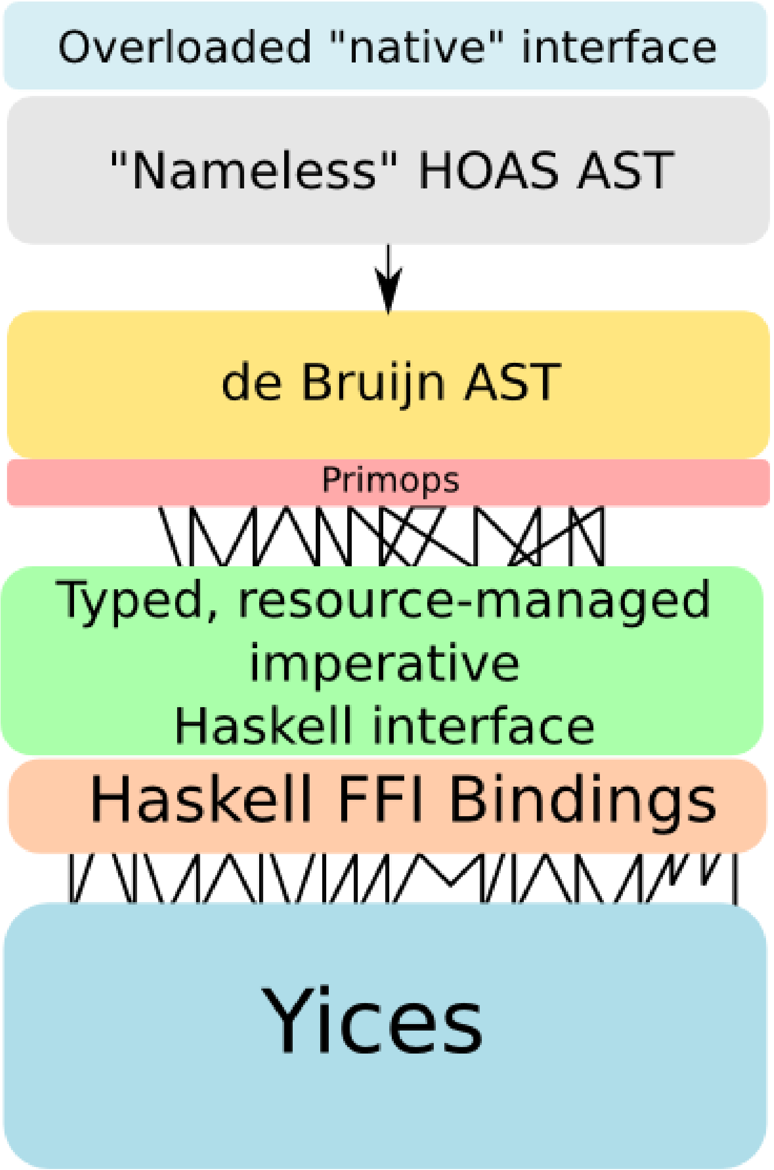
- Type inference
- Native language libraries, functions and types
- Seamless interoperability with the host language
- Clean encoding of solution variables
- Efficient, safe (don't trust SMT type checker, if it has one)
- Extraction of models into usable form
- Reduce amount of functions exposed by solver, by an order of magnitude
- Strategy: a typed, polymorphic, HOAS-*style* EDSL with top-level lambda-bound variables to represent holes, and type-safe optimization layer

Implementation

Decide on how to interact with a solver

- FFI bindings vs scripted processes
- Standard SMT-LIB format (less expressive) vs custom solver languages accepting extensions
- No standard way to extract results from solvers:
  - Get a model
  - Find counter-examples
  - Generate multiple solutions
  - Marshalling function types...
- First cut: one solver (Yices), with FFI bindings
- Next cut: many solvers, language AST input
- But one simple surface EDSL

Tool architecture



Layer 1: bindings

Take the 160 functions and dozen types defined in the Yices C interface, and bind them to Haskell via the FFI:

```
data YContext

foreign import ccall unsafe "yices_mk_true"
    c_yices_mk_true :: Ptr YContext -> IO (Ptr YExpr)

foreign import ccall unsafe "yices_mk_bool_var"
    c_yices_mk_bool_var :: Ptr YContext -> CString -> IO (Ptr YExpr)

foreign import ccall unsafe "yices_mk_and"
    c_yices_mk_and :: Ptr YContext -> Ptr (Ptr YExpr) -> CUInt -> IO
   (Ptr Yexpr)

foreign import ccall unsafe "yices_assert"
    c_yices_assert :: Ptr YContext -> Ptr YExpr -> IO ()

foreign import ccall unsafe "yices_check"
    c_yices_check :: Ptr YContext -> IO YBool
```

