HaskSAT: an Embedded DSL for SMT Queries

Nirav Dave

Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
ndave@csail.mit.edu

Abstract

As SAT and SMT solvers have improved, it has become common to reduce difficult search problems to satisfiability queries in an appropriate logic. While expressing the queries is often straightforward at the high-level of the problem domain, converting to low-level logical formulae can be tedious, difficult, time consuming, and error prone. In addition, there are many data types, like bounded queues and trees, that are common enough to warrant distributing the conversion from data type to logical formula as a library so that they can be reused by others. Toward addressing these issues we introduce HaskSAT, a Domain-Specific Embedded Language (DSEL) in Haskell leveraging both Generalized Abstract Data Types and Template Haskell to ease the construction of type safe queries. HaskSAT gives a *prescribed method* for converting high-level data types to logical formulae and the results back to those same data types. This framework not allows us to define the conversion compositionally, it also makes the queries *portable*. Data types are extensible in the sense that their conversion to logical formulae can be built one upon another, and even *polymorphically*. Additionally, we can define functions on convertible data types for use in queries in a natural way.

1. Introduction

An effective approach to reasoning about complex dynamic processes, such as program execution, is to represent the process symbolically as a formula in the logic of an efficient satisfiability solver. The solver can then be called to prove properties, *e.g.,* bounded safety properties, or to resolve concrete values exhibiting an important device behavior, *e.g.,* a bug. As these SAT (satisfiability, for propositional logic) and SMT (satifiability modulo theories, for a wide set of richer logics) solvers are highly optimized, this approach is often more efficient than any specialized solver which can be generated with reasonable effort. This reduction to SAT/SMT approach has been used successfully in many contexts like program verification, testing, and bug finding (*e.g.,* [1–3]).

While leveraging such solvers is conceptually straightforward, there is a significant amount of work to translate from the original problem to an efficiently understandable result. A key concern in this is expressing high-level concerns in the representation. Representations which concisely express the high-level sharing or expose common patterns can result in orders of magnitude speed up.

In addition to being of high importance the translation itself can be a lot of work. SAT solvers reason about propositional formulae and most require the input in conjunctive normal form. Translating, for instance, a snippet of C code to its individual bits as propositional variables is substantial change of representation. This issue is somewhat mitigated by SMT solvers (*e.g.,* [4–7]) which provide richer data types (adding more theories). For instance, STP [5], the main SMT solver we use as the backend for HaskSAT provides bit vectors and bounded arrays as data types, along with a number of arithmetic and bitwise operations on bit-vectors.

It is natural to think that the best approach to this is to select an SMT solver which has direct representations for as many the types that one needs to represent their query and to translate the unrepresentable types into representable expressions. The issue with this approach is that it locks us into a particular set of theories as our representation. If a better solver comes along and it does not accept necessary theories we have to redo our translation. Even if it does, if the relative efficiencies of theories change, exploiting this requires how we represent our data types (*e.g.,*we can represent a bounded FIFO naturally using an array with a head and tail pointer or as two unbounded FIFOs, one representing values, the other empty spaces).

To realize the full benefit of these solvers, users must be able to *easily* adapt and integrate them into their system. This means we must be able to pose questions from a source domain in a natural way and have it translate automatically into the backend query domain. This translation must maintain sufficient flexibility that retargeting is not onerous.

A natural solution for this to define the query language as an embedded language in a general purpose language. This gives us the not only the standard benefits of sharing much of the parsing and typing infrastructure but also the foreign function interface which is key for such a tool.

In this paper, we present HaskSAT our DSL embedded in Haskell aimed at expressing queries to SMT solvers. HaskSATallows the introduction of user-defined data types and functions. Using HaskSAT one can represent SAT/SMT queries using Haskell expressions (leveraging both Haskell's powerful pattern-matching syntax and function declaration capabilities) and have the query have the query automatically translated to a input for the backend solver, capture the output and translate the result back to original Haskell-level data types. A principle of HaskSAT is that adding data types should be straightforward, modular, and easily sharable.

Specifically, HaskSAT:

- Allows end users to write queries using their own data types and functions. This makes queries more compact and natural.
- Provides a clean higher-level abstraction for queries which SMT developers can leverage
- Provides a way to easily share reduction machinery (expressed at the type-level) between distinct projects.
- Is a Haskell DSEL making interfacing to Haskell-based tools easy. Haskell's FFI gives a natural vector for other systems.

