The Thoralf Plugin

For Your Fancy Type Needs

Divesh Otwani
Haverford College
Haverford, PA, USA
dotwani@haverford.edu

Richard A. Eisenberg
Bryn Mawr College
Bryn Mawr, PA, USA
rae@cs.brynmawr.edu

Abstract

Many fancy types (e.g., generalized algebraic data types, type families) require a type checker plugin. These fancy types have a *type index* (e.g., type level natural numbers) with an equality relation that is difficult or impossible to represent using GHC's built-in type equality. The most practical way to represent these equality relations is through a plugin that asserts equality constraints. *However, such plugins are difficult to write and reason about.*

In this paper, we (1) present a formal theory of reasoning about the correctness of type checker plugins for type indices, and, (2) apply this theory in creating Thoralf, a *generic* and *extensible* plugin for type indices that translates GHC constraint problems to queries to an external SMT solver. By "generic and extensible", we mean the restrictions on extending Thoralf are slight, and, if some type index could be encoded as an SMT sort, then a programmer could extend Thoralf by providing this encoding function.

CCS Concepts • **Software and its engineering → Functional languages**; **Constraints**;

Keywords GHC, constraint solver, type checker plugin, SMT

ACM Reference Format:
Divesh Otwani and Richard A. Eisenberg. 2018. The Thoralf Plugin: For Your Fancy Type Needs. In *Proceedings of the 11th ACM SIGPLAN International Haskell Symposium (Haskell '18), September 27–28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3242744.3242754

1 Introduction

As Haskellers, we want to use our type system to verify that our programs run correctly. Yet, despite the amazing progress GHC has made recently in supporting fancy types, some paradigms remain out of reach.

Specifically, there are certain kinds of *type level data* with equality relations that are difficult or impossible to support in GHC. Take, for example, the canonical length-indexed vector. It depends upon the type index of a natural number. Practical uses of length-indexed vectors require arithmetic expressions for vector lengths. Yet, GHC is not equipped to reason about equalities between such expressions. For instance, GHC cannot deduce $n + m \sim m + n$ for type level naturals n, m. Or, consider type level finite maps; we will see that we can support extensible records using a finite map index. Yet, GHC cannot decide equality between finite maps.

One common approach in dealing with type indices with fancy equality relations is to use type families [2, 7]. A type family is, essentially, a type-level function, allowing for computation in types. Because type family applications perform β-reduction, where the unreduced type is considered equal to the reduced type, they can be used to model certain sets with non-trivial equality relations. Type families can be quite effective and have supported type level data that allows for a type which tracks physical units-of-measure in the type system [13]. However, this approach is limited. Not all equality relations have the property that two terms can be tested for equality by following some deterministic procedure to produce unique normal forms. For example, type families cannot represent addition in a way so that GHC knows $a + b$ equals $b + a$ (and writing proofs is tedious).

When type families fail, we have two options: an open heart surgery of GHC's constraint solver and equality mechanisms or a *type checker plugin* [5, 9]. Undoubtedly, the most practical option is the plugin. A type checker plugin is a small constraint solver. If it can deduce more equality relationships than GHC's constraint solver, it can create equality axioms for GHC to use when type checking. Yet, even this option is *difficult*:

- GHC's type checker plugin interface does not provide a precise, tight specification of its behavior: there is no clear abstraction for how this interface interacts with GHC's internal solver. This means a plugin-writer is often guessing how their solving translates to type checking source code.
- A plugin writer must consider many annoying details about GHC's internals. Take, for example, GHC's

Haskell '18, September 27–28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5835-4/18/09…$15.00
https://doi.org/10.1145/3242744.3242754

type variables. There are essentially two forms of type variables: *skolem variables* and *unification variables*. Skolems are abstract type variables that behave as type constants: these are the type variables brought into scope when you are checking, say, the body of $id :: a \to a$ and cannot assume that a is Int. On the other hand, unification variables stand for unknown types not present in the source code; each one contains a mutable cell GHC fills in with a type without skolem variables. Since GHC represents both of these as type variables, a plugin writer could easily conflate them and write an unsafe plugin. Of course, a plugin writer has to consider many other similar challenges.
- The correctness condition for a plugin that supports some type-level data is non-trivial. How do we provide a specification for some type level data using its equality relation? How does this translate to a correctness condition on the plugin? *How do you know your plugin is type safe?*

Contributions So, our only feasible option for type level data, a type checker plugin, is painful. In this paper, we contribute

- A *theory* of reasoning about the correctness of type checker plugins that solve equality constraints without unification variables (which we will see is a sensible restriction). (Section 3)
- A *translation* of constraint solving to SMT satisfiability based on Diatchki [5]. (Section 4)
- A *generic* and *extensible* plugin for type level data called *Thoralf*[1] that implements our SMT translation. By "generic and extensible" we mean (1) despite Thoralf's restrictions, a vast collection of type level data (e.g., naturals, row types) could be supported via Thoralf extensions and (2) Thoralf can be extended by providing a function that "encodes" some type level data into a SMT sort. We provide examples of using Thoralf in Section 2 and discuss our two claims in Section 4.

Getting Thoralf Thoralf is available here:

https://github.com/Divesh-Otwani/the-thoralf-plugin

2 Examples of Using Thoralf

We start by looking at concrete examples of what Thoralf does and how to use Thoralf. In addition to concretely illustrating the problem we are solving, these examples serve as springboards for anyone who wants to use Thoralf to build fancy types.

[1]This is named after the logician Thoralf Skolem.

2.1 Natural Numbers with Arithmetic

We return to our canonical example: concatenation of length-indexed vectors.[2] We define length-indexed vectors to use GHC's built-in *Nat* type from *GHC.TypeLits*. This *Nat* type is convenient: we can use numerals in types at kind *Nat*, and we do not have to redefine basic arithmetic operations:

```
{-# OPTIONS_GHC -fplugin ThoralfPlugin.Plugin #-}

data Vec :: Nat → Type → Type where
  VNil :: Vec 0 a
  (:>) :: a → Vec n a → Vec (1 + n) a
infixr 5 :>

concatVec :: Vec n a → Vec m a → Vec (n + m) a
concatVec VNil       ys = ys
concatVec (x :> xs) ys = x :> (concatVec xs ys)
```

To use Thoralf, we write the options pragma at the top of the file.[3] With this pragma, this code can compile or load into a REPL successfully.

To see the problem that Thoralf solves, we can remove the pragma and inspect the resulting type error:

```
* Could not deduce: (1 + (n1 + m)) ~ (n + m)
  from the context: n ~ (1 + n1)
  Expected type: Vec (n + m) a
    Actual type: Vec (1 + (n1 + m)) a
```

Under the assumption that $n \sim (1 + n1)$, we want to prove $1 + (n1 + m) \sim (n + m)$. Even though this is straightforward with middle-school algebra, GHC's constraint solver cannot solve this problem on its own. Thoralf solves this problem by translating it to an SMT solver call.

2.2 Row Types and Extensible Records

Now that we've seen a basic example, we turn to a much more powerful and practical example: we use the API Thoralf provides for finite maps to create a row type. We use this row type to index a record type and build an extensible record type. We also build a tiny polymorphic function over our extensible record. In the process we will gain a stronger intuition for the constraint solving problem that Thoralf solves and any type-equality plugin encounters.