Results of Layer 1

- We can now build ASTs in Yices, assert expressions, and solve them
- Downsides: its a fine, unsafe imperative language
  - Exactly the same as programming in C
  - Fully imperative, no resource safety
  - More type safe than C, but only just...
- So next up, retain the imperative layer, but use Haskell types and add resource safety

Layer 2: native types + resource safety

```
data Context = Context { yContext :: ForeignPtr YContext
                       , yDepth   :: !(MVar Integer) }

mkContext :: IO Context
mkContext = do
    ptr <- c_yices_mk_context
    fp  <- F.newForeignPtr ptr (c_yices_del_context ptr)
    n   <- newMVar 0
    return $! Context fp n

assert :: Context -> Expr -> IO ()
assert c e = withForeignPtr (yContext c) $ \cptr ->
    c_yices_assert cptr (unExpr e)
```

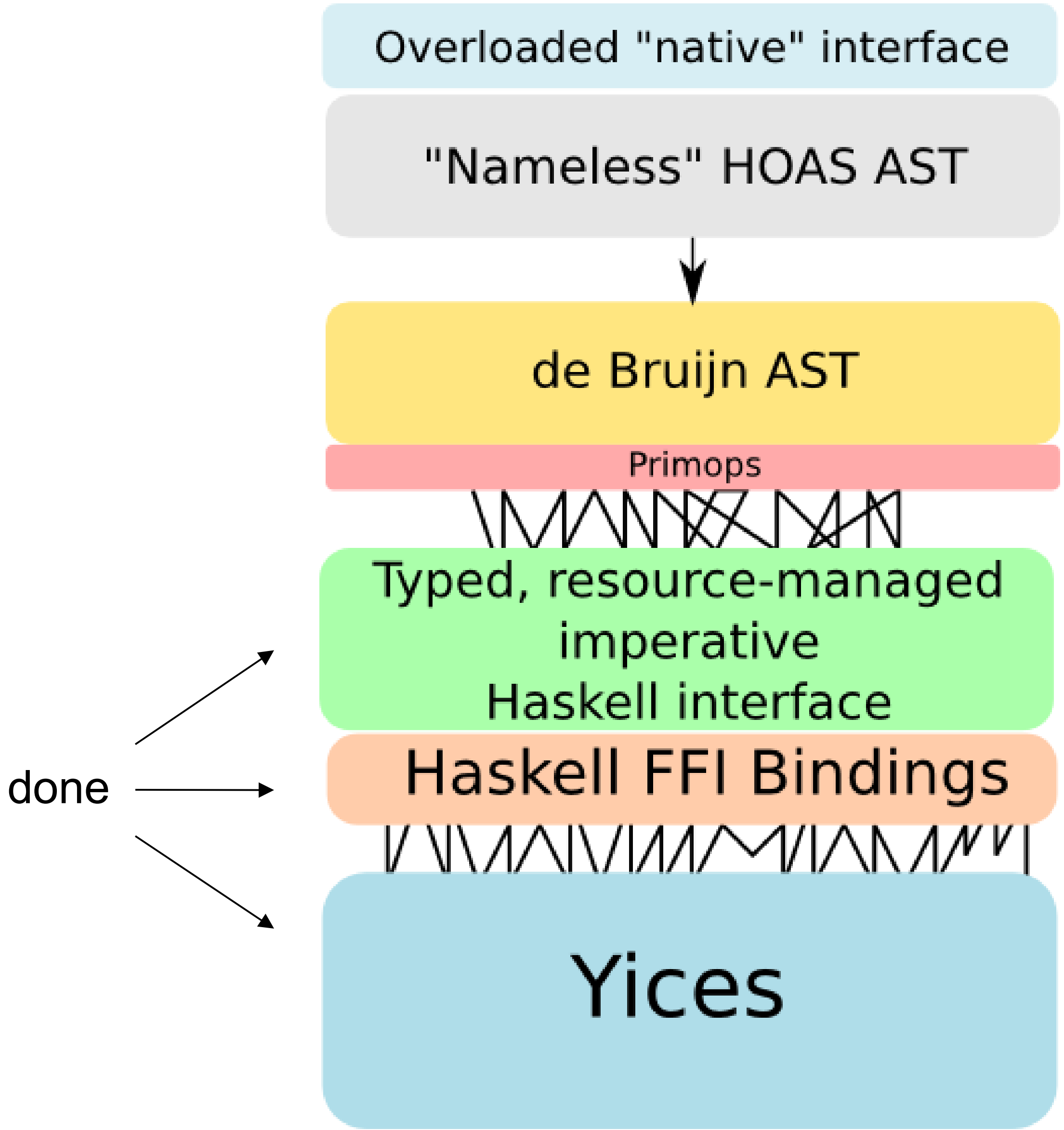
Layer 2: native types + resource safety

```
newtype Expr = Expr { unExpr :: Ptr YExpr }

mkTrue :: Context -> IO Expr
mkTrue c = withForeignPtr (yContext c) $ \cptr ->
    Expr <$> c_yices_mk_true cptr

mkBool :: Context -> String -> IO Expr
mkBool c n =
    withCString n $ \cstr ->
    withForeignPtr (yContext c) $ \cptr ->
        Expr <$> c_yices_mk_bool_var cptr cstr

mkAnd :: Context -> [Expr] -> IO Expr
mkIte :: Context -> Expr -> Expr -> Expr -> IO Expr
```