2. Background: SAT and SMT Queries

To better understand the task at hand let us consider exactly what a query contains and how one might represent a query and show how raising the abstraction level simplifies both expression and increases efficiency. We also include a brief description of the execution strategies used by SAT and SMT solvers.

2.1 A Base Representation: Propositional Satisfiability (SAT)

SAT sovlers determine the satisfiability/unsatisfiability of propositional formula. Loosely speaking, a formula φ is satisfiable when there exists a substitution $\rho : \text{vars}(\varphi) \longrightarrow \{0, 1\}$ such that $\rho(\varphi)$ evaluates to true. When a formula is unsatisfiable the solver will simply return this fact, but when a formula is satisfiable not only is that fact reported, most solvers also return a substitution witnessing it. While most SAT solvers require input to be in conjunctive normal form, below we take for granted that the wider set of propositional formulas can be easily translated to CNF (*e.g.,* see [8]).

To give a concrete example of how to solve another problem using SAT, consider the N queens problem, where the goal is to place N queens on an $N \times N$ chessboard such that no queen lies on the same row, column, or diagonal. A very simple way to encode this into SAT is to have N^2 propositional variables ($n_{\{i,j\}}$) which represent whether a queen exist on each square. The formula we want to create will encode the restriction on placements. Specifically we have to check exactly 1 of the N booleans in each row and column is true and that at most 1 square in the diagonal is true.

For example, the row constraint could be encoded as follows:

$$\bigwedge_{1 \le r \le N} \left[\left[\bigwedge_{1 \le c, c' \le N, c \neq c'} \neg(x_{\{r,c\}} \wedge x_{\{r,c'\}}) \right] \wedge \left[\bigvee_{1 \le c \le N} x_{\{r,c\}} \right] \right]$$

The column constraints are symmetric and while slightly more complicated, the diagonal constraints are also straightforward.

SAT solvers are efficient due to a huge amount of research on methods and algorithms, such as non-chronological backtracking and clausal learning, *e.g.,* see [9, 10].

2.2 Adding Types: Satisfiability-Modulo-Theories (SMT)

SMT solvers work in a richer world than pure propositional logic, typically including additional non-logical symbols (*e.g.,* $+, \leq$) as well as types/sorts (*e.g.,* bitvectors, natural numbers). As input an SMT solver accepts a well-formed formula, φ, in whatever language it reasons in. As output, the solver reports whether it is satisfiable or unsatisfiable, and if so, giving a witness. For most SMT solvers, satisfiability means the existence of a substitution $\rho : \text{vars}(\varphi) \longrightarrow M$, where M is the carrier of some canonical model $\mathcal{M}$, such that $\mathcal{M} \models \rho(\varphi)$. To be completely precise, one would need to define exactly what $\rho(\varphi)$ is, but we avoid such a tangent here.

To understand how this affects our queries, consider the previous N-queens problems encoded to work with a bit vector solver. Now that we have bitvectors, we'd again exploit the property that we know exactly one square in each row has a queen. Instead of using a boolean for each square we can represent the position as a mapping from row position to the unique column where a queen lies. Thus we need $N \log_2(N)$ numbers. Unlike the SAT expression we can encode the constraints using properties of bitvectors. Specifically:

- Validity of placement of queens by asserting that the column values lie between 0 and $N - 1$.
- Rows have a unique queen is preserved by construction
- Columns have a unique queen by asserting that each of the $N(N - 1)$ queen pairs are on different pairs
- Keeping the queens of the same diagonal can be done by noting that one set of diagonal can be indexed by the difference in row and column values, and the other by the sum of the row and column values. Thus we can assert the uniqueness of these values as we did for the columns. This requires we increase the representation of bitvectors to prevent overflow.

It is clear that we could have done this encoding and translation in the SAT example instead of the SMT. The major difference (beyond the ease of writing) in the ability for optimization. For instance it's much easier to exploit the associative property of addition and subtraction when the operators are directly represented and not inferred. This means that the assertions that the queens on the second and third row are on different diagonals:

$(2 - c_2 \neq 3 - c_3) \wedge (2 + c_2 \neq 3 + c_3)$

can be rewritten to:

$(c_3 - c_2 \neq 1) \wedge (c_3 - c_2 \neq -1)$

which is simpler and exposes the sharing of the $c_3 - c_2$ expression.

