# Teaching GHC new tricks: write your own type checker plugins

Gabe Dijkstra

October 11th, 2018

- GHC has various extensions (e.g. *TypeFamilies*, *DataKinds*, *GADTs*) to perform dependently-typed programming
- Dependently-typed programming requires more cleverness from the programmer

- GHC has various extensions (e.g. *TypeFamilies*, *DataKinds*, *GADTs*) to perform dependently-typed programming
- Dependently-typed programming requires more cleverness from the ~~programmer~~ type checker

- GHC has various extensions (e.g. *TypeFamilies*, *DataKinds*, *GADTs*) to perform dependently-typed programming
- Dependently-typed programming requires more cleverness from the ~~programmer~~ type checker

GHC provides us with a plugin interface to extend its type checker's cleverness

# Type-level natural numbers

# Type-level natural numbers

Data type promotion: use data types at the type-level

$$\text{data } Nat = Zero \mid Succ\ Nat$$

$$\begin{aligned}
&\text{type family } Add\ (n :: Nat)\ (m :: Nat) \text{ where} \\
&\quad Add\ Zero\ \qquad m = m \\
&\quad Add\ (Succ\ n)\ m = Succ\ (Add\ n\ m)
\end{aligned}$$

# Type-level natural numbers

Data type promotion: use data types at the type-level

$$\text{data } \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

$$\begin{array}{l}
\text{type family } \mathit{Add}\ (n :: \mathit{Nat})\ (m :: \mathit{Nat}) \text{ where} \\
\quad \mathit{Add}\ \mathit{Zero}\ \qquad\ m = m \\
\quad \mathit{Add}\ (\mathit{Succ}\ n)\ m = \mathit{Succ}\ (\mathit{Add}\ n\ m)
\end{array}$$

(In practice you should use *Nat* from *GHC.TypeLits*)

# Length-indexed lists

$$
\begin{array}{l}
\text{data } \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat} \\
\text{type family } \mathit{Add}\ (n :: \mathit{Nat})\ (m :: \mathit{Nat}) \text{ where} \\
\quad \mathit{Add}\ \mathit{Zero}\ \qquad\ \ m = m \\
\quad \mathit{Add}\ (\mathit{Succ}\ n)\ m = \mathit{Succ}\ (\mathit{Add}\ n\ m) \\
\text{data } \mathit{Vec}\ (a :: *)\ (n :: \mathit{Nat}) \text{ where} \\
\quad \mathit{Nil} :: \mathit{Vec}\ a\ \mathit{Zero} \\
\quad \mathit{Cons} :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Succ}\ n) \\
\mathit{vAppend} :: \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ (\mathit{Add}\ n\ m) \\
\mathit{vAppend}\ \mathit{Nil} \qquad\qquad\ \ \mathit{ys} = \mathit{ys} \\
\mathit{vAppend}\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{ys} = \mathit{Cons}\ x\ (\mathit{vAppend}\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

# Length-indexed lists – equality

Does this typecheck?

- $leftUnit :: Eq\ a \Rightarrow Vec\ a\ n \rightarrow Bool$
  $leftUnit\ xs = vAppend\ Nil\ xs \equiv xs$

- $rightUnit :: Eq\ a \Rightarrow Vec\ a\ n \rightarrow Bool$
  $rightUnit\ xs = vAppend\ xs\ Nil \equiv xs$

# Length-indexed lists – equality

Does this typecheck?

- $leftUnit :: Eq\ a \Rightarrow Vec\ a\ n \rightarrow Bool$
  $leftUnit\ xs = vAppend\ Nil\ xs \equiv xs$

- $rightUnit :: Eq\ a \Rightarrow Vec\ a\ n \rightarrow Bool$
  $rightUnit\ xs = vAppend\ xs\ Nil \equiv xs$

*leftUnit* passes the type checker, *rightUnit* does not:

- Expected type: $Vec\ a\ n$
- Actual type: $Vec\ a\ (Add\ n\ 'Zero)$

# Length-indexed lists – equality

$$
\begin{array}{l}
\text{type family } \mathit{Add}\ (n :: \mathit{Nat})\ (m :: \mathit{Nat}) \text{ where} \\
\quad \mathit{Add}\ \mathit{Zero} \qquad m = m \\
\quad \mathit{Add}\ (\mathit{Succ}\ n)\ m = \mathit{Succ}\ (\mathit{Add}\ n\ m)
\end{array}
$$

