Improving Haskell Types with SMT

Iavor S. Diatchki

Galois Inc., USA
iavor.diatchki@gmail.com

Abstract

We present a technique for integrating GHC's type-checker with an SMT solver. The technique was developed to add support for reasoning about type-level functions on natural numbers, and so our implementation uses the theory of linear arithmetic. However, the approach is not limited to this theory, and makes it possible to experiment with other external decision procedures, such as reasoning about type-level booleans, bit-vectors, or any other theory supported by SMT solvers.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Constraints

Keywords Type systems, SMT, constraint solving

1. Introduction

For a few years now, there has been a steady push in the Haskell community to explore and extend the type system, slowly approximating functionality available in dependently typed languages [16–18]. The additional expressiveness enables Haskell programmers to maintain more invariants at compile time, which makes it easier to develop reliable software, and also makes Haskell a nice language for embedding domain specific languages [15]. The Haskell compiler is not just a translator from source code to executable binaries, but it also serves as a tool that analyzes the program and helps find common mistakes early in the development cycle.

Unfortunately, the extra expressiveness comes at a cost: Haskell's type system is by no means simple. Writing programs that make use of invariants encoded in the type system can be complex and time consuming. For example, often one spends a lot of time proving simple theorems about arithmetic which, while important to convince the compiler that various invariants are preserved, contribute little to the clarity of the algorithm being implemented [18].

Given that in many cases the proofs constructed by the programmers tend to be fairly simple, we can't help but wonder if there might be a away to automate them away, thus gaining the benefits of static checking, but without cluttering the program with trivial facts about, say, arithmetic. This paper presents a technique to help with this problem: we show one method of integrating an SMT solver with GHC's type checker. While we present the technique in the context of Haskell and GHC, the technique should be applicable to other programming languages and compilers too.

The approach presented in this paper uses similar tools—namely SMT solvers—to Liquid Haskell [14]. However, the design and overall goals of the two techniques are somewhat different and, indeed, it makes perfect sense to use both systems in a single project. Liquid Haskell enables Haskell programmers to augment Haskell definitions with refinements of the types, which may be used to verify various properties of the program. The refinement language is separate from the language of Haskell types—it is quite expressive, and is really aimed at formal verification. As such, a function will have its ordinary type and, in addition, a refinement type, which serves to specify some aspect of the behavior of the function. The checks performed by Liquid Haskell are entirely separate from the ones performed by GHC's type checker. This is quite different from the technique described in this paper, which shows how to integrate an SMT solver directly into GHC's type checker. The benefit of of extending GHC's type checker is that we don't need to introduce a separate language for specifications, instead we may reuse types directly. Ultimately, however, the two approaches are not at all mutually exclusive. The next section gives a brief flavor of what can be achieved using the techniques from this paper.

1.1 Examples

We illustrate the utility of the functionality provided by our algorithm with a few short examples. A very common example in this area is to define a family of singleton types that links type-level natural numbers to run-time constants that represent them:

```haskell
data UNat :: Nat -> * where
  Zero :: UNat 0
  Succ :: UNat n -> UNat (n + 1)
```

Here we've used a unary representation of the natural numbers, and each member of the family, `UNat n`, has exactly one inhabitant, namely the natural number `n` represented in unary form. Because we are using a GADT, we can pattern match on the constructors of the type and gradually learn additional information about the value being examined. The kind `Nat` is inhabited by types corresponding to the natural number (e.g., `0`, `1`, ...), and `(+)` is a type-level function for adding natural numbers.

Next, we define a function to add two such numbers:

```haskell
uAdd :: UNat m -> UNat n -> UNat (m + n)
uAdd Zero     y = y
uAdd (Succ x) y = Succ (uAdd x y)
```

While this is a simple definition, and we are unlikely to have gotten it wrong, it is nice to know that GHC is checking our work! Had we made a mistake, for example, by mis-typing the recursive call as `uAdd x x`, we would get a type error:

```
Could not deduce (((n1 + n1) + 1) ~ (m + n))
  from the context (m ~ (n1 + 1))
```

It is a small detail, but it is worth pointing out that our GHC extension would have been just as happy had we defined the type of Succ like this:

```
Succ :: UNat n -> UNat (1 + n)
```

The difference is in the return type, in the one case it is `n + 1`, and in the other it is `1 + n`. While to a human this looks like an insignificant difference, in many systems it is significantly easier to work with the one definition, but not the other.

The unary natural numbers are handy if we are planning to do iteration. However, if we are working on some sort of divide-and-conquer algorithm, we often need to split the input in two. In this situation, a different family of singletons is more useful:

```
data BNat :: Nat -> * where
  Empty :: BNat 0
  Even  :: (1 <= n) => BNat n -> BNat (2 * n)
  Odd   :: BNat n -> BNat (2 * n + 1)
```

This is another singleton family, where the type `BNat n` is inhabited by a single member, `n`. However, in this case we learn different information by pattern examining the values: an `Empty` value may not be split, an `Even` value may be split into two equal non-empty parts, while an `Odd` value may be split into two parts, and there will be one element left over.

So, how do we add such numbers? There are more cases to consider:

```
bAdd :: BNat m -> BNat n -> BNat (m + n)
bAdd Empty x            = x
bAdd x Empty            = x
bAdd (Even x) (Even y)  = Even (bAdd x y)
bAdd (Even x) (Odd y)   = Odd  (bAdd x y)
bAdd (Odd x)  (Even y)  = Odd  (bAdd x y)
bAdd (Odd x)  (Odd  y)  = Even (bSucc (bAdd x y))
```

The correctness of this definition is much less obvious, and we did make a couple of mistakes before getting it right! In the last case, we use the auxiliary function `bSucc`, which increments a binary natural number by one:

```
bSucc :: BNat m -> BNat (m + 1)
bSucc Empty     = Odd Empty
bSucc (Even x)  = Odd x
bSucc (Odd x)   = Even (bSucc x)
```

Note that while these examples are somewhat simplistic, they demonstrate the utility of the technology, and show that using the extension feels quite natural to a Haskell programmer.