As with SAT solvers, there is a substantial amount of work on algorithms and techniques used in SMT solvers. For logics that are easily reduced to SAT, it is sometimes most efficient to do high-level rewriting followed by conversion to SAT, and possibly some form of abstraction refinement (*e.g.,* [5]). In more complicated contexts, most solvers use methods based on DPLL(T) (*e.g.,* [11]).

2.3 Haskell: Using Lambda functions, Tuples to construct queries

Expressing the N queens problem leveraging bitvectors makes the description significantly easier. However, there is still a great deal of effort to generate the assertions which have a great deal of shared structure. To see how we can make this easier, consider how one might construct a SMT query in a Haskell-like language. First, If we assume the queens are represented as pairs of integers we can represent the assertion that two positions do not conflict as:

```
posValid (r,c) n = (0 <= r && r <= n-1) &&
                   (0 <= c && c <= n-1)
pairValid (r1,c1) (r2,c2) = (r1 /= r2) && (c1 /= c2) &&
       (r1 + c1) /= (r2 + c2) && (r1 - c1) /= (r2 - c2)
```

and the assertions (using the pairs function) given a list of queens positions as:

```
pairs (x:xs) = [(x,y) | y <- xs] ++ pairs xs
pairs []     = []
(all posValid queens) && (all pairValid (pairs queens))
```

This representation is concise and reflect a lot of the high-level information. However, the problem is that we have no obvious way of representing unconstrained variables, on which the whole idea of constraint problems is predicated. To completely represent our query in this form, let us add the concept of a "free" expression which can take any possible value of the appropriate type. This gives us the missing bit and we can generate our list of queen positions.

```
queens = [(i,free) | i <- [0..n-1]]
```

At this point we almost all the pieces to ask a query. The only thing left is how to interpret the result of the query, a point we've ignored in the N queens problem so far. One obvious choice is the row-column positions of the queens. For our example, let's just return the column values as the rows are already known. Now we can write our query monadically as:

```
query :: Int -> Query [Int]
query n = do
  let queens = [(i,free) | i <- [0..n-1]]
  assert (all positionValid queens)
  assert (all pairValid (pairs queens))
  return $ map snd queens
```

Here the assert clauses add the boolean assertions to the constraints and the return types gives us the resulting expression (if one exists). Conceptually our solver will be of type `Query a -> Maybe a`. This query has the benefit of being highly readable as well being parameterized by the board size. This is the inspiration for our syntax for HaskSAT.

3. Expressing Queries

Since we encourage users to use their own types and leverage a large set of Haskell we expect the type structure of a query to be quite rich. Thus, it's important for us to guarantee that some measure of type safety. A standard way of doing this in Haskell DSELs is to leverage Haskell's lets and lambdas and define the combinators to work on Haskell typed representations of our distinct types.

Unfortunately, higher-order sharing is of paramount importance in this translation. For example, exploiting the "functionality" of functions (*e.g.,* that $x = y \rightarrow fx = fy$) can help expose dynamic sharing in the query and thus reduce work for "mostly equivalent" values. This necessarily requires us to be able to "see" the sharing of higher-order objects. As a result, we cannot directly leverage both Haskell's lets and lambdas and need to build our own abstract syntax data type.

Despite not being able to directly leverage Haskell's sharing and lambda abstraction mechanisms, we'd still like to get the same type guarantees which they enforce. We do this by representing our queries in the following Generalized Abstract Data Type (GADT).

```
data (Expr n a) where
 ETerm:: (RepValue n)                          -> Expr n a
 EPrim:: Id                                    -> Expr n a
 EVar :: Id                                    -> Expr n a
 EApp :: Expr n (a -> b) -> Expr n a           -> Expr n b
 EIf  :: Expr n Bool -> Expr n a -> Expr n a -> Expr n a
 ELam :: (Expr n a -> Expr n b)          -> Expr n (a -> b)
 ELet :: Id ->        Expr n a -> Expr n b   -> Expr n b
 EWhen:: Expr n a -> Expr n Bool             -> Expr n a
 EFree::                                        Expr n a
(<@>) x y = EApp x y
```