Note: throw away the iterator API

```
getDecls :: Context -> IO [Decl]
getDecls c = do
    i <- newIterator c
    go i
  where
    go i = unsafeInterleaveIO $ do
                b <- iteratorHasNext i
                if b then do
                        d  <- iteratorNext i
                        ds <- go i
                        return (d:ds)
                     else
                        return []
```

Story so far...



Overloaded "native" interface
"Nameless" HOAS AST
de Bruijn AST
Primops
Typed, resource-managed imperative Haskell interface
Haskell FFI Bindings
Yices
done

Next : Step into the effect-free world

- Build an AST representing the proposition expression
- Interpret or compile it, to yield its effects on the SMT solver
- Simple API:
  - data Exp t
  - solve :: Exp t → Result
- Not quite so simple:
  - Free variables in proposition mapped to lambda-bound parameterers to AST
  - Bounded polymorphic functions, need EDSL-level type classes
  - Should be HOAS
  - Type level naturals for bit vector operations
- Design customized version of Chakravarty's CUDA “accelerate” EDSL (both layers)

Primops GADT for an SMT solver

```
data PrimFun sig where
    PrimLt     :: ScalarType a -> PrimFun ((a, a) -> Bool)
    PrimGt     :: ScalarType a -> PrimFun ((a, a) -> Bool)

    PrimAdd    :: NumType a -> PrimFun ((a, a) -> a)
    PrimMul    :: NumType a -> PrimFun ((a, a) -> a)

    PrimLOr    :: PrimFun ((Bool, Bool) -> Bool)
    PrimLNot   :: PrimFun (Bool -> Bool)

    PrimBVXor :: PrimFun ((BitVector,BitVector) -> BitVector)
    PrimBVNot :: PrimFun (BitVector -> BitVector)
    PrimBVSL0 :: PrimFun ((BitVector,Int) -> BitVector)
```

Layer 3: an expression AST

Glue together PrimFuns in interesting ways:

- Application of PrimFuns represented in saturated form
- No lambdas (undecidable)
- Variables represented by typed de Bruijn index into an environment
  - At this layer they're (typed) numbers, they'll be free variables on the surface layer
- Only scalar literals allowed (for now)

- Explicit variables with tags makes code generation easier

Nameless (de Bruijn) AST

```
data OpenExp env t where
  Var           :: IsAType t
                => Idx env t
                -> OpenExp env t

  OConst        :: (IsScalar t)
                => t
                -> OpenExp env t

  OPrimApp       :: PrimFun (a -> r)
                 -> OpenExp env a
                 -> OpenExp env r
```

What about bindings?

- SMT solver expressions have N free variables, representing variables for the solver to search on
- Represented in HOAS as a function:
  - \p q r → … stuff with p q r ….
- So SMT programs have variadic type, depending on the number of free variables
- Should be no truly "free" variables in body of expression

- So represent expression language as two layers: body and binders

What about bindings?

```
data OpenProg t where
    OP :: OFun t -> OpenProg t

type OFun t = OpenFun () t

data OpenFun env t where
  OBody :: OpenExp env t
        -> OpenFun env t

  OLam  :: IsAType a
        => OpenFun (env, a) t
        -> OpenFun env (a -> t)
```