Thoralf comes with full support for type-level finite maps via the API in Figure 1. In our API, a type-level map from keys of kind k to values of kind v has kind $Fm\ k\ v$. Finite maps fm are generated from the following grammar:

$$fm ::= Nil \mid Alter\ fm\ k\ v \mid Delete\ fm\ k$$

where Nil is an empty map, $Delete\ fm\ k$ deletes the key k from fm, and $Alter\ fm\ k\ v$ changes fm to map k to v, updating the key k if it it already mapped in fm. However, our API exports only Nil and $FromList\ elts$, which builds a finite map

[2]This code snippet, among others, uses *Type* (from *Data.Kind*, GHC >= 8.0) as the kind of types with values, as opposed to ⋆ which could mean multiplication.

[3]Of course, calling `ghc` or `ghci` with that option would work as well.

```
data Fm k v where { }
  -- Exported interface:
type family Nil :: Fm k v where { }
type family FromList (list :: [(k,v)]) :: Fm k v where { }
type Has fm k v = fm ~ Alter fm k v
type Omits fm k = fm ~ Delete fm k
type AddField fm1 fm2 k v = fm2 ~ Alter fm1 k v
type DelField fm1 fm2 k = fm2 ~ Delete fm1 k
  -- Not exported:
type family Alter (fm :: Fm k v) (key :: k) (val :: v) :: Fm k v
  where { }
type family Delete (fm :: Fm k v) (key :: k) :: Fm k v
  where { }
```

Figure 1. Thoralf finite map API

from an association list by repeatedly applying *Alter* to *Nil*. *Alter* and *Delete* are not directly exported, because doing so would violate the design principles laid out in Section 4.4.

Thoralf exports four constraints on maps:

- *Has fm key value* asserts that the map *fm* maps *key* to *value*.
- *Omits fm key* asserts that *fm* does not map *key*.
- *AddField* fm_1 fm_2 *k v* asserts that $fm_2 \sim$ *Alter* fm_1 *k v*. Note: fm_1 could map *k* to some value not equal to *v*.
- *DelField* fm_1 fm_2 *k* asserts that $fm_2 \sim$ *Delete* fm_1 *k*. Note: fm_1 might not map *key*; in this case $fm_1 = fm_2$.

2.2.1 Using Thoralf's Finite Maps to Build Extensible Records

What are Extensible Records? There are two defining features of extensible records. First, unlike Haskell's usual records whose members and types are fixed at a declaration site, we can add field-value pairs to an extensible record. Second, we can write *polymorphic* functions that work over any extensible record with a few required fields. For example, we could write a *getName* function that retrieves the `"name"` field of an extensible record, as long as that field exists and has type *String*. The other fields are irrelevant. Note that, of course, users can look up and update fields like they would with an ordinary record (though with different syntax).

Thoralf's Extensible Record This is how we build extensible records using Thoralf:

```
data Record :: Fm Symbol Type → Type where
  EmptyRec  :: Record Nil
  AddField  :: AddField m m' field valty
            ⇒ Record m → SSymbol field → valty
            → Record m'
```

Record is indexed by a row type. In our system, a row type is a type level finite map from GHC's type-level strings, called *Symbol*s, to *Type*s. The finite map holds the field names and types of the values stored in the record.

The datatype has two constructors. The first holds no data, and correspondingly has an empty finite map index. The *AddField* constructor takes an existing finite map, a singleton string, and a value; it constructs a new *Record* indexed by a finite map with that modified field, via the *AddField* constraint.

AddField takes a *singleton* string argument. Singletons [8] are a well-known technique for simulating dependent types in a non-dependent programming language [12]. A singleton type has exactly one inhabitant. For example, with singleton strings, the data of type *SSymbol* `"name"` will store the string `"name"` at runtime; the data of type *SSymbol* `"price"` will store the string `"price"` at runtime. In general, when we learn that the runtime data in a *SSymbol str* is, say, `"hi"`, we also learn that the type index *str* is `"hi"` and vice versa.

We need this correspondence for the *AddField* constructor. Consider what would happen if we provided a *String* instead of a *SSymbol str*. The finite map type index would have no way to store the name of the field that was just added. With a field name of *SSymbol str* we can store the *str* with our *AddField* constraint from the API.

A Polymorphic Record Function In this example we traverse a *Record m*, looking to extract the `"price"` field. This function is polymorphic in that it works on any *Record m* that has a `"price"` field which holds *Int*s. That is, it works on any *Record m* where the constraint *Has m* `"price"` *Int* is satisfied.

```
getPrice :: Has m "price" Int ⇒ Record m → Int
getPrice (AddField rec fld val) =
  case scomp fld (SSym @"price") of
    Refl     → val
    DisRefl → getPrice rec
```

This function is accepted by GHC when running the Thoralf type checker plugin. Though it seems simple, there is a lot going on here! The first step to explaining this function is understanding the function *scomp*.

How Does scomp Work? The *scomp* function decides equality on *SSymbol*s, returning either a proof that *s1* equals *s2* via *Refl* or that *s1* is different from *s2* via *DisEquality*.

```
scomp :: SSymbol s1 → SSymbol s2 → s1 :~?~: s2
data a :~?~: b where
  Refl    ::                    a :~?~: a
  DisRefl :: DisEquality a b ⇒ a :~?~: b
class DisEquality (x :: k) (y :: k) where { }
```

The *DisEquality* class is meaningless to GHC but is used within Thoralf to represent that two types are distinct[4]

The implementation of *scomp* converts its input singleton *SSymbol*s to regular strings and checks string equality. It uses this term-level information to deduce that the types *s1* and *s2* must match or cannot match and *unsafeCoerce*s the appropriate result. In general, this is how *DisEquality* constraints are created—through singleton comparisons and judicious uses of *unsafeCoerce*.

How Does getPrice Work? The one equation of *getPrice* matches on the *AddField* constructor. From this, the equality constraint *AddField m1 m field valty* is brought into scope where

- *rec* :: *Record m1*,
- *fld* :: *SSymbol field*,
- *val* :: *valty*, and,
- *AddField rec fld val* :: *Record m*.

The function pattern matches on the result of comparing *fld* with the singleton version of `"price"`. In the first case, the singleton strings are the same. At the type level, we learn *field* ~ `"price"` from the *Refl*. Then, we make a deduction. If (1) *Has m* `"price"` *Int*, or, equivalently, *m* ~ *Alter m* `"price"` *Int* and (2) *m* ~ *Alter m1* `"price"` *valty* (substituting `"price"` for *field*), then *valty* must be *Int*. In other words, since *m* maps `"price"` to *Int* and `"price"` to *valty*, by the definition of finite maps, *Int* ~ *valty*. So, returning *val* :: *valty* is a well-typed return value of type *Int*.

In the latter case, the singleton strings are different. At the type level, we now learn *field* ~/~ `"price"` from the *DisRefl* match. As before, we make a deduction. Since (1) *m* ~ *Alter m* `"price"` *Int*, (2) *m* ~ *Alter m1 field valty*, and (3) *field* ~/~ `"price"`, we reason that the inner finite map must map `"price"` to *Int*: *Has m1* `"price"` *Int*. That is, since the map *m* must map `"price"` to *Int* and *m1* and *m* share the same mappings except for *field*—which is not `"price"`—we must have *m1* mapping `"price"` to *Int*. Consequently, the record indexed by the *m1* map should have a "price" field with an *Int* value, and hence our recursive call *getPrice rec* is sensible. Thoralf confirms our reasoning and convinces GHC to accept this code.

