Towards Impredicative Types for GHC

A deep refactoring of GHC's type system to accommodate impredicative types

ORESTIS MELKONIAN, Utrecht University, The Netherlands

This report documents the results of the "Experimentation Project" part of the Computing Science MSc degree at the Utrecht University, supervised by Alejandro Serrano Mena.

1 BRIEF OVERVIEW

This experimentation project's goal is to lay the foundations, implementation-wise, that will enable the integration of impredicative types in GHC, the de-facto optimizing compiler for Haskell.

There have been many proposed type systems that support impredicative types, such as MLF [Le Botlan and Rémy 2003], FPH [Vytiniotis et al. 2008], Flexible Types [Leijen 2009], Boxy Types [Vytiniotis et al. 2006], HMF [Leijen 2008] and QML [Russo and Vytiniotis 2009]. Alas, these type systems are either too hard to explain to end-users/programmers or too difficult to implement, which renders them inappropriate to integrate in a realistic compiler.

In this work, we choose to implement *Guarded Impredicative Polymorphism* [Serrano et al. 2018], which enjoys a very intuitive declarative specification (GI) that will allow programmers to easily adopt it. Moreover, it plays nicely with type classes [Wadler and Blott 1989], generalized algebraic datatypes (GADTs) [Schrijvers et al. 2009] and type-level functions [Schrijvers et al. 2008], while there is hope that integrating it with other advanced features of Haskell's type system will not pose any significant problems.

We refrain from motivating the reason why impredicative types are useful, but refer the reader to the second section of [Serrano et al. 2018].

2 IMPLEMENTATION IN GHC

In this section, we describe the so-far implemented features, emitting all intermediate solutions that we found insufficient.

- Subsection 2.1 motivates the addition of an extra flag we keep on each unification variable.
- Subsection 2.2 defines the extra constraints we need to deal with.
- Subsection 2.3 gives the solving rules for these constraints and also provides the translation to GHC's intermediate language Core.
- Subsection 2.4 gives the type-checking rules that emit such constraints.

2.1 Meta-variable flags

A necessary first step is to keep a flag on each meta-variable, which indicates whether we can unify that variable with a polymorphic type. We achieve this by adding a new datatype `TcFlavor` and then augmenting the fields of `TcTyVarDetails.MetaTv` with this information. We then refactor GHC's codebase to always put a `Mono` flag when creating meta-variables; `Poly` flags will only appear when we generate constraints in type-checking.

```
data TcFlavor   = Mono | Poly
data TcTyVarDetails = ... | MetaTv { ... , mtv_flavor :: TcFlavor }
```

Author's address: Orestis Melkonian, Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, o.melkonian@uu.nl.

2.2 Constraint Definition

Guarded impredicative polymorphism requires the addition of two new types of constraints:

Instantiation (lhs $\leq$ rhs): lhs can be instantiated to rhs
Generalisation (lhs $\preceq$ rhs): lhs can be generalized and then instantiated to rhs

We first need to define wired-in definitions in `TysWiredIn.hs` for these constraints (`InstOf` and `GenOf`). In the case of `GenOf`, the variables we can generalize are bound in the outer ∀ of the left-hand side. Since we need to keep track of whether these variables are `Mono` or `Poly`, we keep a list of booleans (`Mono` ⟷ `True`, `Poly` ⟷ `False`) on the type level.

```
newtype (b ↜ a)                = InstOf (b → a)
newtype (b ↝ a) (t :: [Bool]) = GenOf  (b → a)
```

While these provide a nice way to test/debug solving of these contraints by providing them in surface syntax, the `GenOf` constraint will later need to hold implications. Unfortunately, we cannot express this as a normal Haskell type, hence we need to define a new type of constraint `CtGen`:

```
data Ct =  ...  | CGenCan CtGen
data CtGen =
  -- Represents (∀as. C ∧ I ⇒ σ₁) ↝ σ₂
  = CtGen
      { gen_ev :: CtEvidence            -- always of type (σ₁ → σ₂), always Wanted
      , gen_tvs  :: [TcTyVar]           -- as
      , gen_wanted :: Cts               -- C
      , gen_implic :: Bag Implication   -- I
      , gen_lhs :: TcType               -- σ₁
      , gen_rhs :: TcType               -- σ₂
      }
```