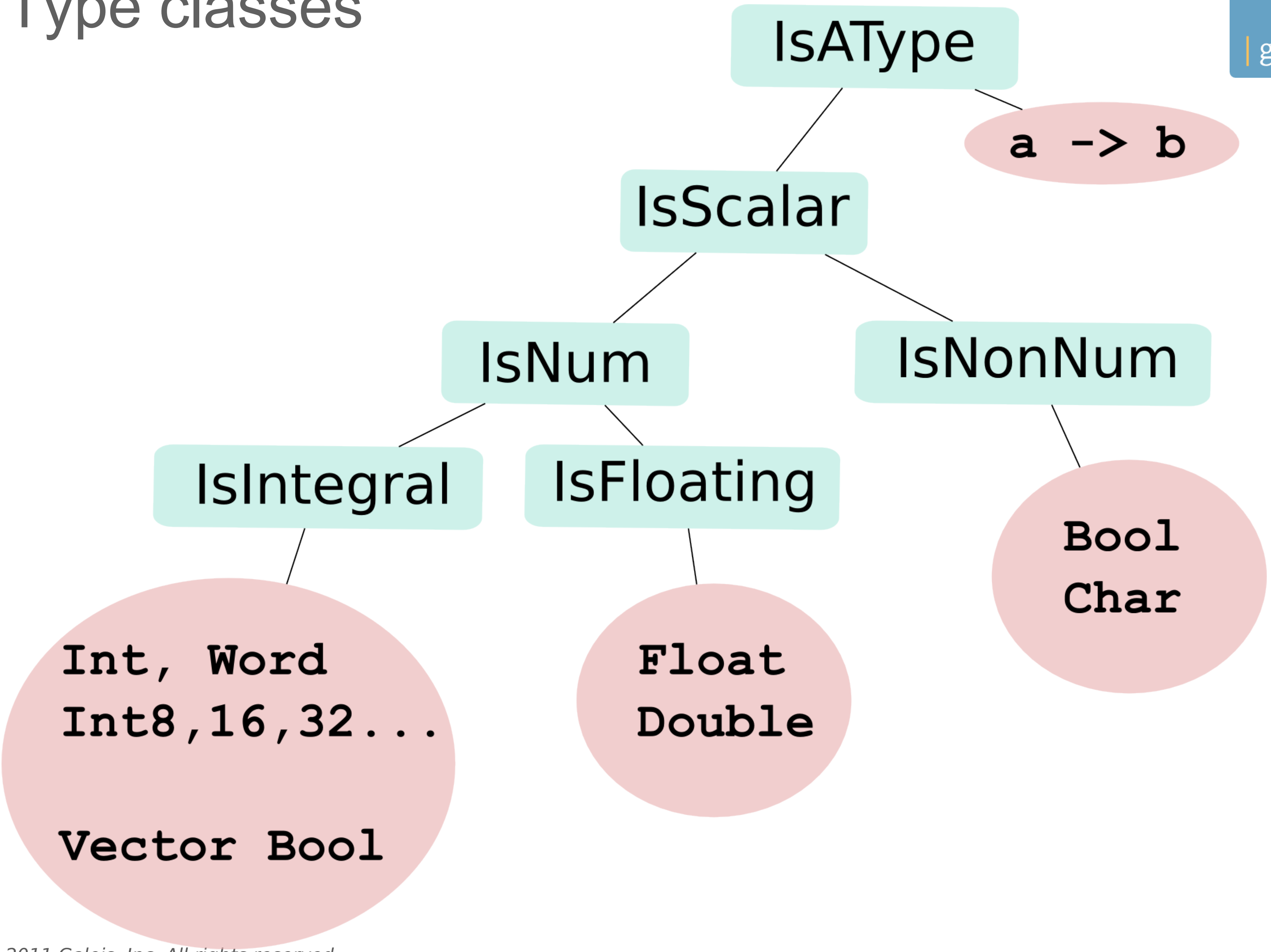
Notes: Type-decorated AST

Want (bounded) polymorphic functions: (+), (*), shiftL, xor

Again, stealing from Chakravarty's accelerate EDSL,

- Decorate AST with type information
- And use new type classes hierarchy for EDSL types
- Reflects dictionaries into data, so we can pattern match on them