- Problem: $\mathit{vAppend}\ \mathit{xs}\ \mathit{Nil} :: \mathit{Vec}\ a\ (\mathit{Add}\ n\ '\mathit{Zero})$
- GHC knows that $\mathit{Add}\ '\mathit{Zero}\ n \sim n$
- GHC *does not* know that $\mathit{Add}\ n\ '\mathit{Zero} \sim n$
- For any *concrete* natural (e.g. $\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})$) everything works
- If we have variables in the first argument, reducing *Add* gets stuck

- Dependently-typed programming is full of these problems
- Sometimes we can cleverly rearrange type-level functions such that type checker sees all the equalities we need
- There is not always such a rearranging
- Type checker should be clever, not programmer

# Type checker plugins

# Type checker plugins – idea

- GHC type checker iteratively
  - generates constraints (e.g. $a \sim b$, $a \sim Int$)
  - tries to solve them
- GHC allows us to extend this process via plugins

# Type checker plugins – idea

- Write Haskell module that exports *plugin* :: *Plugin*
- Use GHC API
- Users can dynamically load type checker plugin:
  - command line option: `-fplugin MyPluginModuleName`
- In this talk: we also depend on Christiaan Baaij's `ghc-tcplugins-extra`

# Type checker plugins – interface

$$
\begin{array}{l}
\text{data } \mathit{Plugin} = \mathit{Plugin} \\
\quad \{ \;\; \ldots \\
\quad , \;\; \mathit{tcPlugin} :: [\mathit{CommandLineOption}] \to \mathit{Maybe}\ \mathit{TcPlugin} \\
\quad \} \\
\text{data } \mathit{TcPlugin} = \forall s. \mathit{TcPlugin} \\
\quad \{ \;\; \mathit{tcPluginInit} \;\; :: \mathit{TcPluginM}\ s \\
\quad , \;\; \mathit{tcPluginSolve} :: s \to \mathit{TcPluginSolver} \\
\quad , \;\; \mathit{tcPluginStop} \;\; :: s \to \mathit{TcPluginM}\ () \\
\quad \} \\
\text{data } \mathit{TcPluginResult} \\
\quad = \mathit{TcPluginContradiction}\ [\mathit{Ct}] \\
\quad \mid \;\; \mathit{TcPluginOk}\ [(\mathit{EvTerm}, \mathit{Ct})]\ [\mathit{Ct}]
\end{array}
$$

$\mathit{Ct}$: type of constraints

# Type checker plugins – interface

- $s$: the type of our type checker plugin's context
- $tcPluginInit :: TcPluginM\ s$
  - Called at start of type checking
  - Example use: fetch information from type checking context, set up SMT solver session, load in DTD
- $tcPluginSolve :: s \rightarrow TcPluginSolver$
  - Called iteratively during constraint solving
  - This where the hard work happens
- $tcPluginStop :: s \rightarrow TcPluginM\ ()$
  - Called at end of type checking
  - Example use: clean up SMT solver session
- *IO* is available in *TcPluginM*

# Type checker plugins – solver interface: input

$$
\begin{aligned}
&\text{type } \mathit{TcPluginSolver} \\
&\quad = [\,Ct\,] \quad \text{-- given constraints} \\
&\quad \to [\,Ct\,] \quad \text{-- derived constraints} \\
&\quad \to [\,Ct\,] \quad \text{-- wanted constraints} \\
&\quad \to \mathit{TcPluginM}\ \mathit{TcPluginResult}
\end{aligned}
$$

- *Given constraints*: stuff we have put before $\Rightarrow$, e.g. ‘Eq a‘
- *Derived constraints*: new given constraints generated from e.g. functional dependencies, superclasses
- *Wanted constraints*: unsolved constraints