Note that mimicking the string equalities and disequalities at the type level with with matches on *Refl* and *DisRefl* was essential. It is insufficient to simply reason at the term level.

The Type Error Thoralf Resolves Again, to concretely illustrate the problem Thoralf solves, we observe one of the type errors without Thoralf. As expected, this matches our second deduction:

```
* Could not deduce: m1 ~ (Alter m1 "price" Int)
  from the context: m ~ (Alter m1 field val)
  or                m ~ Alter m "price" Int
  or from           DisEquality "price" field
```

[4]We view *DisEquality* as a closed class, with no instances. An alternative design could represent this with a closed type family.

3 A Theory of Type-Equality Plugins

Having seen Thoralf at work, we take a step back and present the theory behind what Thoralf does and how to reason about Thoralf's correctness.

However, before we can discuss correctness we need to understand the type checking process with a type-equality plugin. Then, our first concern is the bare-minimum requirement: how do we know a type-equality plugin, and specifically Thoralf, is type safe? Beyond the bare minimum, we want some correctness with respect to our idea of some type level data. For example, type level finite maps should behave like finite maps. Toward this end, how do we specify some type level data? How does this specification translate to a *specification for the constraint solver* of a type-equality plugin?

We answer these questions in this section and provide a correctness condition for a sensibly restricted form of type-equality plugin that, like Thoralf, does not solve problems with unification variables.

3.1 Type Checking with a Type-Equality Plugin

We start by generalizing how GHC type-checked the examples we saw. The plugin asserted type equalities that allowed our code to type check. Well, what does it mean to "assert type equalities to GHC"?

Concretely, it means a type-equality plugin resolves type errors of the form:

```
Main.hs: error:
    * Could not deduce: <ty1> ~ <ty2>
      from the context: <g1> ~ <g1'>
      or                <g2> ~ <g2'>
                ...
      or                <gn> ~ <gn'>
      or                <h1> /~ <h1'>
                ...
      or                <hk> /~ <hk'>
```

This type error says that GHC is unable to deduce some wanted type equality under the assumption of other type equalities and disequalities.

When a plugin resolves this type error, it asserts to GHC (without needing to provide a proof), that `<ty1>` and `<ty2>` are in fact equal. GHC unquestioningly accepts this fact as an axiom and continues type checking.

How does type checking proceed after a plugin asserts a type equality? Either (1) some code type-checks or (2) GHC uses this equality to deduce other equalities. For an example of (1), consider the two case matches of *getPrice*. In the first match, Thoralf asserted *valty* ~ *Int* and the return value *val* type checked. In the second match, to type check the recursive call *getPrice rec*, GHC needed to satisfy the

constraint $m1 \sim \mathit{Alter}\ m1$ "price" Int, which was exactly what the plugin produced.

For our purposes a specific form of (2) occurs: equalities between type indices, such as $1 + (n1 + m) \sim (n + m)$ from *concatVec*, determine equalities on the type they index via GHC's assumption of congruence. When GHC knows $\tau_1 \sim \tau_2$, it can deduce $T\ \tau_1 \sim T\ \tau_2$. Continuing with our *concatVec* example, GHC could deduce $\mathit{Vec}\ (1+(n1+m))\ a \sim \mathit{Vec}\ (n+m)\ a$. GHC then applies the casting rule

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \Vdash \tau_1 \sim \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{Conv}$$

to say $(x :> (\mathit{concatVec}\ xs\ ys)) :: \mathit{Vec}\ (n+m)\ a$ and type check the return value of the second equation of *concatVec*.

All in all, this is the type checking process with a type-equality plugin:

- GHC's constraint solver sends the plugin problems it is unable to solve. These are problems that produce type errors of the general form we have presented: deducing a wanted equality from some given equalities and disequalities.
- The type-equality plugin sometimes asserts the wanted equality it is given.
- GHC uses the equality axiom from a plugin as it normally would to type check code, sometimes using congruence to deduce other type equalities.

3.2 Type Safety

If a plugin is injecting new axioms into GHC's type equality relation, how can we be sure not to break type safety? That is, how do we know when we have accidentally equated *Int* with $\mathit{Bool} \to \mathit{Bool}$, allowing a user to call 5 as a function and jump to arbitrary memory? Let's first consider the axioms produced by the plugin itself and then look at what GHC does with those axioms.

Equality on type-level data Without the help of a plugin, GHC implements equality on types. If, according to the rules of GHC's type equality (for example, as explained by Breitner et al. [1]), two types can be considered equal, GHC will prove this. As such, the primary work of a plugin is not to compute equality on proper types like *Int* and *Bool*, but instead to compute equality on *type-level data* or *type indices*.

While Haskellers casually refer to all subtrees of a type as types—that is, if we have $v :: \mathit{Vec}\ (n+m)\ \mathit{Bool}$, then we might say that the subtree $(n + m)$ is a type—this is not quite true: $(n + m)$ is a *number*, not a type. It happens to be used within a type and syntactically at the type level, but that does not make it a type. By contrast, *Int*, *Bool*, and *Maybe Double* are types. *Maybe* and *Either* are type constructors, which become types when given appropriate arguments. Instead of calling $(n + m)$ a type, we propose calling it *type-level data*.

These bits of type-level data are frequently used to *index* a type, such as in $\mathit{Vec}\ (n+m)\ \mathit{Bool}$. Because of the congruence of equality, proving equality relationship on type indices can indeed induce equalities on types themselves. Thus, while asserting new equalities on *types* is generally unnecessary for a plugin, proving equalities on *type indices*, declared at kinds other than *Type*, is more sensible.

If a plugin is working in a given domain (such as the natural numbers), we generally do not wish for GHC to interfere with the plugin's work. For example, suppose we somehow know that $m \not\sim n$, but we are trying to prove $(n + m) \sim (m + n)$. It would be a shame if GHC decomposed this equality and tried to prove $n + m$ (the left-hand arguments) and $m \sim n$ (the right-hand arguments). These equality checks would (rightly) fail. Instead, we need to keep GHC away from concepts it knows nothing about (like numbers,[5] or finite maps). Happily, GHC refuses to look under type family applications, as type families are neither injective nor generative [6]. If we make + a closed type family [7] with no equations, then GHC will not interfere, giving our plugin the full opportunity to solve the equalities [9].

We thus have some design principles:

- Plugins should proof equalities over type-level data, of kinds different from *Type*.
- All operations in a plugin's theory should be written as empty closed type families.

Another way of stating this is that we want a *kind-indexed* equality relation. While GHC is free to use its internal equality relation (essentially, structural equality with α-equivalence) on proper types, we want to impose a different equality relation on our type indices.

Wonky plugins Having stopped GHC from meddling with our plugin's theory, the plugin is free to produce equality axioms as necessary. Interestingly, if all a plugin does is to produce axioms over type indices, *there is no way to break type safety*. The design principles above are important in leading to this conclusion. Let us explore via an example.