Here function application (`EApp` or `<@>` is assured to be type safe as we insist by insisting that the applied functions match and return the correct return value. Similarly by representing lambda abstractions (`ELam`) in as Higher-Order Abstract Syntax (HOAS) we guarantee that variables uses in the function body have the correct type. What is not guaranteed is that variables represented by the `EVar` construct have the same type as the associated expression, defined in the `ELet`. This property will need to be guaranteed by hiding the constructors from the user and only allowing joint construction of variables and their bindings.

```
makeLet :: Id -> Expr a -> (Expr a -> Expr b) -> Expr b
makeLet i e f = ELet i e (f (EVar i))
```

For data types and functions to embedded in this type we need a notion of what values that can be represented in the base solver. This requires that the expression type also be parameterized by the backend engine in addition to the corresponding Haskell type. For instance consider the STP SMT solver [5] which supports the theory of booleans, bitvectors, and arrays of bitvectors. We need to have a notion of booleans, bitvectors, and arrays. Thus we can represent a values as the type `Value`:

```
type BitVector = Bitval Int Integer
data Value = BV BitVector      -- bitlength value
           | Array [BitVector] --
```

This done by the type family `Representation n` which defines the representations for both concrete values as well as the backend level types.

```
class Representation n where
  type RepValue n :: *
  type RepType  n :: *
  tyArrow :: Maybe (RepType n) -> Maybe (RepType n)
          -> Maybe (RepType n)
```

This defines both a representation for concrete values as well as types of values in the backend. This notion of types is not necessary but can be helpful when converting to the backend. Using this we can represent symbolic values directly representable in the backend using the `ETerm` constructor. For instance, in our STP backend, we can represent the symbolic boolean expression representing true as:

```
eTrue = (ETerm (BV (BitVal 1 1))) :: (Expr Bool)
```

To introduce the concepts specific to our constraint problems, we need two more constructs. First, `EFree` represents an unconstrained value. This may take any value of the appropriate type.

Second, is the notion of a boolean constraint, introduced by the `EWhen` constructor. Unlike the constraints that we had used previously, these constraints need not appear at the top-level only, but can be sprinkled anywhere in our expression. This allows us to construct higher-order values with partially unconstrained values. (*e.g.,* an Int to Int function which returns the input value plus a non-zero number less than the input).

The expression `EWhen e ew` is valid only when the boolean expression `ew` is true. When the guard is true it behaves exactly like `e`. These When guards translated into boolean assertions in our final SMT query. It is important for when guards to have the non-strictness that we want we must be careful to only assert their truth on the condition of their usage. For example the expression:

```
EIf x (EWhen y p) (EWhen z q)
```

uses the guarded expressions conditionally and so is the same as:

```
EWhen (EIf x y z) (EIf x p q)
```

As you can see the guards `p` and `q` only apply the `x` boolean signifies that we would need to evaluate the associated expression. The handling of these guards is similar to guards in the Bluespec hardware description language [12].

3.1 Translating Haskell Types

One of the most important aspects of simplification is how to deal with data types that do not have a direct analog in the backend solver. If a type is not representable, the only way to deal with it is inline the constructs which cannot be represented and rely the system to remove the node.

To be able to distinguish these we define a type class `ConvType` to tell us what type a symbolic expression has in the backend solver if one exists. If the expression type does not have a direct analog. In principle, we only require a boolean, not a full type representation, but we also insist on the type to serve as a debugging aid in the backend.

```
class (Representation n) => (ConvType n a) where
  getRepType :: (Expr n a) -> Maybe (RepType n)
```

We insist that *all* symbolic expressions have a `ConvType` definition. This requires we change all the GADT constructors to insist on the appropriate context.

3.2 Embedding Haskell Values

Given the Expression Type and our restriction on construction the only place of difficulty is how to deal with the representation of a concrete Haskell value as a symbolic expression.

```
class (ConvType n a, Representation n) => (ConvValue n a)
where
  pack       :: a -> Expr n a
  unpack     :: Expr n a -> Maybe a
  freeExpr   :: Expr n a
```

The `ConvValue` type class is responsible for converting Haskell-types into our internal representation via the functions `pack` and `unpack`. The unpack function is partial in that not all expressions of the appropriate type will have corresponding values (for instance a symbolic variable). At minimum the user is responsible for guaranteeing that at `(unpack(pack x)) = Just x`.

