Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Constraint Handling Rules with Binders, Patterns and Generic Quantification

Alejandro Serrano Mena and **Jurriaan Hage**

Department of Information and Computing Sciences, Universiteit Utrecht
J.Hage@uu.nl

August 31, 2017

Overview

- What are we working on?
- Where do CHRs arise in our work?
- Why do they fall short?
- What do we need to make them work (better)?
- For the technical development: see the paper

The Haskell type system

- Haskell is a hotbed of type system innovation: Generalized Abstract Data Types (GADTs), type classes, (closed and open) type families, etc.
- Inspired by developments in the dependently type community, more and more features are added: power but also complexity is on the rise
  - Only rarely does the type system become a bit simpler

Pros and Cons

- New type system features have advantages:
  - More properties of a program can be encoded in the type system,
  - More guarantees on the correctness of our program, checked automatically at every compile
- But also disadvantages:
  - Difficult to implement
  - New features may be difficult to grasp
  - Complicated interactions
  - Introduces new ways for programmers to make mistakes

What is type error diagnosis?

- Type error diagnosis is the problem of communicating to the programmer that and/or why a program is not type correct
- This may involve information
  - that a program is type incorrect
  - which inconsistency was detected
  - which parts of the program contributed to the inconsistency
  - how the inconsistency may be fixed
- Traditionally, functional languages have more room for inconsistencies $\Rightarrow$ at least some attention was paid to type error diagnosis

A small mistake with big consequences

$$
\begin{aligned}
pExpr = \; & pAndPrioExpr \\
\langle|\rangle \quad & sem\_Expr\_Lam \quad \text{-- Semantics for lambda expressions} \\
& \langle\$\ pKey\ \texttt{"\textbackslash\textbackslash"} \\
& \langle*\rangle pFoldr1\ (sem\_LamIds\_Cons, sem\_LamIds\_Nil)\ pVarid \\
& \langle*\rangle pKey\ \texttt{"->"} \\
& \langle*\rangle pExpr
\end{aligned}
$$

The error message that results:

```
ERROR "BigTypeError.hs":1 - Type error in application
*** Expression     : sem_Expr_Lam <$ pKey "\\" <*> pFoldr1 (sem_LamIds_Cons,sem_
LamIds_Nil) pVarid <*> pKey "->"
*** Term           : sem_Expr_Lam <$ pKey "\\" <*> pFoldr1 (sem_LamIds_Cons,sem_
LamIds_Nil) pVarid
*** Type           : [Token] -> [((Type -> Int -> [([Char],(Type,Int,Int))] -> I
nt -> Int -> [(Int,(Bool,Int))] -> (PP_Doc,Type,a,b,[c] -> [Level],[S] -> [S]))
-> Type -> d -> [([Char],(Type,Int,Int))] -> Int -> Int -> e -> (PP_Doc,Type,a,b
,f -> f,[S] -> [S]),[Token])]
*** Does not match : [Token] -> [([Char] -> Type -> d -> [([Char],(Type,Int,Int)
)] -> Int -> Int -> e -> (PP_Doc,Type,a,b,f -> f,[S] -> [S]),[Token])]
```

A small mistake with big consequences

$$
\begin{aligned}
pExpr = \; & pAndPrioExpr \\
\langle|\rangle \quad & sem\_Expr\_Lam \quad \text{-- Semantics for lambda expressions} \\
& \langle\$\ pKey\ \texttt{"\textbackslash\textbackslash"} \\
& \langle*\rangle pFoldr1\ (sem\_LamIds\_Cons, sem\_LamIds\_Nil)\ pVarid \\
& \langle*\ pKey\ \texttt{"->"} \\
& \langle*\rangle pExpr
\end{aligned}
$$

The error message that results:

```
ERROR "BigTypeError.hs":1 - Type error in application
*** Expression     : sem_Expr_Lam <$ pKey "\\" <*> pFoldr1 (sem_LamIds_Cons,sem_
LamIds_Nil) pVarid <*> pKey "->"
*** Term           : sem_Expr_Lam <$ pKey "\\" <*> pFoldr1 (sem_LamIds_Cons,sem_
LamIds_Nil) pVarid
*** Type           : [Token] -> [((Type -> Int -> [([Char],(Type,Int,Int))] -> I
nt -> Int -> [(Int,(Bool,Int))] -> (PP_Doc,Type,a,b,[c] -> [Level],[S] -> [S]))
-> Type -> d -> [([Char],(Type,Int,Int))] -> Int -> Int -> e -> (PP_Doc,Type,a,b
,f -> f,[S] -> [S]),[Token])]
*** Does not match : [Token] -> [([Char] -> Type -> d -> [([Char],(Type,Int,Int)
)] -> Int -> Int -> e -> (PP_Doc,Type,a,b,f -> f,[S] -> [S]),[Token])]
```

What is a Domain Specific Language (DSL)?

