Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules

Alejandro Serrano and Jurriaan Hage
Universiteit Utrecht
A.SerranoMena@uu.nl

NL FP Dag 2016
Utrecht, 8 January 2016

DSLs are the best!

Domain-specific languages are a widely used tool

- Focus on a particular problem
- Embody expert knowledge
- More likely to be used without prior experience

Two approaches to their development

- External: custom compiler and tool chain
- **Internal**: integrated in a host language
  - Common in the functional programming community

One example: Persistent

`persistent` is a Haskell library for database access

- Support for both relational and non-relational databases
- Type-safe approach: each entity is assigned a Haskell type
- Strict separation between:
  1. Values which are kept in the database, e
  2. Primary keys to a certain value, $Key\ e$
  3. Combinations of key and value, $Entity\ e$

$$
\begin{array}{lll}
get & :: Key\ v & \rightarrow m\ (Maybe\ v) \\
insert & :: v & \rightarrow m\ (Key\ v) \\
delete & :: Key\ v & \rightarrow m\ () \\
replace & :: Key\ v \rightarrow v & \rightarrow m\ () \\
update & :: Key\ v \rightarrow [Update\ v] & \rightarrow m\ () \\
select & :: [Filter\ v] \rightarrow [SelectOpt\ v] & \rightarrow m\ [Entity\ v]
\end{array}
$$

But if you ever write ill-typed code...

replace 1 *alejandro*

```
No instance for (Num (Key Person))
arising from the literal '1'
```

replace (*key banana*) *alejandro*

```
Cannot unify 'Fruit' with 'Person'
```

- The DSL is not transparent when an error occurs
- Implementation details leak in error messages
  - It gets worse as the host language becomes more complex

Introducing DOMSTED

DOMain Specific Type Error Diagnosis

- Enable embedded DSL developers to control the error messages produced by the compiler
- Focus on those errors coming from ill-typed expressions
- Target a full-blown type system
  - Not simply-typed λ-calculus with maybe small extensions
  - Haskell 2010 + type classes, functional dependencies, type families, GADTs, kind polymorphism...
  - In the works: higher-rank and impredicative instantiation
- Constraint-based approach to typing

Our solution: specialized type rules

```
rule replace_key
case ((replace .#key) .#value )#e {
  join {constraints #key, constraints #value},
   #key ~ Key v
        error { #key : expr "should be a Key."
                "Did you forget a wrapper?"},
   v    ~ #value
        error {"Key type" v : ty "and value type"
                #value : ty "do not coincide"},
   #e   ~ m ()
}
```

- Custom error messages
- Ordering for constraint solving

Why does ordering matter?

Suppose you have the following constraints:

$$\alpha \sim \mathit{Int} \quad \alpha \sim \mathit{Bool} \quad \alpha \sim \mathit{Char}$$

The error you get depends on the order of solving:

- `Cannot unify Int with Bool`
- `Cannot unify Int with Char`
- `Cannot unify Bool with Char`
- `Cannot unify Int, Bool and Char`

Sometimes you want to suggest reparations

$$(\equiv) \;::\; Eq\ a \Rightarrow a \to a \to Bool$$
$$(\equiv.) :: PersistField\ t \Rightarrow EntityField\ v\ t \to t \to Filter\ v$$

$$select\ [PersonName \equiv .\texttt{"Alejandro"}]\ []$$

Sometimes you want to suggest reparations

$$(\equiv) \;::\; Eq\ a \Rightarrow a \to a \to Bool$$
$$(\equiv.) :: PersistField\ t \Rightarrow EntityField\ v\ t \to t \to Filter\ v$$

$$select\ [PersonName \equiv . \texttt{"Alejandro"}]\ []$$

rule *wrong_eq_filter*
case $(\equiv)\ .\#field\ \ .\#value$
when $\#field \sim EntityField\ \#value\ t\ \{$
  repair $\{$`"Database field"` $\#field : expr$
    `"is being compared using (==)."`
    `"Did you intend to use (==.) instead?"`$\}$
$\}$

Universiteit Utrecht

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

Sometimes you want to get back old messages

Why *map* instead of *fmap*?

- One reason, better error messages for beginners

Sometimes you want to get back old messages

Why *map* instead of *fmap*?