It is possible that we want the codomain of our pack function may be larger than the domain. For instance translating a 3 valued type to a 2-bit value leaves one value meaningless. As a result we cannot directly infer the shape of a free variable and need the user to provide a definition (`freeExpr`) as part of the translation.

3.3 Writing a Query

By using our GADT approach, and structured let constructor we can fairly represent our queries in a relatively straightforward way while maintaining type safety. However, to write something the query represented by this:

```
let x = free :: Maybe Int
    y = free :: Int
    z = case x of
          Nothing -> y
          Just  p -> p + y
in when x (y == z)
```

we have to write something like:

```
query :: (Expr n (Maybe Int))
query =
 makeLet "x" (freeExpr :: Expr n (Maybe Int))$ \x ->
  makeLet "y" (freeExpr :: Expr n Int) $ \y ->
   makeLet "z"
    (EIf (_symbolic_isNothing <@> x) <@> y <@>
     (EIf (_symbolic_isJust <@> x)
       (makeLet "p" (_symbolic_GetJust<@>x) $ \p ->
        _symbolic_plus <@> p <@> y)
     (EWhen freeExpr (pack False)) $ \z ->
      EWhen x (_symbolic_eq y z)
```

Not only did we have to use a less natural let and if notation, but we also needed to desugar the the case representation into lets and ifs, which is significantly more verbose and hard to manage. While annoying, the structure the translation is quite mechanical enough to write a syntax-to-syntax translation to take our description of a expression of type `a`, using the nonexistent functions `free` and `when` to the corresponding symbolic expression of type `Expr a`. We implement such a function in Template Haskell to serve as a function. To do this we need to establish a global naming correspondence between functions and their corresponding symbolic form. The new templated query look like:

```
query :: (Expr n Bool)
query = $(makeSymbolic[|
  let x = free :: Maybe Int
      y = free :: Int
      z = case x of
            Nothing -> y
            Just  p -> p + y
  in x when (y == z) |])
```

This gives us the syntactic benefits of the "ideal" representation, with only a small messiness difficulty in debugging as Template Haskell error messages can be harder to understand than error messages from untemplated code. But If we define correctly typed versions of `when` and `free`, the untemplated version can be type checked with a guarantee that the templated version will also be correct (modulo the existence of needed type class instances).

While it is clear how simple polymorphic code is done, dealing with type contexts is a bit more tricky. For each type class in our input language (*e.g.,* `Eq a`) we need to a corresponding symbolic type class which contains the corresponding symbolic expressions. For instance the symbolic equality type class looks like:

```
class (SymbEq n a) where
  _symbolic_eq  :: (Expr n (a -> a -> Bool))
  _symbolic_neq :: (Expr n (a -> a -> Bool))
```

4. Compilation: Going from Query to Result

Conceptually, the idea of our embedded language is quite straightforward. The user can write queries with relative ease in Template Haskell and they can solve a query using the function:

```
askQuery :: (ConvValue n a) => Query n a) -> IO (Maybe a)
```

which is the previously stated high-level conceptual type in the IO monad to allow external calls.

Internally, the compilation from our Query formulation, to SMT query, to solution, to a result as Haskell type takes a number of steps. There is insufficient space to describe all task in full detail, so we will highlight only the paramount aspects.

4.1 Dealing with Free Expressions

Unlike the lambda calculus, one cannot inline arbitrarily in the context of free expressions. For instance:

```
let x = free in x + x
```

should only allow even results. Inlining `x` causes us to forget this. To deal with this all primitive free expressions should be given a fresh variable name. This is made slightly more complicated by lambda expressions with internal free expressions as each call must refer to a fresh free expression.

4.2 Dealing with Functions

If the backend solver does not directly represent functions, the only interpretation for function is to inline their definition. However, we can exploit the fact that: $x = y \rightarrow fx = fy$
One approach is to treat `f` as an uninterpreted function [13] and abstract away `f`'s properties save for this point. This can drastically simplify the complexity of questions in cases where this is all that is needed, but can result in solutions where the relationship between inputs to `f` and its outputs are not obeyed. A safer, but less abstract way is to encode dynamic sharing. For example, in a system with `v1=f x` and `v2=f x'` to replace v2 with `if x==x' then v1 else v2`.