- According to Walid Taha we deal with a DSL:
  - the domain is well-defined and central
  - the notation is clear,
  - the informal meaning is clear,
  - the formal meaning is clear and implemented.

What is a Domain Specific Language (DSL)?

- According to Walid Taha we deal with a DSL:
  - the domain is well-defined and central
  - the notation is clear,
  - the informal meaning is clear,
  - the formal meaning is clear and implemented.
- Missing is:
  - and an implementation of the DSL can communicate with the programmer about the program in terms of the domain
- As we say: "domain-abstractions should not leak"

Embedded Domain Specific Languages

- Embedded (internal à la Fowler) Domain Specific Languages are achieved by encoding the DSL syntax inside that of a host language.
- Some (arguable) advantages:
  - familiarity host language syntax
  - escape hatch to the host language
  - existing libraries, compilers, IDE's, etc.
  - combining EDSLs
- At the very least, useful for prototyping DSLs
- According to Hudak "the ultimate abstraction"

What host language?

- Some languages provide extensibility as part of their design, e.g., Ruby, Python, Scheme
- Others are rich enough to encode a DSL with relative ease, e.g., Haskell, C++
- In Haskell, EDSLs are simply libraries that provide some form of "fluency"
  - Consisting of domain terms and types, and special operators with particular priority and fixity

Some history: CHRs for typing Haskell programs

- to improve type error diagnosis [Stuckey et al. 2006, Wazny et al., Serrano and Hage 2016]
- to describe and extend type classes to implement ad-hoc polymorphism [Sulzmann et al. 2007, Dijkstra et al. 2007]
- to generalize the shape of algebraic data types [Sulzmann et al. 2006]

Why we opt for CHRs

- CHRs allow the formulation of advanced type systems in simple enough way
- CHRs can be transformed so that domain-type error diagnosis can be injected
  - to achieve control over the order of solving
    - which often determines when/where solving will fail
  - to carry along domain-specific information to use in error messages
- Some languages features we want to support in Haskell are not easily modeled with CHRs
- Solution: extend CHRs

So what exactly are we missing?

- Functions in Haskell can be polymorphic: *id* :: $\forall$ *a* . *a* -> *a*
- Whenever *id* is used, it must be instantiated, choosing a fresh type variable to replace *a*
- For *id id* we can choose α for the former, and β for the latter, and unification will find that $\alpha = \beta \rightarrow \beta$
- In the Hindley-Milner type system, all instantiations can be made at the start of a binding block, all at once
- However, in a type system that supports higher-ranked instantiation often needs to be delayed

What are higher-rank types?

- Types in which not all universal quantifiers occur at top level
  - $id :: \forall\ a\ .\ a \rightarrow a$ is not higher-ranked
  - $gimmeid :: (\forall\ a\ .\ a \rightarrow a) \rightarrow Int$ is
- Not many interesting functions are higher-ranked, but those that are, are hard to work-around
- Delayed instantiation: generate a fresh β and recall the (later-to-take-place) instantiation with a constraint $\forall a.a \rightarrow a \leqslant \beta$
  - $\tau \leqslant \tau'$ means τ is at least as polymorphic as τ'
- The rule expressing this has various problems of hygiene
- Our solution is to express binding explicitly in the terms: λ-tree terms

The CHR formalism

The language of CHRs has three kinds of rules:

$$
\begin{array}{lcccl}
 & H^r & \iff & G \mid B & \text{simplification} \\
H^k & & \implies & G \mid B & \text{propagation} \\
H^k \setminus & H^r & \iff & G \mid B & \text{simpagation}
\end{array}
$$

where H^k (kept), H^r (removed) and B are sets of constraints, G is a guard.