We start with the following definitions:

```
data Number :: Type

type family N (n :: Nat) :: Number where { }
type family (a :: Number) +. (b :: Number) :: Number
  where { }
type family (a :: Number) −. (b :: Number) :: Number
  where { }
```

Here, we have declared a new type *Number*, which will serve as the kind of our new type-level data. The *N* empty type family converts built-in *Nat*s to *Number*s, and we have defined addition and subtraction over these numbers. Let us further define length-indexed vectors with these numbers:

[5] GHC actually ships with a rudimentary ability to deduce natural-number relationships when using its built-in *Nat* kind. This solver is quite simplistic, and we can safely pretend it does not exist.

```
data Vector :: Number → Type → Type where
  VVNil :: Vector (N 0) a
  (:≫)  :: a → Vector n a → Vector (n +. N 1) a
```

Now, let us suppose that the plugin giving meaning to these numbers equated *all* numbers. That is, the following would be accepted:

```
silly :: Vector (N 2) a → Vector (N 3) a
silly VVNil =    VVNil
silly (_ :≫ v) = v
```

Accepting this requires proving $N\ 0 \sim N\ 3$ and $n \sim N\ 3$ (where we assume $n+.\ N\ 1 \sim N\ 2$), both of which our plugin handily provides. One might easily think that our *Vector* type is now unsafe, but there is no type safety problem here. Instead of defining a length-indexed vector, we have defined a type functionally identical to the regular old list type, [ ]. If [ ] is type safe, then so is *Vector*.

What if we equated only *some* numbers some of the time? What if our plugin non-deterministically equates *Number*s, even changing its mind about the same query if posed multiple times? These deficiencies, too, cannot cause GHC to lose type safety. Critically, GHC makes almost no use of a failed check for equality. It tracks no disequality constraints, and lack of equality does not imply apartness (used in the reduction of closed type families and well explored by Eisenberg et al. [7]). A wonky plugin might cause type inference to become unpredictable or otherwise off-kilter, but it will not launch the rockets.

Despite not launching any rockets, two problems may surface with a wonky plugin:

Pattern-match warnings Above, we said that GHC makes *almost* no use of failure to solve an equality check. The one place it does use failure is in deciding whether a pattern match is complete. Consider *silly* again. If the plugin flatly refused to allow $N\ 2$ to equal $n+.\ N\ 1$ (the resulting index in :≫), then the pattern-match completeness checker [10] would say that the second clause is redundant. Note that this drawback does not break type safety—instead, it means that a wonky plugin could lead a programmer to erroneously believe that an incomplete pattern match is complete. This problem might cause an unexpected exception at runtime, but it cannot cause other arbitrary behavior.

Reliability of specifications When we write, say, *reverse* :: *Vector n a* → *Vector n a*, we understand that to mean that *reverse* preserves lengths of vectors. However, with our wonky plugin, this type is no more informative than $[a] \to [a]$. Thus, a wonky plugin might not break type safety, but it very well might violate invariants that the programmer intends to encode.

For our wonky plugin to break type safety, there would have to be some way to branch on different members of *Number*. That is, we would need a way of treating them as distinct types in GHC's type equality reasoning. However, all members of *Number* are stuck empty closed type families, and there is no way to branch on these. We are thus safe.

Note that GADTs, by themselves, do not allow this kind of branching, as we always look at a runtime constructor in a GADT pattern match, never solely type-level data.

Equalities on types* can *break type safety The discussion above all assumes that the plugin "keeps to itself"—injecting equality proofs only on type-level data of some kind other than *Type*. However, it is also possible, of course, for a plugin to introduce an equality between types. Indeed, the finite maps example (Section 2.2) does this to good effect. The key step there is that, from

$$fm_1 \sim fm_2, Has\ fm_1\ \texttt{"x"}\ Bool, Has\ fm_2\ \texttt{"x"}\ ty$$

we can conclude $ty \sim Bool$. This conclusion is an utterly different beast than what we have considered before: it is an equality on *types*. If the plugin did not faithfully implement a theory of finite maps, this equality might be bogus. We thus have another design principle:

- Equality axioms relating GHC types (and only those) *must* be correct.

By *correct* here, we mean that there must exist a consistent model in which the assumptions entail the desired equality. Owing to the soundness of the theory of arrays [15] (readily used to model finite maps), we can be confident that concluding $Bool \sim ty$ above is type-safe.

3.3 The Formal Type-Equality Constraint Solving Problem

Naturally, type safety is not the only property we would like to have. For example, in the case of *Vector*, we really want the index on the types to be the length of the linked list stored at runtime. We present here a formal framework for how to develop a plugin that is not wonky—that is, respects the desired semantics of the type-level data in question.

3.3.1 Building the Grammar

The first step, as usual in a formal system, is to define the *grammar* of the data under consideration. For example, recall the grammar of finite maps: *fm*. It depends upon the grammars for the key and value sets, K and V, respectively, which we leave abstract. We assume that $k \in K$ and $v \in V$.

$$fm ::= Nil \mid Alter\ fm\ k\ v \mid Delete\ fm\ k$$

That is, a finite map is either *Nil*, the addition to an existing map, or the restriction of an existing map. More generally:

Definition 3.1 (Type Index & Equational Theory Sets)**.**

- Let $T_1, \ldots, T_n$ be a list of sets (not necessarily distinct) which we call *background theories*.
- Let $E_1, \cdots, E_n$ be a corresponding list of *background equivalence relations* (that is, $\forall i, E_i \subseteq (T_i \times T_i)$) with

the property that no background theory has two equivalences[6]: if $T_i = T_j$ then $E_i = E_j$.

- Let $\mathbb{T}$ be an operator that takes in n background theories and produces a grammar. That is, $\mathbb{T}(T_1, \cdots, T_n)$ is a grammar where metavariables can be drawn from input sets $T_1, \cdots, T_n$. This grammar gives rise to a set of terms; we conflate the notation of the grammar and the set of terms it gives rise to. Let
$$\mathcal{I} = \mathbb{T}(T_1, \cdots, T_N)$$
be the *type index set.*
- Similarly, we say
$$\mathcal{E} = \mathbb{E}(E_1, \cdots, E_n)$$
is the *equational theory of the type index set*: $\mathcal{E} \subseteq (\mathcal{I} \times \mathcal{I})$.
- Define
$$\hat{T} = \mathcal{I} \cup \left(T_1 \cup \cdots \cup T_n\right)$$
and
$$\hat{E} = \mathcal{E} \cup \left(E_1 \cup \cdots \cup E_n\right)$$
to represent the set of all *literals* and the *equational theory of literals*. We use the word "literal" because the types mentioned here lack abstract variables, which we add later.

Let us unpack that definition. We motivate these by looking at the sets defined for the finite maps. The background theories of finite maps are the sets of keys and values: K and V. These have corresponding equational theories E_1, E_2. The grammar of finite maps is defined in terms of these sets: $\mathcal{I} = \mathit{fm}$ (where fm is from above). The equational theory of finite maps is defined in terms of the equational theories of the key and value sets. For example, the equational theory of fm satisfies the following inference rule:

$$\frac{k \sim k'}{\mathit{Alter}\ (\mathit{Alter}\ \mathit{fm}\ k\ v_1)\ k'\ v_2 \sim \mathit{Alter}\ \mathit{fm}\ k\ v_2}$$

Note that this rule uses the premise $k \sim k'$, drawn from the equivalence relation over K (which we have called E_1).

We write $\mathcal{E} = \mathbb{E}(E_1, E_2)$ for the equational theory of finite maps. The set of all literal types includes all the finite maps, keys and values: $\hat{T} = \mathit{fm} \cup K \cup V$. The equational theory of the literal types is $\hat{E} = \mathbb{E}(E_1, E_2) \cup E_1 \cup E_2$.