4.3 Simplifying Expressions

Part of the task of reducing a query to something the SMT solver can understand requires that we be able to simplifications on expression. For instance we should be able to do the following reductions:

```
(\ x -> b) e      = b[e/x]
if True x else y  = x
isJust   (Just x) = True
fromJust (Just x) = x
```

We can easily simplify the first two cases have explicit syntax (`EApp`, `ELam`, and `EIf`) in our expression type and as such we have a deep understanding of the semantic meaning of these expressions and can do these simplifications However, the later example involve primitive functions which are understood only at the user level. If the user defined these functions in terms of functions we understand then we are good, but if not (*e.g.,* they are represented using as `EPrim`) we do not know what to do.

A partial fix for this problem is to use a smart constructor which can apply the optimization if we understand the input. For instance we can write `isJust` as:

```
isJust = ELam $ \ x -> case (unpack x) of
             Just _  -> (pack True)
             _       -> (EPrim "isJust") <@> x
```

However, if we do not know enough about the input at construction we forget this simplification. As we inline (and thus expose more information) during our translation, we are losing opportunities. What we'd like to have is when a simplification fails, for it to be remembered, and then retry later when more information may be available. To do this we change the definition of `EPrim` to store this information:

```
EPrim :: (Expr n a) -> Id -> (Expr n a)
```

Now, the primitive `isJust` refers to itself for construction.

```
isJust = ELam $ \ x -> case unpack x of
             Just _  -> (pack True)
             _       -> (EPrim isJust "isJust") <@> x
```

Our simplification pass can just replace primitive with the original smarter expression and rely on beta reduction retry the simplification.

```
simplify (EApp (ELam f) x) = simplify (f x)
simplify e@(EIf p t e) = case unpack p of
  Just v -> simplify (if v then t else e)
  _      -> EIf (simplify p) (simplify t) (simplify e)
simplify (EPrim sc i)  = sc
simplify ...
```

4.4 Querying the Solver

Having removed all unrepresentable types from the boolean constraints and local variable definitions, we can directly call our SMT solver. As a result all backend-uninterpretable primitive functions have been removed from our query (save for the return expression which we do not pass). We translate each let binding `x = e` into a constraint asserting that the variable `x` is equal to the expression `e`. Each constraint is then conditionally asserted given its context and passed to the solver. The solver gives back a list of variable name, `RepValue n` pairs. which are then inlined back into the return value. Now we have enough information to finish simplifying, simplifying the remaining primitive functions, until we can unpack to a concrete Haskell value.

5. Adding New Types

To add a type to our system such that it can be used, we need to essentially perform the same task that one would do in an SMT solver, *e.g.,* add a new theory. In addition to defining instances of `ConvValue` and `ConvType` for that type, we must also define how functions on those types can be simplified.

To get a better idea of this consider the encoding of a directly representable type (32-bit integers), and indirectly understandable product type (a 2-tuple) and a union type (the Either type) to our bitvector solver STP. These form a basis for all bounded ADTs we might wish to implement.

5.1 The Int Type

The first type we will deal with is the `Int` type which has a direct implementation in our bitvector solver, a 32-bit value. The first task is the straightforward definition of the conversion classes from Haskell types to the concrete results.

```
instance (ConvType STP Int) where
  getRepType x = BitVectorT 32 -- 32-bit bitvector
instance (ConvValue STP Int) where
  pack x  = ETerm (BV (Bitval 32 (toInteger x)))
  unpack (ETerm (BV (Bitval 32 x)))=Just $ fromInteger x
  unpack _                         =Nothing
  freeExpr = EFree --direct notion of unconstrained value
```

Now all that is left is to define the primitive functions on Int. Since the functions have a direct implementation in the backend, one might initially think we can leave the result as abstract primitives. However, we need to do basic constant propagation on these functions so that the strategy of inlining a solution from the backend is guaranteed to simplify. For instance, the addition function is:

```
instance (SymbNum STP Int) where
  _symbolic_plus = ELam $ \ x -> ELam $ \ y ->
    case (unpack x, unpack y) of
      (Just x, Just y) = pack (x + y)
      _ -> (EPrim _symbolic_plus "bvplus") <@> x <@> y
```

5.2 Record Types: Tuple2

The Tuple2 type `(a,b)` is the simplest "record" joiner type. There are three abstract functions which completely cover all functions we can do on tuples, the constructor `tup2` (or (,) as defined in the Haskell prelude), and the first and second projections (`fst` and `snd`). If we have complete definitions of these functions we can transliterate the Haskell definitions of other functions into the symbolic domain.