In this talk we are only interested in the *wanted* constraints

# Type checker plugins – solver interface: output

$$
\begin{aligned}
&\text{data } \mathit{TcPluginResult} \\
&\quad = \mathit{TcPluginContradiction}\ [\mathit{Ct}] \\
&\quad \mid\ \mathit{TcPluginOk}\ [(\mathit{EvTerm}, \mathit{Ct})]\ [\mathit{Ct}]
\end{aligned}
$$

We give back:

- *TcPluginContradiction*: unsolvable constraints ($\mathit{Vec}\ a\ 2 \sim \mathit{Vec}\ a\ 53$)
- *TcPluginOk*: solved constraints (with *evidence*) along with *new* wanted constraints
- *EvTerm*: type of evidence that a constraint is satisfied
- In this talk: we will tell GHC to just trust us

# Type checker plugins – solver interface: output

$$
\begin{aligned}
&\text{data } \mathit{TcPluginResult} \\
&\quad = \mathit{TcPluginContradiction}\ [\mathit{Ct}] \\
&\quad \mid\ \mathit{TcPluginOk}\ [(\mathit{EvTerm}, \mathit{Ct})]\ [\mathit{Ct}]
\end{aligned}
$$

What if we always give back more constraints than we solve?

- `solveSimpleWanteds: too many iterations (limit = 4)`

# Example

- $leftUnit :: Eq\ a \Rightarrow Vec\ a\ n \rightarrow Bool$
  $leftUnit\ xs = vAppend\ Nil\ xs \equiv xs$

- $rightUnit :: Eq\ a \Rightarrow Vec\ a\ n \rightarrow Bool$
  $rightUnit\ xs = vAppend\ xs\ Nil \equiv xs$

- Expected type: $Vec\ a\ n$
- Actual type: $Vec\ a\ (Add\ n\ 'Zero)$

Goal: teach GHC that $n \sim Add\ n\ 'Zero$

# Setting up

Goal: teach GHC that $n \sim Add\ n\ 'Zero$

- Need to recognise the type family *Add*
- Need to recognise the promoted data constructor *Zero*

$$
\begin{aligned}
&\text{type } PluginCtx = (TyCon, TyCon)\\
&lookupRelevantTyCons :: TcPluginM\ PluginCtx\\
&lookupRelevantTyCons = \text{do}\\
&\quad mdName \leftarrow lookupModule\ (mkModuleName\ \texttt{"Vec"})\\
&\qquad\qquad\qquad\qquad\qquad\quad (fsLit\ \texttt{"tc-plugins"})\\
&\quad add \quad\ \ \leftarrow tcLookupTyCon\\
&\qquad\qquad\quad =\!\!\ll lookupName\ mdName\ (mkTcOcc\ \texttt{"Add"})\\
&\quad zero \quad \leftarrow tcLookupDataCon\\
&\qquad\qquad\quad =\!\!\ll lookupName\ mdName\ (mkDataOcc\ \texttt{"Zero"})\\
&\quad pure\ (add, promoteDataCon\ zero)
\end{aligned}
$$

# Recognising and solving constraints

Goal: teach GHC that $n \sim Add\ n\ 'Zero$

- Need to recognise the correct $Ct$

  type $PluginCtx = (TyCon, TyCon)$

  $solveCts :: PluginCtx \rightarrow TcPluginSolver$
  $solveCts\ ctx\ \_given\ \_derived\ wanteds$
  $\quad = pure\ \$\ TcPluginOk\ (mapMaybe\ (solveCt\ ctx)\ wanteds)\ [\,]$

# Recognising and solving constraints

Goal: teach GHC that $n \sim Add\ n\ 'Zero$

- Need to recognise the correct $Ct$

$$
\begin{array}{l}
solveCt :: PluginCtx \to Ct \to Maybe\ (EvTerm, Ct) \\
solveCt\ (addTc, zeroC)\ ct = \\
\quad \text{case}\ classifyPredType \circ ctEvPred \circ ctEvidence\ \$\ ct\ \text{of} \\
\qquad EqPred\ NomEq\ t1@(TyConApp\ ta\ [t0, TyConApp\ tz\ [\,]])\ t2 \\
\qquad\quad \mid ta \equiv addTc \wedge tz \equiv zeroC \wedge t0\ \text{`}eqType\text{`}\ t2 \\
\qquad\qquad \to Just\ (evByFiat\ \texttt{"tc-plugins"}\ t1\ t2, ct) \\
\qquad \_ \to Nothing
\end{array}
$$