3.3.2 Adding Skolem Variables

Having defined literal type-level data (expressions such as $3+4$ or $\mathit{Alter}\ \mathit{Nil}$ `"x"` Bool without variables), we can extend this definition to include what we actually see in type errors. These include variables. As noted previously, these variables come in two flavors: skolem variables and unification variables. In the errors we have seen so far, only skolems have been present; this is by design. We discuss this design and the distinction between these type variables in Section 4.3.

[6] A theory, like natural numbers, should have only one equivalence relation.

For now, we add only skolem variables to our grammar for type-level data.

Here, we formalize the type-level data we see in type errors and call such types *abstract data.*

Definition 3.2 (Abstract data)**.**

- Declare pairwise disjoint enumerable sets of variable names $X, X_1, \ldots, X_n$ where $X \cap \mathcal{I} = \emptyset$ and $X_i \cap T_i = \emptyset$.
- For any grammar A and set Y, we define the *augmented grammar*
$$A[Y] ::= (\cdots \text{production rules of } A \cdots) \mid y$$
where $y \in Y$.
- Define
$$\boldsymbol{T_i} = T_i[X_i],$$
$$\boldsymbol{\mathcal{I}} = \mathcal{I}[X],$$
and
$$\boldsymbol{T} = \boldsymbol{\mathcal{I}} \cup \left(\bigcup \boldsymbol{T_i}\right).$$
Here, $\boldsymbol{T}$ represents the *abstract data*, the set of types that could appear in the type errors we encounter.

All we are doing here is augmenting each set with the ability to hold variable names, where each set has its own distinguished set of names to choose from.

3.3.3 Equality Constraint Solving

Now that we have a set which models the types we see in our type errors, we can describe the inputs and outputs to the constraint solving problem we encounter.

All our type errors try to deduce a single *wanted* equality $\tau_1 \sim \tau_2$ from a set of *given* equalities and disequalities in the context, $\tau_i \sim \tau_i'$ and $\sigma_j \nsim \sigma_j'$. Here, we can concretely view each equality constraint and disequality constraint as an element of the set $\boldsymbol{T} \times \boldsymbol{T}$.

Definition 3.3 (Equality Constraint Inputs and Outputs)**.**

- The inputs are a set of *wanted equality constraints*,
$$W \subseteq (\boldsymbol{T} \times \boldsymbol{T})$$
and givens, comprising a set of equalities and a set of disequalities
$$(G_e, G_d) \text{ where } G_e, G_d \subseteq (\boldsymbol{T} \times \boldsymbol{T}).$$
- The output is one of three outcomes:
  1. A result of $\bot$ means that the context of given equalities is inconsistent or nonsensical.
  2. The set W represents that all the wanted equalities can be deduced from the givens.
  3. $\emptyset$ indicates that at least one of the wanted equalities cannot be deduced.

Now that we can precisely describe the constraint solving problems the plugin sees, we can specify the correctness condition for its constraint solver.

3.3.4 Equality Constraint Solver Specification

At a high level, we wish to deduce a wanted equality from a set of givens if the wanted is true in any model where the given equalities and disequalities are true. We unpack this high level intuition. First, what is a model, exactly?

Definition 3.4 (Models)**.**

- A function $\varphi : \boldsymbol{T} \to \hat{T}$ is a *model on* $\boldsymbol{T}$ if it homomorphically substitutes all skolem variables in its argument with literals. That is, φ leaves literals untouched but substitutes the variables according to some substitution functions
$$(\pi : X \to \hat{T}), (\pi_1 : X_1 \to T_1), \ldots, (X_n \to T_n).$$
- If $G = (G_e, G_d)$ is the set of givens, the set of *G-consistent models on* $\boldsymbol{T}$ is the set of models Φ_G where

$$\Phi_G = \{\varphi \mid (\forall e \in G_e, \varphi(e) \in \hat{E}) \wedge (\forall d \in G_d, \varphi(d) \notin \hat{E})\}$$

What is going on here? A *model* is a function φ that chooses literal values for each free variable of some type-level data. Building from there, Φ_G is the *set of models* that are consistent with assumptions G—that is, applying a $\varphi \in \Phi$ to the components of a given equality yields two literals considered equivalent by $\hat{E}$; doing the same to a given disequality yields two distinct literals. Now, we define the set of models where the givens are true:

Definition 3.5 (Deductive closure)**.** The *deductive closure* of givens G is defined to be

$$\mathcal{DC}(G) = \{e \in \boldsymbol{T} \times \boldsymbol{T} \mid \forall \varphi \in \Phi_G, \varphi(e) \in \hat{E}\}.$$

According to this definition, $\mathcal{DC}(G)$ is the set of pairs of data where the first member of the pair equals the second in *all* models consistent with the givens G. We can now define concretely what a plugin solver does and how to judge its correctness, naming the solver PluginSolve.

Definition 3.6 (The PluginSolve Correctness Condition)**.** Let $\mathcal{I}$ be a description of type-level data with background theories $T_1, \cdots, T_n$. Let $\mathcal{E}$ and $E_1, \cdots, E_n$ be the corresponding equivalence relations. We define $\hat{T}, \hat{E}, \boldsymbol{T}$ as before.

Inputs: $G = (G_e, G_d), W$ where $W, G_e, G_d \subseteq \boldsymbol{T} \times \boldsymbol{T}$.
Output: One of $\bot$, $\emptyset$ or W.
Correctness condition:

- The output is $\bot$ iff $\Phi_G = \emptyset$.
- The output is W iff $\Phi_G \neq \emptyset$ and $W \subseteq \mathcal{DC}(G)$.
- The output is $\emptyset$ iff $\Phi_G \neq \emptyset$ and $W \nsubseteq \mathcal{DC}(G)$.

3.4 Correspondence Between Theory and Code

This theory is all well and good, but GHC defines the interface to a type checker plugin. Does that interface correspond with our theory? We explore the relationship here.

A GHC type checker plugin is essentially one function, *tcPluginSolve*. We (1) introduce this function, (2) explain how GHC's constraint solver uses it and (3) connect this function to the PluginSolve correctness condition.

3.4.1 Type Checker Plugin Interface

A type checker plugin writer implements a small constraint solver via the function *tcPluginSolve*. It is packed into a *TcPlugin* package, with the following type signature.[7]

```
data TcPluginResult = Problem [Ct]
                    | Ok [Ct] [Ct]
data TcPlugin = ∀s. TcPlugin
  { tcPluginInit  :: TcPluginM s
  , tcPluginSolve :: s → [Ct] → [Ct] → [Ct]
                  → TcPluginM TcPluginResult
  , tcPluginStop  :: s → TcPluginM () }
```

The *Ct* type represents constraints. For our purposes, these will either be equalities or disequalities.

The monad *TcPluginM* allows for looking up information from the environment and wraps *IO*. A plugin writer chooses the instantiation for the existential variable *s* (for "state") and can save custom information there. Because *TcPluginM* wraps the *IO* monad, a plugin writer can use, e.g., *IORef*s to store information that needs to be updated between runs of the solver. This state can be initialized in *tcPluginInit*, which is called before any solving is done.