Type classes



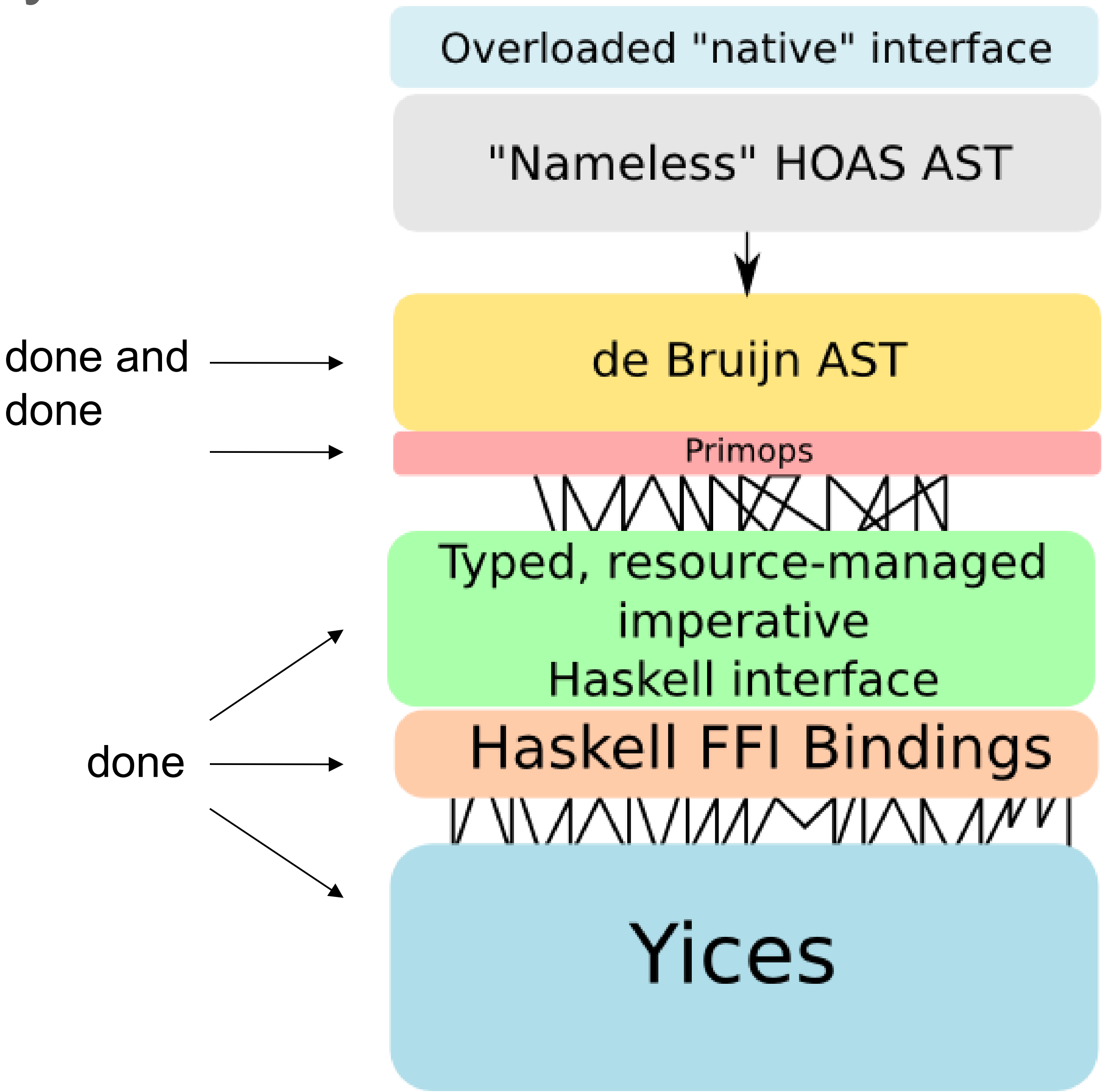
Type class reflection

```
data NonNumDict a where
  NonNumDict :: (Eq a, Ord a, Show a)
             => NonNumDict a

data IntegralDict a where
  IntegralDict :: ( Bounded a, Enum a, Eq a, Ord a, Show a
                  , Bits a, Integral a, Num a, Real a)
               => IntegralDict a

data IntegralType a where
  TypeInt     :: IntegralDict Int     -> IntegralType Int
  TypeInt8    :: IntegralDict Int8    -> IntegralType Int8
```

Type class reflection: keep dictionaries around

```
class (IsScalar a, IsNum a) => IsIntegral a where
  integralType :: IntegralType a

instance IsIntegral Int where
  integralType = TypeInt IntegralDict

instance IsIntegral BitVector where
  integralType = TypeVectorBool IntegralDict

instance IsIntegral Int8 where
  integralType = TypeInt8 IntegralDict
```

Story so far...