*evByFiat* (from `ghc-tcplugins-extra`): "just trust us"

# Type-level sets

# Type-level sets

We want to define type-level sets such that we have the following:

- a kind $\mathit{TypeSet}$
- for every list of types $\mathit{xs} :: [*]$, $\mathit{Elems}\ \mathit{xs} :: \mathit{TypeSet}$
- for every $x, y :: \mathit{TypeSet}$, $\mathit{Union}\ x\ y :: \mathit{TypeSet}$

They should behave like sets, e.g. we have should have:

- $\mathit{Elems}\ [\mathit{Int}, \mathit{Bool}] \sim \mathit{Elems}\ [\mathit{Bool}, \mathit{Int}]$
- $\mathit{Elems}\ [\mathit{Int}, \mathit{Int}] \sim \mathit{Elems}\ [\mathit{Int}]$
- $\mathit{Union}\ a\ a \sim a$

# Type-level sets – why?

Example use case of type-level sets: tracking effects

$$
\begin{aligned}
&\text{data } \mathit{ReadFile} :: * \\
&\text{data } \mathit{WriteFile} :: * \\
&\text{type } \mathit{NoEffects} = \mathit{Elems}\ '[] \\
&\text{data } \mathit{Action}\ (\mathit{es} :: \mathit{TypeSet})\ (a :: *) \text{ where} \\
&\quad \mathit{Read} :: \mathit{FilePath} \\
&\qquad\quad \to \mathit{Action}\ (\mathit{Elems}\ '[\mathit{ReadFile}])\ \mathit{String} \\
&\quad \mathit{Write} :: \mathit{FilePath} \\
&\qquad\quad \to \mathit{String} \\
&\qquad\quad \to \mathit{Action}\ (\mathit{Elems}\ '[\mathit{WriteFile}])\ () \\
&\quad \mathit{Pure} :: a \to \mathit{Action}\ \mathit{NoEffects}\ a \\
&\quad \mathit{Bind} :: \mathit{Action}\ \mathit{es0}\ a \\
&\qquad\quad \to (a \to \mathit{Action}\ \mathit{es1}\ b) \\
&\qquad\quad \to \mathit{Action}\ (\mathit{Union}\ \mathit{es0}\ \mathit{es1})\ b
\end{aligned}
$$

# Type-level sets – how? (approach 0)

- Define type family *Normalise* that sorts and nubs a list of types
- Define type $Set\ tys = Normalise\ tys$
- Approach taken in `type-level-sets` package by Dominic Orchard and Tomas Petricek
- Pros: no need for GHC type checker plugins, works for concrete lists
- Cons: GHC does not know that $Union\ a\ a \sim a$

# Type-level sets – how? (approach 1)

- Define data type:

$$
\begin{aligned}
&\text{data } \mathit{Set}\ a \\
&\quad = \mathit{Elems}\ [a] \\
&\quad \mid \mathit{Union}\ (\mathit{Set}\ a)\ (\mathit{Set}\ a) \\
&\quad \mid \ \ldots
\end{aligned}
$$

- Use *promoted* version of $Set$ and define type $TypeSet = Set\ *$

However, we do not want $Elems$ to be injective:

$$
\begin{aligned}
&\text{type family } \mathit{Bad}\ a :: * \text{ where} \\
&\quad \mathit{Bad}\ (\mathit{Elems}\ '[\mathit{Bool}, \mathit{Bool}]) = () \\
&\quad \mathit{Bad}\ (\mathit{Elems}\ '[\mathit{Bool}]) \qquad\quad = \mathit{Char}
\end{aligned}
$$

# Type-level sets – how? (approach 2)

- Define "abstract" type:

  data *TypeSet*

- with abstract operations:

  type family *Elems* $(x :: [*]) :: \mathit{TypeSet}$
  type family *Union* $(x :: \mathit{TypeSet})\ (y :: \mathit{TypeSet}) :: \mathit{TypeSet}$