Abstractly the only axioms we need to understand how to simplify tuple is the definition of the projections, namely:

```
fst (tup2 x y) = x  and  snd (tup2 x y) = y
```

To implement this type we new three new primitives `_symbolic_tup2`, `_symbolic_fst`, and `_symbolic_snd` for each backend.

```
class (SymbTup2 n a b) where
  _symbolic_tup2 :: Expr n (a -> b -> (a,b))
  _symbolic_fst  :: Expr n ((a,b) -> a)
  _symbolic_snd  :: Expr n ((a,b) -> b)
```

We also need instances for our bitvector backend. First we define `ConvType` and `ConvValue`.

```
instance (ConvType STP (a,b) where
  getRepType x = Nothing
instance (ConvValue STP a, ConvValue STP b
         ,SymbTup2 STP a b) => (ConvValue STP (a,b)) where
  pack (a,b) = _symbolic_tup2 <@> (pack x) <@> (pack y)
  unpack (EApp (EApp tup2 x0) y0) | isTup2 tup2 = do
    x <- unpack =<< (cast x0) -- Maybe Monad
    y <- unpack =<< (cast y0)
    return (x,y)
  freeExpr   = _symbolic_tup2 <@> freeExpr <@> freeExpr
```

This done all that is left is to define the three functions. Given our simplification strategy based on smart constructors, we need to exploit the tuple axioms when defining the projections. The constructor can remain abstract.

```
instance (ConvType STP a, ConvType STP b)
                              => (SymbTup2 STP a b) where
  _symbolic_tup2 = EPrim _symbolic_tup2 "tup2"
  _symbolic_fst  = ELam $ \ x -> case x of
    ((EApp (EPrim _ tp) a) b) | tp == "tup2" -> a
    _ -> (EPrim _symbolic_fst "fst") <@> x
  _symbolic_snd  = ELam $ \ x -> case x of
    ((EApp (EPrim _ tp) a) b) | tp == "tup2" -> b
    _ -> (EPrim _symbolic_fst "fst") <@> x
```

Now all that is left is to define the non-primitive functions on tuple (`==`, `<`, etc.) by transliterating the appropriate class instances and functions.

5.3 Union Types: Either

The `Either a b` type can either be a `Left a` value or a `Right b` value. Unlike products, choice cannot directly be implemented in a SAT solver. Instead we need to encode this as product by modelling `Either a b` as the type `(Bool, a, b)` where the boolean signifies if we have a `Left` or a `Right` and therefore which field is meaningful; the unused field is left unconstrained. represent the either type we need to define primitives for construction(Either), tag matching(isLeft and isRight), and projection(getLeft and getRight).

```
instance (ConvType STP (Either a b) where
  getRepType x = Nothing
instance (ConvValue STP a, ConvValue STP b
  ,SymbEither STP a b) => (ConvValue STP (Either a b)) where
  pack (Left a) = _symbolic_Either <@> (pack False)
                                   <@> (pack a) <@> freeExpr
  pack (Right a) = _symbolic_Either <@> (pack True)
                                   <@> freeExpr <@> (pack b)
  unpack (EApp (EApp (EApp f x) y) z) | isEither f =
    case unpack x of
     Just p -> if p then (unpack =<< cast y)
                    else (unpack =<< cast z)
     _      -> Nothing
  freeExpr = _symbolic_Either <@> freeExpr
                                   <@> (pack a) <@> freeExpr
```

Defining the primitives is exactly like the tuple case, save that we have the following axioms:

```
  isLeft   (Left x) = True  isLeft   (Right x) = False
  isRight  (Left x) = False isRight  (Right x) = True
  getLeft  (Left x) = x     getLeft  (Right x) = free
  getRight (Left x) = free  getRight (Right x) = x
```

6. Discussion and Future Work

There has been a fair amount of work in embedding SAT/SMT solvers in Haskell. Many are simple frontends to SMT solvers, the most relevant being Yices [14] which has direct support for lambdas and record types. Satchmo [15] is a project very much in line with our goal. It provides a mechanism of encoding some data types in a way that allows containers and associated functions to be used on "symbolic" expressions and passed into a backend SMT solver. Unfortunately, there is no sugar to deal symbolic union types which makes this approach difficult.