Overloaded "native" interface
"Nameless" HOAS AST
done and done
de Bruijn AST
Primops
Typed, resource-managed imperative Haskell interface
done
Haskell FFI Bindings
Yices

But explicit variable binding are annoying

|galois|

Want to just use host language's binding forms (let, lambda)

And not worry about substitution

However, still need the tagged variable representation for easier manipulation

So translate from a HOAS-style represention into the de Bruijn form

- Chakravarty. "Converting a HOAS term GADT into a de Bruijn term GADT" 2009
- Atkey, Lindley, and Yallop. "Unembedding domain-specific languages." 2009

HOAS representation: expressions
Hides the environment

```
data Exp t where
    Tag         :: IsAType t
                => Int -- binding site count
                -> Exp t

    Const       :: IsScalar t
                => t
                -> Exp t

    PrimApp     :: PrimFun (a -> r)
                -> Exp a
                -> Exp r
```

HOAS representation: outermost binders

```
data Prog r where
     P :: Prog f r => f → Prog r
```

SMT solver programs are variadic – so use recursive instances to convert from HOAS style to de Bruijn form, one bind at a time...

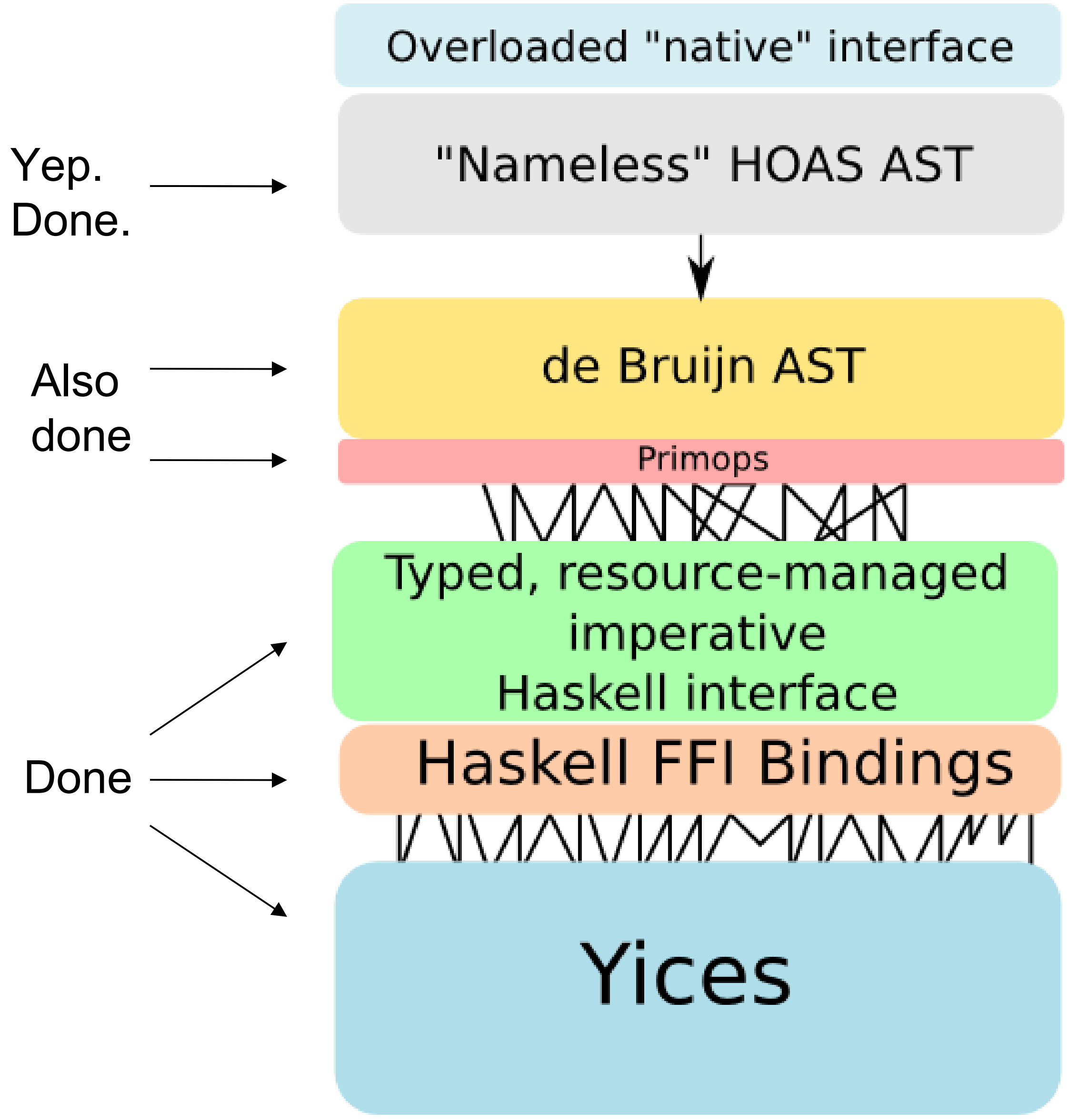
(Same as Text.Printf variadic type trick)

```
convertAll :: Prog r -> OpenProg r
convertAll (P f) = OP $ convert EmptyLayout f
```