A more conventional example is the type of fixed-length vectors:

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec 0 a
  Cons :: a -> Vec n a -> Vec (n + 1) a
```

Note that this is very similar to `UNat`, except that it also stores a value in each `Cons` node. The operation of appending two vectors together follows exactly the same pattern as `uAdd`. An example of a function that is meaningful for vectors, bit no so much for the unary natural numbers is reversing a vector, which may be defined as follows using our extension:

```
vecReverse = go Nil
  where
  go :: Vec m a -> Vec n a -> Vec (m + n) a
  go xs Nil = xs
  go xs (Cons y ys) = go (Cons y xs) ys
```

This example also illustrates that our extension does not change any type signature requirements beyond what is already in GHC:

the helper function `go` needs a type signature, because it pattern-matches using GADT constructors, however the type of `reverse` can be inferred as usual:

```
*Example> :type vecReverse
vecReverse :: Vec n a -> Vec n a
```

1.2 Structure of the Paper

The rest of the paper is organized as follows: we start with a brief overview of SMT solvers from a user's perspective (Section 2). Then, in Section 3, we introduce the basic concepts of GHC's constraints solver, which is necessary for putting the rest of the paper in context. Section 4 contains the details of the algorithm for integrating an SMT solver with GHC, and Section 5 explains other theories that could be added to GHC using the same technique. Finally, Section 6 discusses the possibility of using the ideas in this paper, and previous work on the implementation of SMT solver, to engineer a modular constraint solver.

2. SMT Solvers

This section contains an introduction to the core functionality of a typical SMT solver, and may be skipped by readers who are already familiar with similar tools.

2.1 The Core Functionality

SMT solvers, such as CVC4 [5], Yices [1], and Z3 [12], implement a wide collection of decision procedures that work together to solve a common problem. They have proved to be a useful tool in both software and hardware verification. From a user's perspective, the core functionality of an SMT solver is fairly simple: we may declare uninterpreted constants, assert formulas, and check if the asserted formulas are *satisfiable*. Checking for satisfiability simply means that we are asking the question: are there concrete values for the uninterpreted constants that make all asserted formulas true. The solver may respond in one of three ways:

- `sat`, which means that a satisfying assignment exists,
- `unsat`, which means that a satisfying assignment does not exist,
- `unknown`, which means that the solver did not find an answer, often this is due to reaching some sort of limit (e.g, the solver used up too much time).

This example uses the notation of the SMTLIB standard [4]:

```
(declare-fun x () Int)
(assert (>= x 0))
(assert (= (+ 3 x) 8))
(check-sat)
```

The example declares an integer constant, x, asserts some formulas—using a prefix notation—about it, and then asks the solver if the asserted formulas are satisfiable. In this case, the answer is affirmative, as choosing 5 for x will make all formulas true. Indeed, if an SMT solver reports that a set of formulas is satisfiable, typically it will also provide a *satisfying assignment*, which maps the uninterpreted constants to concrete values that make the formulas true.

The same machinery may also be used to prove the validity of a universally quantified formula. The idea is that we use the SMT solver to look for a *counter-example* to the formula, and if no such example exists, then we can conclude the formula is valid. For example, if we want to prove that $\forall x.(3+x = 8) \implies x = 5$, then we can use the SMT solver to try to find some x that contradicts it:

```
(declare-fun x () Int)
(assert (= (+ 3 x) 8))
(assert (not (= x 5)))
(check-sat)
```

To invalidate an implication, we need to assume the premise, and try to invalidate the conclusion, which is why the second assertion is negated. In this case the SMT solver tells us that the asserted formulas are not satisfiable, which means that there are no counter examples to the original formula and, therefore, it must be valid.

2.2   Incremental Solvers

Many SMT solvers have support for asserting formulas *incrementally*. This means that the solver performs a little work every time a new formula is asserted, rather than collecting all formulas and doing all the work in batch mode, once we ask about satisfiability. Furthermore, incremental solvers have the ability to mark a particular solver state, and then revert back to it, using a stack discipline. Here is an example:

```
(declare-fun x () Int)
(assert (= (+ 5 x) 3))
(push 1)
  (assert (>= x 0))
  (check-sat)
(pop 1)
(check-sat)
```

This example asks the solver two questions, and the answer to the first one is `unsatisfiable`, while the answer to the second one is `satisfiable` and the solution is `x = -2`.

The `push` command instructs the solver to save its state, then proceed as normal. When the solver encounters a `pop` command, it reverts to the last saved state.

This functionality is extremely useful when we want to ask many questions that are largely the same, and differ only in a few assertions: we can perform the majority of the work once, and then use `push` and `pop` to just assert the differences. In our small example, the work for the first assertion is shared in both calls to `check-sat`.

3.   GHC's Constraint Solver

In this section, we present relevant aspects of GHC's constraint solver. The full details of the algorithm [7] are beyond the scope of this paper.

3.1   Implication Constraints

During type inference, GHC uses *implication constraints*, which do not appear in Haskell source code directly. An implication constraint is, roughly, of the form $G \implies W$, where W is a collection of constraints that need to be discharged, and G are assumptions that may be used while discharging W. In the GHC source code, the constraints in G are referred to as *given constraints*, while the ones in W are known as *wanted constraints*. The intuition behind an implication constraint is that W contains the constraints that were collected while checking a program fragment, while G contains local assumptions that are available only in this particular piece of code. For example, consider the following program fragment, which uses a GADT:

```
data E :: * -> * where
  EInt :: Int -> E Int

isZero :: E a -> Bool
isZero (EInt x) = x == 0
```

When we check the definition of `isZero`, we end up with an implication constraint like this:

```
(a ~ Int) => (Num a, Eq a)
```

Here, `a` is the type of the pattern variable `x`, the wanted constraints arise from the use of `0` and `(==)` respectively, and the given constraint is obtained by pattern matching with `EInt` because we know that the type parameter of `E` must be `Int`.

3.2   The Constraint Solver State

Implication constraints are solved by two calls to the constraint solver: the first call processes (i.e., assumes) the given constraints, and the second one processes the wanted constraints.

The constraint solver has two central pieces of state: the *work queue*, and the *inert set*. The work queue contains constraints that need to be processed, while the inert set contains constraints that have already been processed. Constraints are removed—one at a time—from the work queue, and *interacted* with the solver's state. In the process of interaction we may solve constraints, generate new work, or report impossible constraints. If nothing interesting happens, then we relocate the constraint to the inert set. It is also possible that during interaction a previously inert constraint may be reactivated in a rewritten form and re-inserted in the work queue. A single invocation of the constraint solver keeps interacting constraints until the work queue is empty and all constraints become inert.

3.3   Type-Checker Plugins

In this paper we describe a fairly general extension to the constraint solver, but other researchers are interested in different extensions, for example, to add support for units of measure [2].

Instead of having many ad-hoc extensions directly in GHC's constraint solver, we collaborated to define and implement an API for extending GHC's functionality via *type-checker plug-ins*. At present, we are aware of three users of this API—our work, a related plug-in that can solve numerical constraints using custom rewrite rules [3], and the work on units of measure [2].

While this infrastructure is still brand new, and not yet stable, we hope that this mechanism may be useful to other researchers too, as it makes it fairly easy to experiment with various extensions to GHC's constraint solver. Since the framework is aimed at GHC developers, and is not intended for use in everyday Haskell programming, the plug-ins have a lot of power. Indeed, it is not difficult to write a plug-in that would violate type-safety, in much the same way one could do this by modifying GHC's constraint solver directly. The full details of the type-checker plug-ins API may be found in the GHC User's Guide [20]. Here we discuss only some of the design choices that we considered, as they have a direct bearing on the workings of our plug-in.

An interesting question about type-checker plug-ins is: at what point in GHC's type checker should we invoke them? The answer to this question affects their functionality, and the interface that a plug-in would have to implement. We considered a few alternatives:

1. Add a pass to the solver's pipeline, meaning that plug-ins process constraints one at a time, the way GHC does.

2. Add a call to the plug-ins after the constraints solver has reached an inert state.

3. Hand off implication constraints directly to the plug-ins, before invoking GHC's constraint solver.

Option 1 has the closest integration with GHC, but since plug-ins need to process constraints one at a time, then they often ended up needing some state, which than has to be stored and managed somewhere, which became rather complicated.

Option 3 is the most general, as it would allow for the plug-in to completely override GHC's behavior. However, we were more interested in *extending* GHC's capabilities rather than *replacing* them, so we opted against it.

We chose option 2 as a nice middle ground. It allows a plug-in to work with all constraints at once, but it has the benefit that standard work done by GHC has already happened. So, after the constraint solver reaches an inert state, it calls into the plug-ins, which examine the inert state and attempt to make progress. If they do, then the constraint solver is restarted and process repeats.

3.4 Improvement and Derived Constraints

An improving substitution [13] allows us to instantiate variables in the constraints with types, potentially enabling further constraint simplification. Of course, we should only do so, as long as we preserve soundness and completeness. In this context, preserving soundness means that the improved constraints should imply the original constraints (i.e., we didn't just drop some constraints), while completeness means that the original constraints imply the improved ones (i.e., we didn't loose generality by making arbitrary assumptions).

In GHC, improvement happens by rewriting with *equality constraints*. There are three sources of equality constraints:

- *given equalities* are implied by the given constraints,
- *wanted equalities* arise when solving wanted constraints,
- *derived equalities* are implied by the given and wanted constraints, together.

The provenance of an equality constraint determines the kinds of improvements that we can use it for. Given equalities have solid proof, and so we may use congruence to rewrite any other constraint. On the other hand, wanted constraints are goals that need to be proved, so they *cannot* be used to rewrite given constraints. We may still use them to rewrite other wanted or derived constraints though. Finally, derived equalities are implied by the given and wanted constraints jointly, so they may not be used in proofs directly, as doing so may lead to circular reasoning.

Instead, derived equalities help with type inference, by *guiding* the instantiation of *unification variables*. In general, it is always sound to instantiate a unification variable with whatever type we want: doing so may reject valid programs, but it will not accept invalid ones, because we still need to solve all necessary constraints. Of course, we don't want to reject valid programs, and this is where derived equalities help us: since they are implied by the goals and assumptions together, we know that we are not loosing any generality when instantiating as suggested by a derived constraint. So, for example, if we compute a derived constraint x ~ Int, and we have a wanted constraint Eq x, and x is a unification variable, then we may rewrite Eq x to Eq Int, which we would proceed to solve as usual. This is bit subtle: a derived equality is never used directly in a proof—instead it allows us to *change what we are trying to prove* by instantiating unification variables, which may help discharge all constraints that need solving.

4. Integrating GHC with an SMT Solver

We describe the algorithm in the context of the theory of linear arithmetic over the natural numbers, as having a concrete theory makes it easier to explain the process and illustrate it with examples. In the next section, we discuss how, and why, we might want to consider other theories also.

Input. Currently, the algorithm is implemented as a type-checker plug-in, and so the input to the algorithm is a collection of constraints that GHC has determined to be inert. This means that GHC simplified everything as much as it could, improved using equalities, and cannot see anything else to do. The inert constraints are presented to the plug-in in three collections based on the constraints' provenance: one group contains the given constraints (i.e., assumptions), one group contains derived constraints (see Section 3 for details), and one group contains the wanted constraints, which are the goals that need solving.

Output. The desired output of the algorithm is as follows:

- Solve as many wanted constraints as possible.
- Notice if the wanted constraints are inconsistent.
- Compute new given and derived equalities to help solve constraints that are outside this decision procedure's scope.

The first task is the most obvious purpose of the algorithm, but the other two are quite important also.

Noticing inconsistencies avoids the inference of types with unsatisfiable constraints. Consider, for example, a single wanted constraint, (x + 5) ~ 2. Since we are working with natural numbers, this constraint has no solution, so we cannot solve it. As a result, we may end up inferring a type like this:

```
someFun :: forall x. (x + 5) ~ 2 => ...
```

While, technically, this is not wrong, it is undesirable because the type error is clearly in the definition of someFun, but we would delay reporting the error until the function is called. So, we'd like to notice constraints that are impossible to solve (i.e., they are logically equivalent to ⊥), so that we can report type-errors that are closer to their true location. Note that the source of a contradiction may be a combination of constraints, and not just a single one. For example, the constraints x >= 5 and 3 >= x are inconsistent together, but have solutions when considered individually.

Computing new equations is also very important, as it enables collaboration between the plug-in and the rest of GHC (i.e., the main constraint solver, and other plug-ins). For example, consider an implication constraint of the form:

```
forall x. (x + 5) ~ 8  =>  KnownNat x
```

In this case, we'd like to use the SMT solver to compute a new given equality, x ~ 3. Then, this equality can be used by GHC to rewrite the wanted constraint KnownNat x to KnownNat 3 which, in turn, can be discharged by the custom solver for the KnownNat class[1]. Such collaboration between different solvers is quite common. Interestingly, this looks a lot like the collaboration between decision procedures in an SMT solver! We discuss this observation further in Section 6.

4.1 The Language of Constraints

Our first task is to identify constraints that are relevant to our solver. In the current implementation, we consider equalities and inequalities between a subset of the type-expressions of kind Nat. The kind Nat is inhabited by an infinite family of type-constants:

```
0, 1, 2, .. :: Nat
```

These constants may be combined and compared using the following type-level functions:

```
type family (+)   :: Nat -> Nat -> Nat
type family (*)   :: Nat -> Nat -> Nat
type family (<=?) :: Nat -> Nat -> Bool
```

These functions have no user-specified definitions: instead, we extended the core GHC simplifier with support for forward evaluation

---

[1] The class KnownNat is defined in module GHC.TypeLits. It is used to implement singleton types linking type-level literals to their run-time representations as Integer.

on concrete values, so it will evaluate concrete expressions, such as `2 + 3`. This simple forward evaluation cleans up the constraints, leaving the more complex reasoning—involving variables—to the algorithm being presently described. Technically, the clean-up is not necessary because the plug-in will perform as much evaluation as it needs to solve the constraints, but it is a lot more efficient to simply evaluate known constants without consulting the external solver. Also, at present there is no standard way to declare that these type function are handled by a special solver—we could use a closed type family with no equations to prohibit user defined instances, but this is a bit of an abuse of an unrelated feature of the type system.

The declaration for (`<=?`) refers to the kind `Bool`, which is simply the lifted `Bool` type. As expected, it is inhabited by the empty types `True` and `False`. Having (`<=?`) be a function that returns a boolean is a little more convenient for programmers than having a constraint. In this form, a programmer may inspect the result of the inequality and make a decision based on it. We can use a simple abbreviation to implement the corresponding inequality constraint:

```
type (<=) a b  = (a <=? b) ~ True
```

The constraints supported by the external solver are summarized by the following grammar:

$$c = e \overset{\mathbb{N}}{\sim} e \mid e \overset{\mathbb{B}}{\sim} e$$
$$e^{\mathbb{N}} = \alpha \mid n \mid e + e \mid n * e$$
$$e^{\mathbb{B}} = \alpha \mid \text{False} \mid \text{True} \mid e \leq^? e$$
$$n = 0 \mid 1 \mid \ldots$$

4.2 Importing Constraints

Of course, general constraints may be more complex. In particular, programmers are free to define their own type functions that return results of kind `Nat`, so it is entirely possible to encounter constraints like (`F 3 + 4`) `~` `7`, where `F` is some programmer-defined type-family. It is also possible to encounter non-linear constraints such as `x * y`.

The process of identifying relevant constraints is as follows:

1. Set aside constraints that do not have a relevant top-level predicate symbol.

2. Import the parameters to the predicates guided by the kind.

3. If we encounter a type outside of our theory, then we name it and replace it with a variable.

4. We remember the names assigned to various types, so that we may reuse the same name if a type appears multiple times.

For example, the constraint (`F 3 + 4`) `~` `7` will be imported as (`x + 4`) `~` `7`, and we will remember that `x` stands for `F 3`. Later on, if we need to return results to the core GHC solver that mention `x`, we replace it by the original expression, `F 3`.

The process of abstracting foreign constraints makes sense in our context, because it *generalizes* the constraint. Consider a constraint $C(t)$, and its generalized form $C(x)$, where the sub-term t is replaced by the variable x, which does not occur in $C(t)$.

Lemma. *If $C(t)$ is satisfiable, then $C(x)$ is also satisfiable.*

Proof. This follows because, by definition, $C(t) \iff C(x) \wedge x = t$. Therefore, if σ is satisfying assignment for $C(t)$, then $\sigma \cup \{x = \sigma t\}$ is satisfying assignment for $C(x)$. $\square$

Corollary. *If $C(x)$ is not satisfiable, then neither is $C(t)$. This is simply the contrapositive form of the previous Lemma.*

Recall from Section 2, that to prove something, we make sure that there are no counter examples. The Corollary states that if there are no counter examples to the abstracted constraint, then there will be no counter-examples to the original constraint as well, and so our proof is sound. From a logical stand-point, naming away sub-terms amounts to trying to prove a more general fact, and so if we can complete the proof in the more general setting, then the proof will work for the special case of the original constraint.

Aside. The step of naming foreign types is similar to the flattening step performed by GHC's constraint solver[2]. The main difference being that GHC names every call to a type-function, while we name only terms outside of the theory. Still, this duplication of work is somewhat unsatisfactory, and it would be nice to extend GHC's flattening pass so that it can be reused by external plug-ins.

Natural Numbers. We are interested in working with natural numbers, however, the SMT decision procedure works with integers. To work around this mismatch, we are careful that whenever we declare an SMT variable corresponding to a type of kind `Nat`, we also add a formula asserting that the variable is not negative.

4.3 Communication with the Solver

Different SMT solvers support different mechanisms for interacting with other programs. The most efficient way to interact with a solver is to link it with the program, and call into the various methods, using whatever API is exported by the solver. Unfortunately, this approach requires commitment to a specific solver as the low-level API of the solvers differ.

In our work, we wanted to have the flexibility to experiment with different solvers, so we opted for a slightly less optimal but considerably more portable approach. The multitude of SMT languages has been recognized as a problem by the SMT community, and the SMT-LIB standard [4] was developed to address the issue. A fair number of solvers have support for the standard, so to communicate with the solver we developed a Haskell library [9], which can interact with an SMT solver process by using the SMTLIB language. The library manages the connection to the solver, and also provides combinators to create terms in the common SMT theories. In the future, we may add support for direct integration with an SMT solver, and the only impact should be improved performance.

The code samples in the rest of this section use the following API for interaction with the solver:

```
data Solver :: *
data Expr   :: *
data Result = Sat | Unsat

solverAssert :: Solver -> Expr -> IO ()
solverCheck  :: Solver -> IO Result
solverPush   :: Solver -> IO ()
solverPop    :: Solver -> IO ()
```

These correspond directly to the commands used in the examples in Section 2.

4.4 Checking for Consistency

Given some constraints in our theory, we would like to know if a solution exists. If the constraints are inconsistent, then we can terminate the constraint solving problem early, and report an error, as discussed previously.

To check for consistency, we assert all constraints, and ask the SMT solver if the result is satisfiable. If this is not the case, then we report an error, otherwise we proceed with the algorithm.

---

[2] GHC's "flattening" pass names the results of all type functions. This is done before passing constraints to GHC's internal solver, but the process is reverted before passing the constraints to the external plug-ins.