- Use type checker plug-in to implement type-level operations *Elems* and *Union*

# Type-level sets – plugin architecture

- Define data type that represents types we care about

$$\text{data } \mathit{TypeSetExpr} = \ldots$$

- Write function that recognises types we care about:

$$\mathit{toTypeSetExpr} :: \mathit{PluginCtx} \rightarrow \mathit{Type} \rightarrow \mathit{Maybe}\ \mathit{TypeSetExpr}$$

- Implement function that checks whether two expressions represent the same set of types:

$$\mathit{compareTypeSetExpr} :: \mathit{TypeSetExpr} \rightarrow \mathit{TypeSetExpr} \rightarrow \mathit{Bool}$$

# Type-level sets – plugin boilerplate

Set up is similar to before:

```
type PluginCtx = (TyCon, TyCon)
lookupRelevantTyCons :: TcPluginM PluginCtx
lookupRelevantTyCons = do
  md ← lookupModule (mkModuleName "Set")
                    (fsLit "tc-plugins")
  elems ← tcLookupTyCon
    =≪ lookupName md (mkTcOcc "Elems")
  union ← tcLookupTyCon
    =≪ lookupName md (mkTcOcc "Union")
  pure (elems, union)
```

# Type-level sets – plugin boilerplate

Our main solver function looks similar to the one we had before, but this one also deals with contradictory constraints itself:

```haskell
solveCts :: PluginCtx -> TcPluginSolver
solveCts ctx _ _ wanteds = do
  let (proven, incorrect)
      = partitionEithers
      . mapMaybe (solveCt ctx)
      $ wanteds
  if null incorrect
    then pure $ TcPluginOk proven []
    else pure $ TcPluginContradiction incorrect
```

# Type-level sets – plugin boilerplate

Our main solver function looks similar to the one we had before:

```
solveCt :: PluginCtx → Ct → Maybe (Either (EvTerm, Ct) Ct)
solveCt ctx ct =
  case classifyPredType ∘ ctEvPred ∘ ctEvidence $ ct of
    EqPred NomEq t1 t2 →
      case compareTypeSetExpr
          <$> toTypeSetExpr ctx t1
          <*> toTypeSetExpr ctx t2 of
        Just True → pure $ Left (evByFiat "tc-plugins" t1 t2, ct)
        Just False → pure $ Right ct
        Nothing → Nothing
    _ → Nothing
```

# Type-level sets – expression data type

We turn:

$$
\begin{array}{l}
\text{data } \mathit{TypeSet} \\
\text{type family } \mathit{Elems}\ (x :: [*]) :: \mathit{TypeSet} \\
\text{type family } \mathit{Union}\ (x :: \mathit{TypeSet})\ (y :: \mathit{TypeSet}) :: \mathit{TypeSet}
\end{array}
$$

into:

$$
\begin{array}{l}
\text{data } \mathit{TypeSetExpr} \\
\quad = \mathit{ElemsExpr}\ [\mathit{Type}] \\
\quad \mid\ \mathit{UnionExpr}\ \mathit{TypeSetExpr}\ \mathit{TypeSetExpr}
\end{array}
$$

# Type-level sets – recognising expressions

```haskell
toTypeSetExpr :: PluginCtx -> Type -> Maybe TypeSetExpr
toTypeSetExpr ctx@(elemsTf, unionTf) t = case splitTyConApp t of
  (tyCon, args)
      | tyCon == elemsTf -> case args of
        [tyList] -> pure $ ElemsExpr (getTypes tyList)
        _ -> Nothing
      | tyCon == unionTf -> case args of
        [l, r] -> UnionExpr <$> toTypeSetExpr ctx l
                            <*> toTypeSetExpr ctx r
        _ -> Nothing
      | otherwise -> Nothing
```