For our purposes, HaskSAT has been successful in expressing the problems which were the inspiration for this work. Perhaps more important, when we showed our tool to others who were considering using SMT solvers for their own work, they immediately saw the value provided here; some even volunteered to help improve HaskSAT themselves to make it more efficient, to port the backend to different SAT and SMT solvers, or even to just define useful library types. This is exactly what we wanted; sharing and interaction between tool users.

In terms of improving the system, there are a few important places where HaskSAT stands to improve. Most obviously is the amount of work required to define add a new type into the system. Assuming we are merely simplifying the type away there is no reason which the definitions cannot be automatically generated from the type definition and/or function definitions.

A small technical problem is an issue with unrolling recursive data types and functions. As we need to completely unroll the symbolic expression to be able to query the solver, this may involve infinite work. One solution is to provide a partial unrolling and iteratively unroll the query until a solution is found r we have a proof that further unrolling will not help. Incremental SMT solvers may provide a partial solution to this by allowing use to pass part of the query into the solver and wait for a partial solution before providing (or removing) more variables and constraints.

The Template Haskell parser is proving to be a hefty but relatively clean approach to getting the full sharing information. A lot of thinking was put forth in making sure that the transformation scheme could handle such things as monadic code (which we do by desugaring to the underlying binds and returns) as well as all of the pattern matching (which are desugared into if conditionals). However, the Haskell parsed expression type is quite complex and the code is quite long. The current parser is about 50% of the total HaskSAT project and many corner cases are still not handled. This task is likely to continue as more syntax additions are tried. One possible improvement would be to provide a "diet" parsed expression type with all of the sugar removed which could be easier to cover and remain fairly stable.

Another concern about the current system is that the optimization engine is relatively naive. There is a large body of work on simplifying rewrites which we could implement despite the backend solver not supporting the optimization or even the type. For instance for an array type, you can rewrite $read(y, upd(arr, x, val))$ as $read(y, arr)$ when $x \neq y$. Currently, when we introduce a new type we do nothing beyond translating it down to our backend solver, because we do not know a priori, any non-trivial optimizations for user-defined types. A natural way to get this information would be to augment HaskSAT, to allow users to provide their own hand-determined rewrites. If we could construct something like GHC's rewrite optimization engine we would cover the majority of optimizations which a user might expect of their SMT solver engine.

References

[1] Ho, P.H., Shiple, T.R., Harer, K., Kukula, J.H., Damiano, R.F., Bertacco, V., Taylor, J., Long, J.: Smart Simulation Using Collaborative Formal and Simulation Engines. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD-00). (2000)

[2] Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI-08). (2008)

[3] Banerjee, A., Barnett, M., Naumann, D.A.: Boogie Meets Regions: A Verification Experience Report. In: Second International Conference on Verified Software: Theories, Tools, Experiments (VSTTE-08). Volume 5295 of Lecture Notes in Computer Science., Springer (2008)

[4] Barrett, C., Tinelli, C.: CVC3. In: 19th International Conference on Computer Aided Verification (CAV-07). Volume 4590 of Lecture Notes in Computer Science. (2007)

[5] Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: 19th International Conference on Computer Aided Verification (CAV-07). Volume 4590. (2007) 519–531

[6] de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-08). Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 337–340

[7] Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-09), Springer (2009)

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edition. MIT Press, McGraw-Hill Book Co. (2000)

[9] Silva, J.P.M., Sakallah, K.A.: Grasp - a new search algorithm for satisfiability. In: ICCAD. (1996) 220–227

[10] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD. (2001) 279–285

[11] Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Dpll(t): Fast decision procedures. In: Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings. (2004) 175–188

[12] Dave, N., Arvind, Pellauer, M.: Scheduling as Rule Composition. In: Proceedings of Formal Methods and Models for Codesign (MEMOCODE), Nice, France (2007)

[13] Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification. (2002)

[14] Ki Yung Ahn. `http://hackage.haskell.org/package/yices`

[15] Johannes Waldmann. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/satchmo`