2.3 Constraint Solving

We define several solving rules for `InstOf` and `GenOf` constraints and integrate them in GHC's constraint solver in `TcCanonical.hs`. Each solving step must be justified by providing evidence in the form of Core expression, which converts expressions of the source constraint/type to expressions of the target constraint/type. It is important to note that the newtype declarations we defined give rise to the following axioms (and their inverse by using `sym`):

- [`Ax↜`]: $\sigma_1 \leq \sigma_2 \iff \sigma_1 \rightarrow \sigma_2$
- [`Ax↝`]: $\sigma_1 \preceq \sigma_2\,'[flags...] \iff \sigma_1 \rightarrow \sigma_2$

For convenient notation, μ and σ represents monomorphic and polymorphic variables, respectively. The rules for instantiation constraints are the following:

$$[\text{INST}\epsilon]\ \frac{\overbrace{\mu \sim \eta}^{\texttt{co :: Coercion}}}{\mu \leq \eta}$$

<u>evidence</u>: `(λ(x::μ). μ ▷ co) ▷ (sym Ax↜)`

$$[\text{INST}\forall\ \text{L}]\ \dfrac{\overbrace{\text{fresh } \alpha}^{\texttt{tys :: [TcTyVar]}} \qquad \overbrace{Q[\alpha/a]}^{\texttt{qs :: [EvTerm]}} \qquad \overbrace{\sigma[\alpha/a] \leq \eta}^{\texttt{innerEv :: EvId}}}{(\forall a.Q \Rightarrow \sigma) \leq \eta}$$

evidence: `(λ(x :: forall a. Q ⇒ σ). (innerEv ▷ Ax⇜) (x @tys @qs)) ▷ (sym Ax⇜)`

The rules for generalisation constraints are the following:

$$[\text{GEN\_INIT}]\ \dfrac{\overbrace{\text{fresh } \alpha}^{\texttt{tys :: [TcTyVar]}} \qquad \overbrace{(\forall a. \quad \overbrace{Q[\alpha/a]}^{\texttt{qs :: [EvTerm]}} \quad \Rightarrow \sigma_1[\alpha/a]) \leq \sigma_2}^{\texttt{innerEv :: EvId}}}{(\forall a.Q \Rightarrow \sigma_1) \preceq_{\text{surface}} \sigma_2\ '[flags]}$$

evidence: `(λ(x :: forall a. Q ⇒ σ₁). innerEv (x @tys @qs) ) ▷ (sym Ax⇝)`

$$[\text{GEN}\forall\text{L}]\ \dfrac{\overbrace{\sigma \leq \eta}^{\texttt{innerEv :: EvId}}}{\sigma \leq \eta}$$

evidence: `λ(x :: σ). (innerEv ▷ (sym Ax⇜)) x`

$$[\text{GEN}\forall\text{R}]\ \dfrac{\overbrace{\overbrace{\forall a.}^{\texttt{tys :: [TcTyVar]}} \quad (\quad \overbrace{Q}^{\texttt{qs :: [EvId]}} \quad \supset \quad \overbrace{g \leq \mu}^{\texttt{innerEv :: EvId}} \quad)}^{\texttt{bnds :: [EvBind]}}}{g \leq (\forall a.Q \Rightarrow \mu)}$$

evidence: `λ(x :: g). λ(tys, qs). let bnds in innerEv x`

2.4 Constraint Generation

With instantiation and generalisation constraints in place, we can now define the type-checking rules for the main Haskell constructs. As an initial implementation, we replace functions `tcApp`, `tcFun` and `tcArg` in `TcExpr.hs` as follows:

$$[\text{APP}]\ \dfrac{\overbrace{a_i}^{\text{fresh and poly}} \qquad \Gamma \vdash^{\text{arg}} e_i : a_i \rightsquigarrow e_i' \qquad \Gamma \vdash^{\text{fun}} f : \sigma \rightsquigarrow f' \qquad \text{emit } \{\sigma \leq a_1 \to \cdots \to a_n \to \beta\} \rightsquigarrow ev}{\Gamma\ \vdash\ f\ e_1 \ldots e_n : \beta \rightsquigarrow (ev\ f')\ e_1' \ldots e_n'}$$

$$[\text{FUN\_VAR}]\ \dfrac{x : \sigma \in \Gamma}{\Gamma \vdash^{\text{fun}} x : \sigma \rightsquigarrow \emptyset}$$

$$[\text{FUN\_HEAD}]\frac{\neg\text{var}(f) \qquad \Gamma \vdash f : \mu \rightsquigarrow Q}{\Gamma \vdash^{\text{fun}} f : \mu \rightsquigarrow Q}$$

$$[\text{ARG}]\frac{\Gamma \vdash e' : \sigma \rightsquigarrow Q \qquad v = \text{fuv}(\sigma, Q) - \text{fuv}(\Gamma) \qquad \text{emit } \{(\forall v.Q \Rightarrow \sigma) \leq \alpha\} \rightsquigarrow ev}{\Gamma \vdash^{\text{arg}} e : \alpha \rightsquigarrow (ev\ e')}$$

NOTE. The constraint generation is still incomplete; there are several implementation holes with the most significant being the extension of the `InertCans` datatype to also hold generalisation constraints (i.e. `inert_ctgen :: [CtGen]`) and handle them. To check other things that are incomplete grep for `T0D0` in all `.hs` files (notice the use of zeros instead of letters!!).

3 NEXT STEPS

While there is still a lot of work to be done to fully integrate impredicative types in GHC, more so with the advent of linear types. Nonetheless, there are a handful of immediate next steps that will greatly simplify the type-checking part of GHC.

Remove $ rule. A nice property of GI is that there is no need to have a separate rule for function application using $, since it is equivalent to the APP rule. Hence, a viable next step to simplify the inner workings of GHC is to remove this special rule.

Remove **`tcSubtype`**. While GHC did not previously support ≤ constraints, it still uses the notion of subtyping but without deferring any instantiation constraints. The subtyping behaviour manifests itself with calls to the `tcSubtype` function inside the compiler, thus it would be reasonable to remove it completely and replace it with emissions of ≤ constraints.

Support for Visible Type Application (VTA) [Eisenberg et al. 2016]. GHC currently handles VTA on the spot when type-checking a function application, but we now have to defer the decision for later (i.e. emit a delayed VTA constraint).

4 CONCLUSION

We have presented the current implementation efforts to integrate impredicative types in GHC's type checker and constraint solver. The code is publicly available in a Github fork of GHC[1], which we tried to keep in sync with the master branch of GHC[2]. We hope that this work will enable future experimentation and eventually lead to stable support for impredicative types in Haskell!

REFERENCES

Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. 2016. Visible type application. In *European Symposium on Programming Languages and Systems*. Springer, 229–254.

Didier Le Botlan and Didier Rémy. 2003. ML F: raising ML to the power of system F. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 27–38.

Daan Leijen. 2008. HMF: Simple type inference for first-class polymorphism. In *ACM Sigplan Notices*, Vol. 43. ACM, 283–294.

Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 66–77.

Claudio V Russo and Dimitrios Vytiniotis. 2009. QML: Explicit first-class polymorphism for ML. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*. ACM, 3–14.

[1] https://github.com/omelkonian/ghc-fork/commits/ore

[2] https://github.com/ghc/ghc

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type Checking with Open Type Functions. *SIGPLAN Not.* 43, 9 (Sept. 2008), 51–62. https://doi.org/10.1145/1411203.1411215

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and decidable type inference for GADTs. In *ACM Sigplan Notices*, Vol. 44. ACM, 341–352.

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 783–796.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 251–262.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class polymorphism for Haskell. *ACM Sigplan Notices* 43, 9 (2008), 295–306.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 60–76.