Converting HOAS to de Bruijn environment

```
class Prog f r | f -> r where
  convert :: Layout env env -> f -> OpenFun env r

instance Prog (Exp b) b where
  convert lyt e = OBody (convertOpenExp lyt e)

instance (IsAType a, Prog f r)
     => Yices (Exp a -> f) (a -> r) where
  convert lyt f = OLam (convert lyt' (f a))
    where
      a    = Tag (size lyt)
      lyt' = inc lyt `PushLayout` ZeroIdx
```

Story so far...



Overloaded "native" interface
"Nameless" HOAS AST
de Bruijn AST
Primops
Typed, resource-managed imperative Haskell interface
Haskell FFI Bindings
Yices
Yep. Done.
Also done
Done

Home stretch: smart constructors

```
mkAdd :: IsNum t => Exp t -> Exp t -> Exp t
mkAdd x y = PrimAdd numType `PrimApp` tup2 (x, y)

mkMul :: IsNum t => Exp t -> Exp t -> Exp t
mkMul x y = PrimMul numType `PrimApp` tup2 (x, y)

instance (IsNum t) => Num (Exp t) where
  (+)        = mkAdd -- overloaded, bitvectors in IsNum
  (-)        = mkSub
  (*)        = mkMul
  negate x   = 0 - x
```

Home stretch: bit vectors

```
mkBVAnd :: Exp BitVector -> Exp BitVector -> Exp BitVector
mkBVAnd x y = PrimBVAnd `PrimApp` tup2 (x, y)

mkBVOr  :: Exp BitVector -> Exp BitVector -> Exp BitVector
mkBVOr x y = PrimBVOr `PrimApp` tup2 (x, y)

instance Bits (Exp BitVector) where
    (.&.)      = mkBVAnd
    (.|.)      = mkBVOr
    xor        = mkBVXor
    complement = mkBVNot
```

Challenge : bit vector operations

Most bit vector operations care deeply about the size of the vector coming in.

Need to statically enforce constraints:

- Bit vector: (+), (-), (*), (<), (&&) etc.

Must be bitvector expressions of same size.

More interesting types: extracting sub-vectors

"/a/ must a bitvector expression of size /n/ with begin < end < n.

The result is the subvector slice a[begin .. end]."

Home stretch: sigh: Eq, Ord, Bool

```
infix 4 ==*, /=*,  <*, <=*, >*, >=*

(==*) :: (IsScalar t) => Exp t -> Exp t -> Exp Bool
(==*) = mkEq

(<*) :: (IsScalar t) => Exp t -> Exp t -> Exp Bool
(<*)  = mkLt

infix 0 ?
(?) :: Exp Bool -> (Exp t, Exp t) -> Exp t
c ? (t, e) = Cond c t e
```