The first [*Ct*] in the type of *tcPluginSolve* is the set of given constraints and the next two [*Ct*]s are wanted constraints.[8] The output *Problem xs* means that the given constraints are contradictory; *xs* is a subsequence of the input list of given constraints that is contradictory on its own. The output *Ok ys zs* indicates that the subsequence of wanted constraints *ys* can be deduced from the input list of given constraints. The second list of constraints *zs* are new constraints for GHC to consider in its own solver algorithm. We will not make use of these, but see Section 4.3 for more discussion.

Now, with a basic understanding of this function, when does GHC's solver call this?

3.4.2 GHC's Solver

At a broad level, GHC walks through user-written source code and generates an implication tree of constraints. The interior nodes are givens (these correspond to places in the source code where GHC learns assumptions, such as a GADT pattern-match or a function with a type signature), and the leaf nodes are wanteds. Recall *getPrice* from Section 2.2.1:

```
getPrice :: Has m "price" Int ⇒ Record m → Int
getPrice (AddField rec fld val) =
  case scomp fld (SSym @"price") of
    Refl    → val
    DisRefl → getPrice rec
```

[7] We have made several simplifications throughout for readability.
[8] The second argument is actually a list of *derived* constraints. The difference between these constraints and wanted constraints is a technical detail, a full coverage of which would derail our discussion.

Consider the type checker state when checking the recursive *getPrice rec* call. There are three[9] givens, stored in different interior nodes in this branch of the implication tree: the *Has m* `"price"` *Int* assumption from the type signature, the *AddField m1 m field valty* constraint from the *AddField* pattern-match, and the *field* ~ `"price"` fact from the *DisRefl* pattern-match. Under all these interior nodes is the leaf wanted constraint *Has m1* `"price"` *Int*, required in order to call the *getPrice* function.

GHC's constraint solver traverses this constraint tree and solves what it can. After doing its own solving, if there are unsolved wanteds, GHC traces every downward path from the root to a node of wanted constraints accumulating givens along the way and then calls *tcPluginSolve* with its accumulated given constraints and the wanted constraints at the destination node.

3.4.3 Correspondence Between *tcPluginSolve* and PluginSolve

Input The constraints we handle are of one of two forms. Either we have an equality constraint or a disequality constraint. The constraints in the input to *tcPluginSolve* have the type *Ct*. The details of that type are not germane here, but it is easy to extract the nature of a constraint from a *Ct*, classifying it into either an equality or disequality constraint.

Once we do this, we are left with two types, ty_1 and ty_2. These correspond to our abstract data set $\boldsymbol{T}$.

Now that we can break down constraints we can connect the inputs of this function to our abstract correctness condition. Considering a call to *tcPluginSolve state xs ys zs*, we can say the first input list *xs* :: [*Ct*] corresponds to the given pair $G = (G_e, G_d)$ where

- every constraint $e \in xs$ that is an equality between ty_1 and ty_2 corresponds to a pair $(ty_1, ty_2) \in G_e$, and
- every constraint $d \in xs$ that is a disequalities between ty_1 and ty_2 corresponds to a pair $(ty_1, ty_2) \in G_d$.

Call the sublist of *xs* that are either equalities or disequalities *xs'*, and let *ws* = *ys* ++ *zs*. Then, *ws* is the list of wanted constraints. For every $e \in ws$ is classified as an equality corresponds to a pair $(ty_1, ty_2) \in W$. Call the sublist of *ws* that are equality or disequality constraints *ws'*.

Output Now, we can easily connect the output *TcPluginResult* to the output of our abstract solver PluginSolve. For our purposes, *tcPluginSolve* should either return *Problem xs*, *Ok ws'* [ ] or *Ok* [ ] [ ]. The output *Problem xs* corresponds to $\bot$. (We have not identified a minimal set of contradictory givens, just returning them all.) In our case, this would mean there are no models of the abstract data we are dealing with, $\boldsymbol{T}$, that are consistent with our givens. The output *Ok ws'* [ ] corresponds with the output W, and the output *Ok* [ ] [ ] corresponds with the output $\emptyset$.

[9] The given from the *Refl* pattern match is not in this branch of the tree.

Now we have a correctness condition for *tcPluginSolve state xs ys zs*: it should return either *Problem xs*, *Ok* [ ] [ ], or *Ok ws'* [ ] where *ws'* is the set of relevant constraints from the list *ys* ++ *zs*. The choice of return value is as specified under the definition of PluginSolve.

4 Thoralf: Building a Generic and Extensible Plugin with SMT

We have developed a deep understanding of the type-equality constraint solving problem. We apply this understanding to develop a plugin that uses an SMT solver in order to check whether a particular wanted is in the deductive closure of a set of givens.

SMT stands for "satisfiability modulo theory". An SMT solver, at its core, tries to find a model—that is, a concrete instantiation for unknowns—that is consistent with a set of assertions. Different SMT solvers support different theories on top of this core, where they can reason about, for example, numbers, strings, or (in our case) arrays. In our work toward Thoralf, we used the Z3 solver [3].

4.1 Examples

4.1.1 Natural Number Arithmetic

Suppose we wanted *stripPrefix* for length-indexed vectors:

```
stripPrefix :: Eq a ⇒
  Vec n a → Vec m a → Maybe (Vec (m − n) a)
stripPrefix VNil ys = Just ys
stripPrefix _ VNil =  Nothing
stripPrefix (x :> xs) (y :> ys) =
  if x == y then (stripPrefix xs ys) else Nothing
```

Without a plugin, we get a type error:

```
* Could not deduce: (n2 - n1) ~ (m - n)
  from the context: n ~ (1 + n1)
  or from: m ~ (1 + n2)
```

From our work in the previous section, we know this is just an equality problem for some givens G and wanteds W:

- $G = (\{(n, 1 + n1), (m, 1 + n2)\}, \emptyset)$
- $W = \{(n2 - n1, m - n)\}$

We encode this problem for an SMT solver like this:

```
(declare-const n Int)
(declare-const m Int)
(declare-const n1 Int)
(declare-const n2 Int)

; Assert all givens.
(assert (= n (+ 1 n1)))
(assert (= m (+ 1 n2)))
(check-sat) ; check if givens are consistent

; Assert at least one wanted is false.
(assert (not (= (- n2 n1) (- m n))))
(check-sat) ; we want "unsat"
```

First, we introduce the type variables in the constraints as constants to the solver. We then introduce the givens to the solver via the `assert` directive.[10] Then, following Diatchki [5], we assert the *negation* of our wanted and seek an `unsat` result. An SMT solver's main purpose is to find *some* model that supports all the facts in its database. If we asserted the unnegated wanted and ran `(check-sat)`, the solver would see if the wanted is consistent with the givens. This is not good enough—we need to verify that the wanted is *entailed* by the givens. The way to see this is to ensure (1) that the givens themselves have at least one model, and (2) that no model exists containing the givens and the negation of the wanted. This corresponds directly with Definition 3.5, where we seek to ensure that *all* models of the givens contain the wanteds, not just some of them.

To be more formal, the solver is proving here that

$$\neg\,(\exists n, m, n1, n2 \mid (n = 1 + n1) \wedge (m = 1 + n2) \wedge (n2 - n1 \neq m - n))$$

A little algebra reduces this to

$$\forall n, m, n1, n2\ \neg(n = 1 + n1 \wedge m = 1 + n2) \vee (n2 - n1 = m - n)$$

$$\forall n, m, n1, n2\ (n = 1 + n1 \wedge m = 1 + n2) \Rightarrow n2 - n1 = m - n$$

which is exactly what we wanted.