# Type-level sets – comparing expressions

$$
\begin{aligned}
&\mathit{compareTypeSetExpr} :: \mathit{TypeSetExpr} \to \mathit{TypeSetExpr} \to \mathit{Bool} \\
&\mathit{compareTypeSetExpr} = \mathit{eqTypes} \ \text{`}\mathit{on}\text{`} \ \mathit{normalise} \\
&\quad \text{where} \\
&\qquad \mathit{normalise} :: \mathit{TypeSetExpr} \to [\mathit{Type}] \\
&\qquad \mathit{normalise} = \mathit{nubBy} \ \mathit{eqType} \circ \mathit{sortBy} \ \mathit{nonDetCmpType} \circ \mathit{flatten} \\
&\qquad \mathit{flatten} :: \mathit{TypeSetExpr} \to [\mathit{Type}] \\
&\qquad \mathit{flatten} \ (\mathit{ElemsExpr} \ \mathit{ts}) = \mathit{ts} \\
&\qquad \mathit{flatten} \ (\mathit{UnionExpr} \ l \ r) = \mathit{flatten} \ l +\!\!+ \ \mathit{flatten} \ r
\end{aligned}
$$

# Type-level sets – examples

The following definitions now type check:

$$\text{type } \mathit{NoEffects} = \mathit{Elems}\ '[]$$

$$\begin{array}{l}
\text{data } \mathit{Action}\ (\mathit{effects} :: \mathit{TypeSet})\ (a :: *) \text{ where} \\
\quad \mathit{Read} :: \mathit{FilePath} \rightarrow \mathit{Action}\ (\mathit{Elems}\ '[\mathit{ReadFile}])\ \mathit{String} \\
\quad \mathit{Write} :: \mathit{FilePath} \rightarrow \mathit{String} \rightarrow \mathit{Action}\ (\mathit{Elems}\ '[\mathit{WriteFile}])\ () \\
\quad \mathit{Pure} :: a \rightarrow \mathit{Action}\ \mathit{NoEffects}\ a \\
\quad \mathit{Bind} :: \mathit{Action}\ \mathit{effects0}\ a \\
\qquad \rightarrow (a \rightarrow \mathit{Action}\ \mathit{effects1}\ b) \\
\qquad \rightarrow \mathit{Action}\ (\mathit{Union}\ \mathit{effects0}\ \mathit{effects1})\ b
\end{array}$$

$$\begin{array}{l}
\mathit{testAction0} :: \mathit{Action}\ \mathit{NoEffects}\ \mathit{Int} \\
\mathit{testAction0} = (\mathit{Pure}\ 1)\ `\mathit{Bind}`\ (\lambda x \rightarrow \mathit{Pure}\ (10 * x))
\end{array}$$

$$\begin{array}{l}
\mathit{testAction1} :: \mathit{Action}\ (\mathit{Elems}\ '[\mathit{WriteFile}, \mathit{ReadFile}])\ () \\
\mathit{testAction1} = \mathit{Read}\ \texttt{"in.txt"}\ `\mathit{Bind}`\ \mathit{Write}\ \texttt{"out.txt"}
\end{array}$$

# Type-level sets – what's missing?

We can add variables to *TypeSetExpr* to support equalities involving type variables, e.g.:

- $Union\ a\ a \sim a$
- $Union\ a\ b \sim Union\ b\ a$

To do so, we would have to:

- Add a constructor *VarExpr* to *TypeSetExpr* (easy)
- Update *toTypeSetExpr* to recognise type variables (easy)
- Update *compareTypeSetExpr* to recognise the above equalities (involved)

# Type-level sets – general architecture

To implement a type-level gadget like type-level sets, we can define:

- In module: An abstract data type (no constructors)
- In module: Abstract type families (no clauses)
- In plugin: function that recognises type expressions that involve our type-level gadget
- In plugin: function that compares two such expressions

# Further reading

- Christiaan Baaij: "GHC type checker plugins: adding new type-level operations"
- Adam Gundry: "A Typechecker Plugin for Units of Measure"
- Richard Eisenberg, Divesh Otwani: "The Thoralf plugin: for your fancy type needs"
- Jan Bracker, Henrik Nilsson: "Supermonads: One Notion to Bind Them All" (talk later today!)

# Final remarks

What we have seen:

- When GHC's type checker is insufficiently clever
- How to extend it by writing a type checker plugin
- That writing such a plugin requires surprisingly little boilerplate

What we have not seen:

- How to write plugins in a principled way

Should you write type checker plugins?

- Yes!
- In practice: most likely not
- It is a lot of fun however!