```
checkConsistent :: Solver -> [Expr] -> IO Bool
checkConsistent s cs =
  do mapM_ (solverAssert s) cs
     res <- solverCheck s
     return (res == Sat)
```

Note that since we are working with generalized constraints, if we find an inconsistency, then we are sure that we've detected a problem. However, if the SMT solver says that the constraints are satisfiable, then we now that this is the case from the point of view of our theory, however the constraints might still be unsatisfiable if one took a global view of the problem. Consider, for example, (F a + 4) ~ 7. This will get imported into our plug-in as (x + 4) ~ 7, which is satisfiable by x = 3. Now, some other theory might know that F a ~ 3 is not actually possible, but we won't detect this problem at this stage.

4.5 Minimizing Conflicts

If we detect a contradiction, then we know that there is no possible solution to the constraints. However, often only a few of the constraints are involved in the actual problem, and it is much nicer if we report only them as the cause of the error, rather than getting a huge error involving all constraints. If an SMT solver detects that a collection of assertions is inconsistent, it may be able to compute an *unsatisfiability core*, which is the sub-set of the assertions that lead to the conflict—exactly what we want!

Unfortunately, not all SMT solvers support this feature and CVC4—the SMT solver that we used during the development of this work—does not have this ability. To work around this, we implemented a custom reduction algorithm which work as follows:

```
reduce :: Solver ->
          [Expr] -> -- part of conflict
          [Expr] -> -- search for conflict
          IO [Expr]
reduce s yes [] = return yes
reduce s yes mb =
  do solverPush s
     search s yes [] mb


search :: Solver ->
        [Expr] -> -- part of conflict
        [Expr] -> -- currently asserted
        [Expr] -> -- search for conflict
        IO [Expr]

search s yes mb (c : cs) =
  do solverAssert s c
     res <- solverCheck s
     case res of
       Unsat ->
         do solverPop s
            solverAssert s c
            reduce s (c : yes) mb

       _ -> search s yes (c : mb) cs

search s yes mb [] = error "Impossible!"
```

The algorithm keeps track of constraints that we are certain are part of the conflict in the variable yes. In addition, the algorithm has the invariant that the constraints in yes are always asserted in the solver's state, represented by the variable s. Initially, yes starts off empty.

We search for conflicts in the function search, which asserts constraints one at a time, until a conflict is discovered. The con-

straint that caused the conflict is added to the yes set, and then we examine the previously asserted constraint, to check if they are necessary for the conflict.

The reduction process performs in $O(n^2)$ time, where n is the number of constraints. This has not been a big problem, as this reduction happens only when we report errors and, typically, the number of constraints is fairly low.

Also note that the reduction algorithm may not produce the *smallest* collection of constraints to cause a contradiction. For example, consider the constraints A, B, C, and suppose that A, B lead to a contradiction together, and also C causes a contradiction on its own. Now, depending on the order in which we process the constraints, we may either compute C as the result of the reduction, or A, B. This is OK—in this case the context happens to contain multiple errors, so the reduction algorithm will simply pick one of them to report. Of course, a program would have to eventually fix both errors before the program is accepted. The important property of the algorithm is that the reduced set does not contain redundant constraints, in the sense that if we removed some of the constraints the resulting constraint set would be satisfiable.

4.6 Improvement

If all constraint are consistent (i.e., have at least one solution), then we check for computed equality constraints. As discussed previously, these help other parts of the constraint solver to make progress.

It is convenient to compute the improvements right after the consistency check because at this point we already have all constraints asserted in the solver's state. If all constraints in scope are given constraints, then we also generate given equalities, otherwise we generate derived equalities.

The current implementation considers three forms of improvement, two of which are completely generic, and one of which is specific to linear arithmetic.

Improve to Constant. As we know that the constraints are consistent, we can ask the SMT solver for a satisfying assignment. This assignment contains one possible solution to constraints, but we should not emit an improving equality unless we are sure that this is the *the only* possible solution. So, if $x = v$ is in the satisfying assumption, then we try to prove that the currently asserted constraints imply this fact. We do this by temporarily asserting $x \neq v$, and checking for satisfiability. If the resulting is unsatisfiable, then we know that v is the only possible value for x, and we can emit the corresponding improving equation.

```
solverProve :: Solver -> Expr -> IO Bool
solverProve s p =
  do solverPush s
     solverAssert s (SMT.not p)
     res <- solverCheck s
     solverPop s
     return (res == Unsat)

mustBeK :: Solver -> Name -> Value -> IO Bool
mustBeK s x v =
  solverProve s (eq (SMT.const x) (value v))
```

This process takes $O(n)$ time, where n is the number of variables. Note that here we are making use of the incremental capabilities of the solver, and reusing all asserted constraints, just doing one additional assert per variable. To see the process in action, consider the constraint (2 * x) ~ 16. The steps that we'd perform are:

```
(assert (= (* 2 x) 16))
(check-sat)
; SMT response: SAT
(get-value (x))
; SMT response: (8)
(push 1)
(assert (not (= x 8)))
(check-sat)
; SMT response: UNSAT
(pop 1)
; We generate a computed improvement: x ~ 8
```

Improve to Variable. We may also use the satisfying assignment
to look for improvements of the form $x = y$, where x and y are both
variables. We look at the satisfying assignment, ignoring variables
that were already improved to constants, and consider pairs of
variables that happen to have the same value in this assignment.
If x and y are two such variables, then we try to prove that $x = y$
must hold under the current assumptions:

```
mustEqual :: Solver -> Name -> Name -> IO Bool
mustEqual s x y =
  solverProve s (eq (SMT.const x) (SMT.const y))
```

This process take $O(n^2)$ time, where n is the number of variables,
but we only need to call the solver for pairs of variables that
have the same value—if they do not, then the current assignment
provides a counter example indicating that the variables do not need
to be equal in general.

Improve Using a Linear Relation. This improvement is a gen-
eralization of the previous example, that is specific to linear arith-
metic. The idea is to try to discover improving equations of the
form $y = A * x + B$. This type of improvement is a little dif-
ferent from the others, in that the right-hand side of the equation
contains terms that are in the theory that we are solving. In gen-
eral, such constraints are not very useful for the rest of GHC, as
other parts of the constraint solver do not know about our theory
(i.e., the functions `*` or `+`) and, if we can prove it, then we must
already know about this improvement! This sort of improvement,
is useful in one very common case, however: if y is a unification
variable, then having this type of improving equation enables GHC
to instantiate the variable, which leads to better type inference and
simpler constraints, so we attempt this improvement only when the
previous two have failed, and only for the unification variables in
the constraints.

We can compute the values for A and B, if we have two exam-
ples for x and two examples for y. It is quite standard to ask the
SMT solver for additional examples: this is done by adding addi-
tional constraints that force the models to differ. In our setting we
know that we can always get another such examples—if this was
not the case, a variable would have exactly one possible values, and
we would have improved it to a constant, as describe previously.
Furthermore, since we do the linear relation improvement last, we
usually can avoid extra calls to the solver because by this point we
typically already have two models where the variables differ. It is
also important that the x values be different, but we can always find
such examples: if we cannot find such an example, then x would
have only one possible value, and it would have been improved al-
ready.

Once we compute candidate values for A and B, we still need to
invoke the solver to validate that the computed relation holds in all
instances of the constraints, and not just the two that we happened
to pick.

To see when this improvement is needed, consider the following
program fragment:

```
f :: Proxy (a + 1)
f = Proxy

g :: Proxy (b + 2)
g = f
```

To check this program, GHC needs to solve the constraint
`(?a + 1) ~ (b + 2)`. The question mark in `?a` emphasizes that
this is a unification variable, standing-in for a type that needs to be
inferred. We cannot discharge this constraint immediately, because
it does not hold *for all* possible values of `?a` and `b`. However, since
`?a` is a unification variable, we have the option of instantiating it
to change the constraint. Of course, we have to be careful that in
doing so we do not loose any generality and, in this case, the safe
instantiation is `?a := b + 1`. The purpose of the linear-relation
improvement is to discover these kinds of instantiations.

At first sight, this example may appear artificial, however it is an
abstraction of a very common pattern: namely when one function
calls another, and the type of the provided argument does not
match syntactically the type of the expected argument. As another
example, consider the function `vecHead`, which requires that its
parameter has at least one element:

```
vecHead :: Vec (n + 1) a -> a
vecHead (Cons x _) = x
```

Now, if we tried to call `vecHead` with an expression of type
`Vec (b + 2) a`, we'd end up in exactly the same situation as
the previous example, where we'd have to infer that `?n = b + 1`.

Our current implementation only considers linear relations be-
tween pairs of variables as we were concerned about the perfor-
mance penalty if we tried combinations of more variables: the math
will work out, but the algorithm might become too slow to be prac-
tical, and alternative solutions based on rewriting might be more
suitable.

We admit that this improvement is somewhat ad-hoc, but as the
previous examples illustrate, it helps in situations that are quite
common in practice. It was also fairly simple to implement using
the solver.

Custom Improving Rewrites. All the improvements that we've
defined so far work by *only* using the solver—note that we were
just looking at satisfying assignments, and did not have any special
rules about the shapes of the constraints. It also makes sense to ex-
tend the system with custom rewrite rules, which has the potential
of speeding up performance, and also adding support for features
that go beyond the solver's capabilities. We have experimented with
various custom rules, but it is as yet unclear what constitutes a good
set of rewrites. Concrete custom rules that would help are various
normalization rules, for example applying distributivity repeatedly,
and also rules that rewrite equations so that there is just a unifica-
tion variable on the one side of the equation, which would help it
get instantiated.

4.7 Solving Constraints

The process of solving constraints is a straight-forward call to the
solver. The only thing we need to do before solving constraints is
to remove the assertions that were added during the consistency
and improvement stages. Recall that GHC solves implication con-
straints with two calls to the constraint solver: the first one asserts
the given constraints, while the second one actually solves goals.

While we are asserting the assumption, we just check for consis-
tency and improvement (to generate new given equalities), but we
perform no solving as there is nothing to solve. During the solving
stage, we mark the solver's state after we've asserted the givens,

then we assert the new goals, check for consistency, and improvement, and then, before solving, revert back to the state where only the givens are asserted.

```
solverSimplify :: Solver ->
                  [Expr] -> IO ([Expr],[Expr])
solverSimplify s wanteds =
  solverPrepare s wanteds $ \others our_wanteds ->
  do res <- mapM tryToSolve our_wanteds
     let (unsolved, solved) = partitionEithers res
     return (solved, unsolved ++ others)
  where
  tryToSolve (ct,e) =
    do proved <- solverProve s e
       if proved
          then return (Right ct)
          else return (Left ct)
```

The call to `solverPrepare` identifies the constraints that belong to our theory and translates them the SMTLIB language, where `others` are the constraints that are completely outside our theory, and `out_wanteds` are the constraints that we know about. Then, we invoke the solver for each goal, and see if we can prove it.

Evidence. When we solve a constraint, GHC expects some evidence to be produced, explaining why the constraint holds [6]. The same is expected when generating new given constraints. While this evidence is not used for code generation, it is a very useful for sanity checking while working with the constraint solver. Unfortunately, at present, our plug-in does not produce any meaningful evidence, beyond indicating that the fact was produced by using the SMT solver. In principle, SMT solvers should be able to produce a proof, when they have concluded that a set of assertions is unsatisfiable. Unfortunately, this is not a commonly used feature, so many provers do not support it out of the box, and the format of the proofs is not standardized.

4.8   Our Implementation

To demonstrate the feasibility of the ideas described in this section, we implemented a GHC plug-in, available from GitHub [10]. The implementation is intended as a proof-of-concept, and is not yet "industrial strength"—more engineering work would be required to make it more robust and easier to install and configure.

The majority of the effort so far has been focused on understanding how to integrate an SMT solver with the type-checker, and we have not conducted extensive testing and measurement of the performance of the plug-in. It is clear that calling out to an external tool has a cost associated with it. However, the preliminary results are encouraging—the plug-in functions quite well on small examples, but certainly more testing and experience is needed to find out if the approach will scale to larger projects.

If despite all the disclaimers, the reader is still interested in experimenting with the plug-in, we provide the list of GHC options and extensions that we used in the examples from this paper:

```
{-# OPTIONS_GHC -fplugin=TypeNatSolver #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE GADTs #-}
```

The option `fplugin=TypeNatSolver` instructs GHC to load the SMT solver plug-in. The rest are Haskell extensions that are commonly used in this setting:

- `DataKinds` enables natural number literals at the type level.
- `TypeOperators` and `TypeFamilies` are need to work with type-level functions such as `+`.
- `KindSignatures` is used to specify the kinds of type variables, for example, to state that some type-variable is of kind `Nat`.
- `GADTs` enables Generalized Algebraic Data-types; strictly speaking, these are not required to experiment with the plug-in, however, many of the more interesting uses of the plug-in end up using GADTs.

For the experiments in this paper, we used CVC4, version 1.4. There was no deep reason for using this particular solver, other that when we started working on the project this was the solver with the least restrictive license. To communicate with the solver, plug-in uses the SMTLIB format (version 2). Many other SMT solvers support this format (e.g., Z3, Yices), so they should work just as well.

5.   Other Theories

In the previous section, we described the core working of a procedure for integrating an SMT solver with GHC's constraint solver. While we concentrated on the theory of natural numbers and linear arithmetic, little of the algorithm is specific to that theory.

In principle, it should be fairly easy to add support for other theories, simply by declaring additional "uninterpreted" kinds and type-functions in Haskell. Of course—as is often the case—doing so is not completely trivial, which is why we have not done so yet. For example, while we have presented a mechanism for working with the symbols of a new theory in the type-checker, one still needs to make decisions about which symbols to import, or what notation to use. In the rest of this section we explore some of these possibilities.

5.1   A Theory for Booleans

We already made use of the lifted `Bool` type, but so far Haskell does not support any interesting operations beyond the `<=?` relation that we described. Programmers are already experimenting with defining type functions. For example, with the current version of GHC we may define a function for conjunction of type-level booleans:

```
type family And a b where
  And False x = False
  And True x  = x
```

Definitions like this work for simple evaluation, but do not support more sophisticated reasoning. For example, if GHC encounters the constraint `And x y ~ True`, it will not be able to conclude that the only way to solve this is if `x ~ True` and `y ~ True` both hold. Using the approach outlined in the previous section, we gain the full power of a SAT solver in the type system—SMT solvers can reason about booleans, indeed a SAT solver is usually an important part of the implementation of an SMT solver.

A good starting point for the theory of booleans, would be the following signature:

```
type family And (a :: Bool) (b :: Bool) :: Bool
type family Or  (a :: Bool) (b :: Bool) :: Bool
type family Not (a :: Bool)             :: Bool
```

5.2   A Theory for Integers

So far, our work uses natural numbers, which are quite useful for keeping track of the sizes of data-structures such as arrays, vectors, lists, bit-vectors, etc. The standard theory supported by the SMT solvers is actually linear arithmetic over the integers—in our

implementation we have to do some extra work to assert that we are only interested in non-negative solutions.

The main challenge for supporting both integers and natural numbers at the type level is not the technology for solving the equations, but the notation! One possible signature for the theory of type-level integers is:

```
type family ToInt  (a :: Nat) :: Int
type family Negate (a :: Int) :: Int
type family Add    (a :: Int) (b :: Int) :: Int
type family Sub    (a :: Int) (b :: Int) :: Int
type family Mul    (a :: Int) (b :: Int) :: Int
```

While this notation is somewhat verbose, at least it has the benefit of exposing the functionality to programmers, who may use a more suitable notation depending on their needs. For example, if a program works mostly with integers, one could add the following declaration, to get the more usual mathematical notation:

```
type (+) a b = Add a b
```

5.3 A Theory of Bit-Vectors

SMT solvers also have good support for reasoning about fixed-length bit-vectors. Unlike the theory of linear arithmetic, the theory of bit-vectors supports arbitrary multiplication and division, as well as various bit-wise operations, similar to the operations available in hardware.

Tracking Sizes. If we are interested in keeping track of the sizes of data-structures in memory, using 64-bit bit-vectors is a reasonable compromise which provides more expressive reasoning than linear arithmetic, and accommodates virtually all data-structures that fit in a computer's memory.

Finite Sets. Another potential application for the theory of bit-vectors is to implement finite sets of integers at the type level. As is usual at the value level, each bit is used to indicating the presence or absence of an element in the set, and the bitwise `and` and `or` operations correspond to intersection and union. Such functionality would be useful in the implementation of various effect systems.

6. A Modular Constraint Solver

In Section 4 we outlined an algorithm for using an SMT solver to solver a certain class of constraints, and then in Section 5 we observed that it is possible—and potentially useful—to solve many other constraints. This raises an interesting question: is it possible to structure the entire constraint solver for a complex programming language, such as Haskell or ML, using a modular approach, similar to the structure of an SMT solver?

We do not know that answer to this question, but the rest of this section, we present a little bit about the structure of SMT solvers, and point out similarities with GHC's constraint solver, which suggests that this might be a fruitful area for future research.

6.1 The Nelson-Oppen Method

An SMT solver uses a collection of independent decision procedures, and orchestrates them to compute an answer to a given query. A well-known technique for combining decision procedures is due to Nelson and Oppen [8]. Their algorithm starts with a collection of orthogonal theories (i.e., the symbols in the theories do not overlap), and shows how we may simplify terms in the combined language of the theories using *equality propagation*. The equality propagation procedure has a lot of similarities to the algorithm presented in this paper, and this is entirely by design on our part.

First, constraints that mention symbols from multiple theories are rewritten so that each constraint mentions only symbols in a single theory. This is done by naming "foreign" sub-terms, in the same way we described earlier in the paper.

Each collection of constraints is sent to the appropriate decision procedure to be checked for consistency. If one of the procedures detects an inconsistency, then we know that the whole collection of constraints is inconsistent and we may stop.

If the constraints look consistent, then the decision procedures are given a chance to "communicate" with each other via equality constraints. This is analogous to the improvement step in what we described. A key observation of Nelson and Oppen is that the only constraints that need to be communicated are equalities between variables. The intuition behind this observation is that other equalities would feature symbols that are "foreign" to the other decision procedures, and they would not know what to do with them.

So, if one decision procedure emits an equality constraint that was not known by some of the others, we propagate this information, and again check for consistency and further improvement.

Finally, the Nelson-Oppen technique also allows for a more general form of communication between procedures—a decision procedure may indicate that it can solve a problem in multiple ways, by emitting a *disjunction* of equality constraints. If this happens, then the coordinating logic has to explore all possibilities, the problem being satisfiable if any one of the options succeeds, and it is not satisfiable otherwise. Searching the options introduced by disjunctions is typically implemented with the aid of a SAT solver, which is usually considerably more efficient than simply trying all possibilities, as SAT solvers propagate information to prune the search space.

To implement the backtracking search efficiently, the decision procedures are typically incremental (i.e, they can revert back to a previous known state). Some techniques can improve this search process further, by using decision procedures that also produce proofs, [19], as these proofs can be used to identify dependencies between variables.

So, the overall setups is that we have a collection of independent decision procedures—each specializing in a single theory—and they communicate by disjunctions of equalities. There are three cases, depending on the shape of the disjunction:

- an empty disjunction indicates a conflict,
- a singleton disjunction is a certain improvement, and we propagate it across theories,
- a proper disjunction results in search.

6.2 Theories of Haskell

While GHC's constraint solver currently is structured quite differently from an SMT solver, the similarities—in particular, the use of equality constraints to disseminate information—suggest that there may be a more modular way to structuring the solver.

While GHC currently does not do any back-tracking search (i.e., decision procedure may only report inconsistency, or certain knowledge), this is also the case for Nelson-Open if all theories are convex [8]. In addition, there are interesting cases where backtracking might be potentially useful in Haskell too. For example, when solving class constraints in the context of overlapping instances, or instance chains [11], it is useful to consider the context of instances, but back-track and try a different instance, if the context is found to be unsatisfiable.

GHC's implementation of Haskell contains numerous extensions, and so there are many candidates for potential "theories", that is, constraints that are solved by a dedicated decision procedure. Engineering a constraint solver in this style would be very beneficial, both as one could reason about the correctness of each

decision procedure separately, but also, because it would make it simpler to experiment with new extensions to the system.

7. Conclusion

The paper introduces the basics of GHC's constraint solver, and shows how to connect it with an external decision procedure that produces satisfying assignments. The initial motivation for this work was to improve the type-level support for natural numbers, but the solution turned out to be more general, and opens up the door for interesting new extensions to Haskell's type system. More generally, we have identified similarities between the techniques used to implement SMT solvers, and the implementation of GHC's constraint solver, which suggests the possibility of a modular constraint solver design.

References

[1] *Yices Manual*, 2015. URL `http://yices.csl.sri.com/papers/manual.pdf`.

[2] Adam Gundry. A typechecker plugin for units of measure. Submitted for publication., 2015.

[3] Christiaan Baaij. A plug-in for normalising type literals in ghc. 2015. URL `https://github.com/clash-lang/ghc-typelits-natnormalise`.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. URL `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf`.

[5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, Snowbird, Utah, USA, 2011. URL `http://dl.acm.org/citation.cfm?id=2032305.2032319`.

[6] Dimitrios Vytiniotis and Simon Peyton Jones. Evidence normalization in System FC. In *24th International Conference on Rewriting Techniques and Applications*, Eindhoven, Netherlands, 2013. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/fc-new-tyco.pdf`.

[7] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, September 2011. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=162516`.

[8] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2), October 1979. URL `http://doi.acm.org/10.1145/357073.357079`.

[9] Iavor S. Diatchki. *simple-smt: a Haskell library for working with SMT solvers*, 2015. URL `http://hackage.haskell.org/package/simple-smt`.

[10] Iavor S. Diatchki. A plug-in for solving linear constraints using an smt solver. 2015. URL `https://github.com/yav/type-nat-solver`.

[11] J. Garrett Morris and Mark P. Jones. Instance chains: Type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*, Baltimore, Maryland, USA, 2010. URL `http://web.cecs.pdx.edu/~mpj/pubs/instancechains.pdf`.

[12] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference*, Budapest, Hungary, 2008. URL `http://research.microsoft.com/projects/z3/z3.pdf`.

[13] Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California, USA, 1995. URL `http://web.cecs.pdx.edu/~mpj/pubs/fpca95.pdf`.

[14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, Gothenburg, Sweeden, 2014. URL `http://goto.ucsd.edu/~rjhala/papers/refinement_types_for_haskell.pdf`.

[15] Patrick C. Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. Building embedded systems with embedded DSLs (experience report). In *Proceedings of the 2014 ACM SIGPLAN Conference on Functional Programming*, Gothenburg, Sweden, 2014. URL `https://github.com/GaloisInc/smaccmpilot-experiencereport/blob/master/embedded-experience.pdf?raw=true`.

[16] Richard A. Eisenberg and Jan Stolarek. Promoting functions to type families in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Gothenburg, Sweden, 2014. URL `http://www.cis.upenn.edu/~eir/papers/2014/promotion/promotion.pdf`.

[17] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 ACM SIGPLAN Symposium on Haskell*, Copenhagen, Denmark, 2012.

[18] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed Haskell programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Boston, Massachusetts, USA, 2013. URL `https://personal.cis.strath.ac.uk/conor.mcbride/pub/hasochism.pdf`.

[19] Sergey Berezin, Vijay Ganesh, and Daivd L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, 2003. URL `http://hci.stanford.edu/cstr/reports/2007-07.pdf`.

[20] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.10.1*, 2015. URL `https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/index.html`.