4.1.2 Finite Maps

The last example was simple enough that we did not notice we were *encoding* a Haskell type into a SMT expression. In this example we revisit *getPrice*:

```
getPrice (AddField rec fld val) =
  case scomp fld (SSym @"price") of
    DisRefl → getPrice rec
```

Suppose that *rec* :: *Record m1*. We want to deduce the wanted equality $m1 \sim Alter\ m1$ `"price"` *Int* from the given equalities $m \sim Alter\ m1\ field\ val$ (learned from the pattern-match on *AddField*), $m \sim Alter\ m$ `"price"` *Int* (learned from the type signature of *getPrice*), and `"price"` $\nsim$ *field* (learned from the pattern-match on *DisRefl*). Translated into our formal specification, we have givens and a wanted set $G = (G_e, G_d), W$ where

$$G_d = \{(\texttt{"price"},\ field)\}$$

$$G_e = \{(m,\ Alter\ m\ \texttt{"price"}\ Int)\ (m,\ Alter\ m1\ field\ val)\}$$

$$W = \{(m1,\ Alter\ m1\ \texttt{"price"}\ Int\}$$

We translate this as follows.

```
(declare-datatypes (T)
  ((Maybe nothing (just (fromJust T)))))

(declare-const m (Array String (Maybe String)))
(declare-const m1 (Array String (Maybe String)))
(declare-const field String)
(declare-const val String)

; Assert Givens
(assert (not (= "price" field)))
(assert (= m (store m "price" (just "Int"))))
(assert (= m (store m1 field (just val))))
(check-sat)

; Assert at least one wanted is false.
(assert (not (= m1 (store m1 "price" (just "Int")))))
(check-sat)
```

As before, we get the desired result of `unsat`. Our plugin thus returns W, which resolves the type error.

Encoding The only difference between this example and the last is that the finite maps are encoded according to the theory of arrays in the SMT solver. The standard theory of arrays [11, 15] describes an array as a *total* mapping from keys to values that supports the `select` and `store` operations. However, we wish to model *finite*, partial maps from keys to values. We thus use a sturdy, well-worn trick: we map our desired keys to optional (*Maybe*) values. Accordingly, we translate abstract data like *Alter m1* `"price"` *Int*) into `(store m1 "price" (just "Int"))` and use the constant array that maps all keys to `nothing` as an empty finite map.

While this translation is straightforward, the fact that Thoralf works with finite maps shows that it supports some level of translation from the desired theory (finite maps) to one supported by the solver (arrays).

The Encoding Property While we can see at a high level how this translation is working, it is helpful to have a formal treatment of the interaction with the SMT solver.

Let S be the set of well-formed SMT expressions, and let $\cong \subseteq S \times S$ be the SMT solver's equivalence relation. As usual, we denote the set of abstract data with $\boldsymbol{T}$—this includes keys, values and finite maps; $\hat{E}$ is our equivalence relation on these types (i.e., the union of our equality relations of the keys, values and finite maps). Our encoding is a function is $f : \boldsymbol{T} \to S$. The *encoding property* states that

$$\forall t, t' \in \boldsymbol{T}, (t, t') \in \hat{E} \iff f(t) \cong f(t').$$

Our encoding of finite maps indeed satisfies this property, by appeal to the similarity of the theory of finite maps and the theory of arrays.

4.2 Extending Thoralf

If we look back at these two examples, we see that the overall structure is remarkably similar: Thoralf asserts the givens and the negation of the wanted, and then checks for a model. In fact, the *only* difference between natural number arithmetic and finite maps is the encoding function (f, above). It is thus remarkably easy to extend Thoralf into new domains: just provide an encoding function, and off you go.

[10] This is SMT-LIB syntax, which works with a range of solvers. See http://smtlib.cs.uiowa.edu/.

Encoding functions have to translate from abstract data—which are encoded as types within GHC—to SMT expressions. Morally, an encoding function looks like this:

```
encode :: Type → Maybe SExpr
```

Here, *Type* is not the kind of types with values (formerly known only as ⋆), but an ordinary datatype declared in the GHC API that forms GHC's internal AST for types. A *SExpr* is a representation of a s-expression to be given to an SMT solver. The *encode* function is partial because a given encoding will handle only certain constraints for one type index.

In the real implementation, however, the type of *encode* is not this simple. The *Type encode* is passed is a representation of abstract data. This data might have other translatable abstract data inside it. Take, for example, a finite map from *Symbol*s to natural numbers, or even to other finite maps. So, as *encode* is translating to an *SExpr*, it will need to recur on these inner pieces. Because the inner pieces may belong to *different* theories, it is awkward to make this function both recursive and extensible. Thus, the real version of *encode* returns a list of sub-components (*Type*s) that need to be converted, and a continuation function that takes the converted data (*SExpr*s, now) and builds a *SExpr* from these pieces. Further, when declaring variables in SMT, we need to determine the sort of those variables. For this, we have an analogous *encodeKind* :: *Kind* → *Maybe SExpr* function that returns a list of *Kind*s and a continuation.

4.3 Type Variables

When translating data from GHC's representation to an *SExpr*, we have to also translate variables that occur in the data. The theory we worked out in Section 3 includes the possibility of skolem variables, but we studiously left out the possibility of dealing with unification variables. We explain that design decision here.

To recap: a skolem variable is a rigid, abstract variable. It equals no other type (except by explicit assumption from, say, a *Refl* pattern match). If we write

```
length :: [a] → Int
length [True, False] = 2
```

our program is rejected because the skolem *a* is not *Bool*.

In contrast, a unification variable is a placeholder for some other not-yet-known type. When we apply the function *length*, we do not yet know what type *a* should become. If we say *length* ['x'], *a* becomes *Char*. GHC models this uncertainty by instantiating the *a* with a unification variable, often written with a Greek letter such as α. Then, as type checking proceeds, GHC can learn what α should really be and set it accordingly. (Recall that unification variables are implemented with mutable cells, under the hood.)

In practice, both skolem variables and unification variables may appear in the abstract data encoded in GHC's types. However, Thoralf's approach will not work with unification variables.

Essentially, unification with SMT solvers is difficult. Recall that PluginSolve is looking to see whether the wanted constraint holds in *all* possible models consistent with the givens. For skolem variables (translated into the SMT solver using the variable's unique identifier and making it an uninterpreted function), this is the correct behavior. However, for unification variables, type checking must end with a concrete assignment for the variable. While an SMT solver can furnish this through the model it builds, the model is arbitrary, leading unification variable choices to be capricious. Diatchki [5, Section 4.6], describes a possible way forward here, but it requires, in the general case, $O(n^2)$ calls to the solver for n unification variables, and so we have not implemented this idea.

Given that unification variables may appear in GHC's types, does this limitation mean we are unexpressive? Happily, no. Though we have not proved it formally, we conjecture that every construction that yields unification variables can be rewritten to avoid them. In an application *length* [...], a unification variable arises, as we need to know the element type of the list. However, a simple change to *length*'s type avoids this. We can declare *length* :: *IsList x* ⇒ *x* → *Int*, where *IsList x* holds for any *x* that is a list type. Now, when applying *length*, we simply have to infer the type of its argument and use that for *x*—a much simpler process than needing to find *a*, which lives under a type constructor.