Haskell's not quite the perfect EDSL host

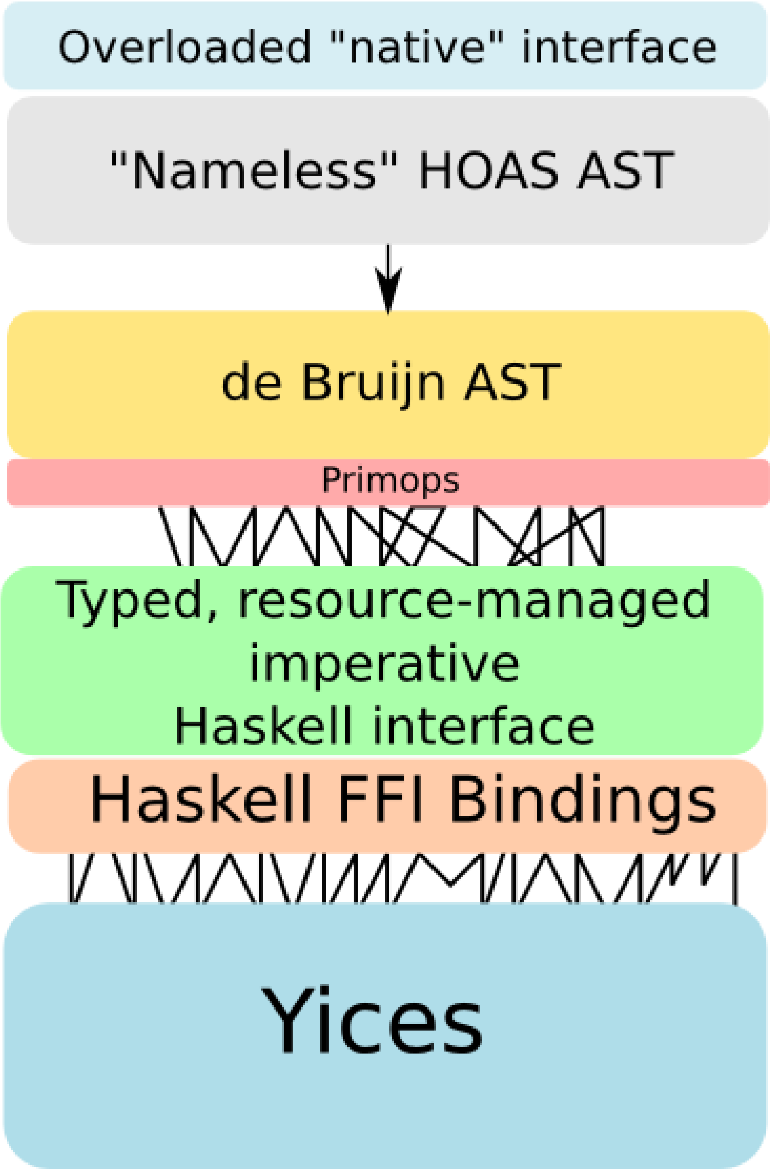
Programs interpreted or compiled

Resolve overloading when calling the solver

```
exec c (OPrimApp  (PrimMul (IntegralNumType (TypeVectorBool _)))
          (OTuple (NilTup `SnocTup` x1 `SnocTup` x2))) = do
            e1 <- exec c x1
            e2 <- exec c x2
            Yices.mkBVMul c e1 e2


exec c (OPrimApp  (PrimMul _)
          (OTuple (NilTup `SnocTup` x1 `SnocTup` x2))) = do
            e1 <- exec c x1
            e2 <- exec c x2
            Yices.mkMul c [e1,e2]
```

TODO: compile to SMT-LIB format

That's the full stack



Much simpler API to the solver now

EDSL makes the interface dramatically simpler.

- 160 functions exposed as 1 Exp type and 1 “solve” method.
- Everything else reuses existing language types and instances (functions)!
- Huge reduction in cognitive load
- Well-typed solver programs are well-typed in Haskell too
    - Need -XTypeNats for bitvectors though
- Bounded polymorphism in the EDSL methods reduced the interface size a lot
- Still can't get there with Eq/Ord/Bool though :(

So I can specify and solve now

```
> let prop = \p q r -> (p --> q) &&* (q --> r) --> (p --> r)

> :t prop
prop :: Exp Bool -> Exp Bool -> Exp Bool -> Exp Bool
\x2 x1 x0 -> (||*) (not ((&&*) ((||*) (not x2, x1),
                                (||*) (not x1, x0))),
                        (||*) (not x2, x0))

> solve prop
x0 => False
x1 => True
x2 => False
Satisfiable
```

Or do things with bit vectors

```
> solve $ \b1 b2 →  b1 + 1 ==* b2
            &&* b2 /=* b2 `xor` 7 + ((1 + b1) :: Exp BitVector)
```

```
\x1 x0 -> (&&*) ((==*) ((+) (x1, 0b1), x0),
                (/=*) (x0, (+) (xor (x0, 0b111), (+) (0b1, x1))))
```

```
x0 => 0b101
x1 => 0b100
Satisfiable
```

Or do puzzles

```
latin :: Array (Int,Int) (Exp Int) → Exp Bool
latin env =
      and [ v >=* 1 &&* v <=* n | v ← vars env ]
   &&*
      and [ env ! a /=* env ! (i0,j)
          | a@(i0,j0) <- cells
          , j <- [j0+1 .. n-1] ]
   &&*
      and [ env ! a /=* env ! (i,j0)
          | a@(i0,j0) <- cells
          , i <- [i0+1 .. n-1] ]
```

Still to do...

Generating SMT-LIB output and talking over pipes to other solvers

Recovering sharing! (Essential for industrial-scale solvers)

Support deriving solver embedding for in-language data types

Integrate Diatchki's type level naturals support to give types to bitvector operations with sizes

A design for a scripting monad for interacting with the solver