- CHRs are applied non-deterministically, no backtracking
- Proving confluence (order does not matter) and termination is for the author of the rules

What have we accomplished?

- extending the matching of CHRs from ground terms to λ-tree terms
- introducing nominal constants and the ∇ operator, leading to CHR$^{\nabla}$
- generalize existing techniques for dealing confluence and termination in this setting
- we illustrate our extensions by modeling simple higher-rank types
- The mostly technical details are in the paper.
- Note: integration of λ-tree syntax with ∇ is known (Baelde, 2014), integration with CHRs is new

Ingredient 1: λ-tree syntax

- We introduce λ-tree syntax [Miller, 2000] for modeling binding in our language
- In addition to syntactic equality we also have:

$$\begin{array}{rcll} \lambda x.\, B & = & \lambda y.\, B[x \mapsto y] \quad \text{if } y \text{ not free in } B & (\alpha) \\ (\lambda x.\, B)\, E & = & B[x \mapsto E] \text{ if } E \text{ does not contain } x & (\beta) \\ \lambda x.\, F\ x & = & F & (\eta) \end{array}$$

- We must generalize from unification to L_λ unification (higher-order pattern unification), a decidable restriction of higher-order unification
- Aside: in this setting a simpler rule for (β), (β_0) will do
- Aside: Lambda-Prolog also uses λ-tree terms

Ingredient 2: the generic quantifier ∇

- Setting out to prove $\forall x.F$, we can
  (1) prove $F[x \to T]$ for all possible T
  (2) prove $F[x \to c]$ for a new nominal constant c
- But $\forall x\, y.P(x, y) \implies \forall z.P(z, z)$ holds for (1) but not for (2).
- Miller and Tiu (2005) introduce ∇ as the quantifier to distinguish the 2nd from the 1st
- It generates fresh nominal constants during proof development
- Aside: nominal constants are also known as Skolems and rigid variables

The CHRs for dealing with higher-rank types

$$
\begin{array}{rcl}
T \leqslant T & \Longleftrightarrow & \mathit{true} \\
\mathit{con}(C_1, \mathit{Args}_1) \leqslant T_2 & \Longleftrightarrow & \mathit{con}(C_1, \mathit{Args}_1) = T_2 \\
\mathit{fn}(S_1, T_1) \leqslant T_2 & \Longleftrightarrow & \mathit{fn}(S_1, T_1) = T_2 \\
\mathit{forall}(Q) \leqslant T_2 & \Longleftrightarrow & \exists V.\, Q\ V \leqslant T_2 \\
 & & \text{if } T_2 \not\equiv \mathit{forall}(R) \\
T_1 \leqslant \mathit{forall}(Q) & \Longleftrightarrow & \nabla V.\, T_1 \leqslant Q\ V
\end{array}
$$

- The last rule used to strip off $\forall$s on the right, then we strip them off on the left, and then the first three can make progress
- The semantics deal with $\exists$ and ∇ differently: in the latter case also α, β_0 and η rules are applied

Ingredient 3: permuting nominal constants

- Guaranteed distinctness of nominal constants has its problems: $\nabla x.B(x) \vdash \nabla y.B(y)$ is not true!
- A special rule $\text{ID}\pi$ is added to permute names of nominal constants
- Eg. choosing fresh constant a for x and b for y, we get $B(a) \vdash B(b)$. Under permutation $[a \mapsto b, b \mapsto a]$ $B(b)$ becomes $B(a)$ and all is right.
  - Note: both sides may use their own permutation

What about termination and confluence?

- As usual, it is up to the author to prove this
- Standard techniques apply, however
  - the level mapping needed for proving termination must be independent of the particular choice of nominal constant
  - for confluence unifiers come with a renaming for the nominal constants

Meanwhile, in the UK...

- Alejandro is in Bristol this week, presenting another part of our work at Implementation and Application of Functional Language
- The subject of that work is to add implications by means of scopes to CHR$^{\nabla}$to model local reasoning (Haskell's GADTs), and higher-rank types that also support type classes
- There is an implementation of an impredicative, higher-rank type system at `https://git.science.uu.nl/f100183/quique` that uses our implementation of CHR$^{\nabla}$, separately available at `https://git.science.uu.nl/f100183/uchrp`

Thank you for your attention.