Perhaps *length* was easy; let us try *map* :: (*a* → *b*) → [*a*] → [*b*]. This case is indeed harder, but it still succumbs to this general trick, if we write

```
map :: (IsFunc f, EltType list1 ~ ArgType f
       , list2 ~ [ResType f]) ⇒ f → list1 → list2
```

Once again, we simply have to infer the types of *map*'s arguments and then solve constraints. There is no need to create unification variables that might live under type constructors.

Therefore, our task now is to use this technique to design an API for a given theory we wish Thoralf to consider that avoids unification variables. The API in Figure 1 is just such an API, where all the statements about finite maps are expressed as constraints, not, say, as type families that could occur in the middle of types, giving rise to unification variables.

4.4 Finite Maps

We now look at the details of how the finite maps are encoded as an example of how to extend Thoralf with a new theory.

```
data Fm (k :: Type) (v :: Type) :: Type where { }
type family Nil :: Fm k v where { }
type family Alter (m :: Fm k v) (key :: k) (val :: v)
  :: Fm k v where { }
```

```
type family Delete (m :: Fm k v) (key :: k)
  :: Fm k v where { }
```

The *Fm* type is an empty datatype, and all its inhabitants are empty closed type families, following the design criterion of Section 3.2. The library offers three primitive operations on maps: *Nil* creates the empty map, *Alter* changes or adds a new entry to the map, and *Delete* removes an entry (if it exists). While *Nil* is exported to clients, the other type families are not, as they can induce the plugin to encounter a unification variable. For example, consider if we had declared the *AddField* constructor of our *Record* like this:

```
data Record :: Fm Symbol Type → Type where
  AddField :: Record r → SSymbol field → valty
           → Record (Alter r field valty)
```

A use of *AddField* could now have a unification variable standing in for *r* that the solver could not handle.

Instead, we export constraints such as *Has*:

```
type Has (m :: Fm k v) (key :: k) (val :: v)
  = (Alter m key val ~ m)
```

A use of *Has* asserts that adding a new entry into *m* will not change it—in other words, *m* already must have that entry. Because this constraint will not be decomposed, we can be sure its use will not induce unification variables.

Therefore, we have two design principles:

- Represent the type with abstract closed type families.
- Only export *constraints* and *constants*.

4.5 Why SMT: Limitations and Advantages

We consider here the limitations and advantages of our choice to use SMT over a custom solver.

The chief advantage, of course, is that we do not have to write our own solver. In our experience, a critical application of type checker plugins is to support finite maps, the need for which has come up several times in unrelated projects. Given that Z3 supports the theory of arrays [4], it seems redundant to write our own solver. Relatedly, by taking advantage of the expertise that has gone into creating and optimizing the Z3 solver, we could hope to be confident that the solver is efficient and correct.

Modern SMT solvers support many theories: integers, bit-vectors, datatypes, etc. With all these theories, a key benefit of using SMT is *composibility*. Finite maps are parameterized, and we want the value type to range over any type. This means we might have a finite map as the value type, or some other type with an SMT-supported theory. Translating the equality problem into SMT leverages the composibility built into the theories of SMT solvers.

Not all is rosy, however. GHC's type language is much richer than the language that SMT-LIB supports, which lacks, for example, polymorphism in functions. Solvers also have no ability to perform type inference, which makes polymorphic datatype constants (which are allowed) sometimes difficult to use in practice. SMT solvers' architecture makes working with unification variables nearly impossible. Further, we were surprised to learn that the implementation of Z3 was sometimes erroneous, having witnessed some segmentation faults (among other misbehavior) along the way. We have not tried other solvers, but we learned afresh in this project how taking a dependency can be painful. Lastly, using an SMT solver limits the quality of the type errors[11].

Our hunch is that the future lies in SMT and generic solvers. As these solvers get more and more advanced (and stable) they will surely surpass custom solvers. Right now, however, the call remains close.

5 Related Work

Diatchki's SMT Solver Plugin Our work here builds most directly on that of Diatchki [5]. He described the technique we adopted here of asserting the negation of the wanted and then checking for unsatisfiability. His work also discusses the possibility of *improvement*, wherein a unification variable gets filled in, perhaps only with partial information. Diatchki's approach does not scale, however. Our approach here of eliminating unification variables by design is novel and should have no trouble scaling. Our work focuses more on the *theory* of a type checker plugin and on correctness than on implementation.

Units-of-measure Gundry [9] has also described GHC's plugin interface, focusing more on its integration with GHC's OutsideIn algorithm [16] and on writing his own solver for units-of-measure. Gundry's work does not consider correctness of plugins in the way we do here. We have a proof-of-concept that there is a valid encoding function for Gundry's type-level data and believe Thoralf can subsume his plugin.

Extensible Records There are a solid handful of implementations of heterogeneous maps (of which *Record* is an example), including HMap [14], CTRex, and row-types. These all use closed type families and thus all suffer from the inherent limitations of that approach: closed type families tend to work well on concrete data, but get stuck when polymorphism comes in. By contrast, our work with an SMT solver means that we are not relying on type families and can have more flexible equality relations.

Acknowledgments

The authors thank Kenny Foner for his collaboration during early explorations into this idea and Lennart Augustsson, whose conversation sparked the idea for this work. This material is based upon work supported by the National Science Foundation under Grant No. 1704041.

[11] Though, it seems possible for a plugin to explore which assertions cause issues and reverse engineer helpful error messages.

References

[1] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe Zero-cost Coercions for Haskell. *J. Funct. Program.* 26 (2016), 1–79.

[2] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyon Jones. 2005. Associated Type Synonyms. In *International Conference on Functional Programming (ICFP '05)*. ACM.

[3] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[4] Leonardo de Moura and Nikolaj Bjørner. 2009. Generalized, efficient array decision procedures. In *2009 Formal Methods in Computer-Aided Design*. 45–52.

[5] Iavor S. Diatchki. 2015. Improving Haskell Types with SMT. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM.

[6] Richard A. Eisenberg and Jan Stolarek. 2014. Promoting Functions to Type Families in Haskell. In *ACM SIGPLAN Haskell Symposium*.

[7] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Principles of Programming Languages (POPL '14)*. ACM.

[8] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *ACM SIGPLAN Haskell Symposium*.

[9] Adam Gundry. 2015. A Typechecker Plugin for Units of Measure: Domain-specific Constraint Solving in GHC Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM.

[10] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. GADTs meet their match. In *International Conference on Functional Programming (ICFP '15)*. ACM.

[11] John McCarthy. 1962. Towards a mathematical science of computation. In *IFIP Congress*. 21–28.

[12] Stefan Monnier and David Haguenauer. 2010. Singleton types here, singleton types there, singleton types everywhere. In *Programming languages meets program verification (PLPV '10)*. ACM.

[13] Takayuki Muranushi and Richard A. Eisenberg. 2014. Experience Report: Type-checking Polymorphic Units for Astrophysics Research in Haskell. In *ACM SIGPLAN Haskell Symposium*.

[14] Atze van der Ploeg, Koen Claessen, and Pablo Buiras. 2016. The Key Monad: Type-safe Unconstrained Dynamic Typing. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA.

[15] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy Levitt. 2001. A decision procedure for an extensional theory of arrays. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 29–37. https://doi.org/10.1109/LICS.2001.932480

[16] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of Functional Programming* 21, 4-5 (Sept. 2011).