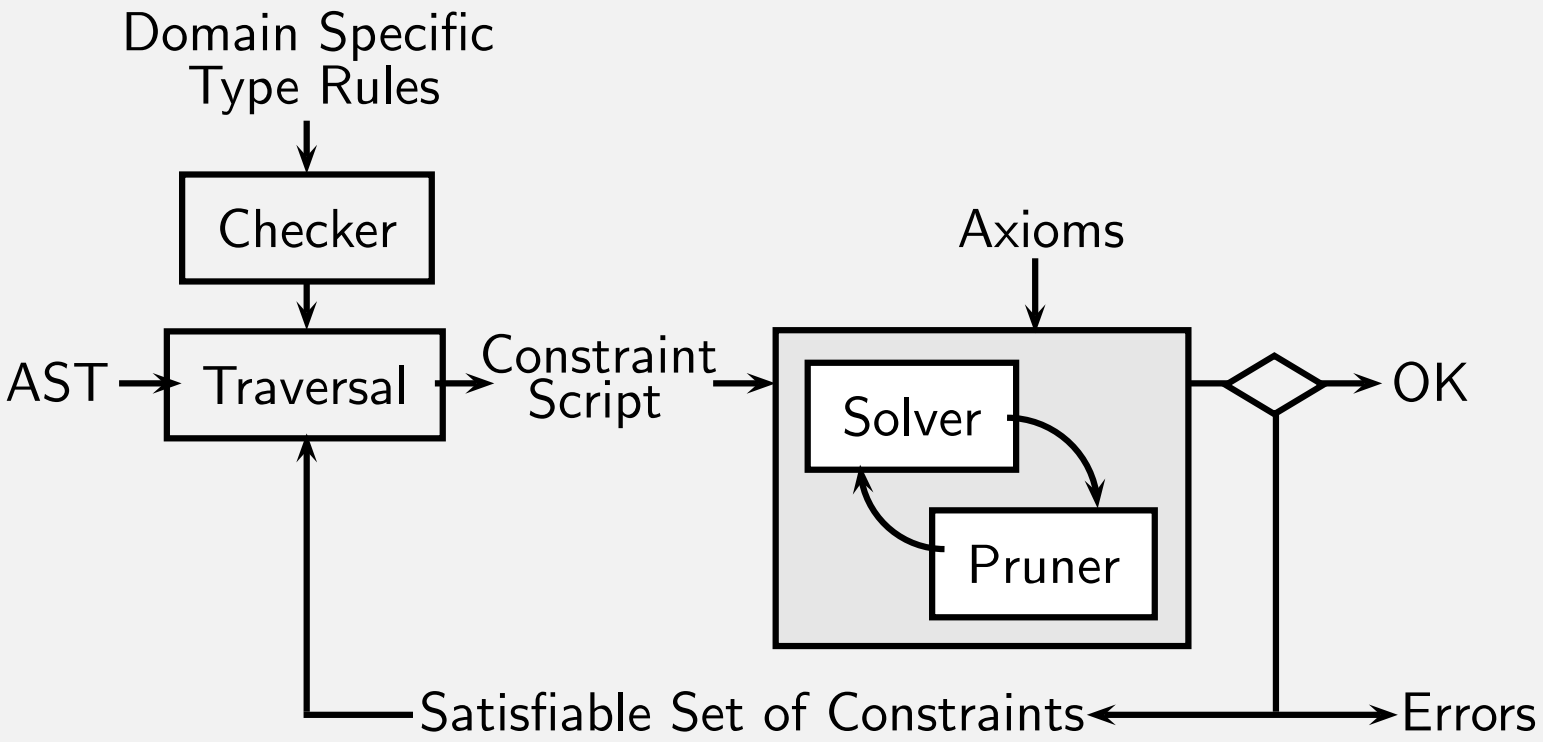
- One reason, better error messages for beginners

```
rule  fmap_on_lists
case  ((fmap .#fn) .#lst )#e
when #lst ~ [a] {
   constraints #fn,
   #fn ~ s → r error { #fn : expr "is not a function" },
   constraints #lst,
   #lst ~ [b],
   s ~ b error {"Domain type" s : ty
                "and list type" b : ty
                "do not coincide"},
   #e ~ [r]
}
```

How to approach type-sensitive rules?

1. Interleave constraint gathering and solving
   - It is not clear how to proceed if the solver finds an inconsistency while gathering
   - The decision to apply a type rule is biased by the order of gathering, bottom-up or top-down
     - A bidirectional solution seems overly complex
2. Perform two stages of gathering and solving

Two-stage specialized type rules, of course!



Two stages for one example

$$(\mathit{PersonName}\ ^{\beta} \equiv^{\alpha}\ \texttt{"Alejandro"}\ ^{\gamma})\ ^{\delta}$$

Two stages for one example

$$((\equiv)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

No specialized type rule is applied

$$\alpha \sim \rho \to \rho \to \mathit{Bool} \qquad \alpha \sim \beta \to \gamma \to \delta$$
$$\alpha \sim \mathit{EntityField\ Person\ String} \qquad \beta \sim \mathit{String}$$

Two stages for one example

$$((\equiv)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

No specialized type rule is applied

$$\alpha \sim \rho \to \rho \to \mathit{Bool} \qquad \alpha \sim \beta \to \gamma \to \delta$$
$$\alpha \sim \mathit{EntityField\ Person\ String} \qquad \beta \sim \mathit{String}$$

$$\Downarrow$$

Inconsistent!

Two stages for one example

$$((\equiv)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

No specialized type rule is applied

$$\alpha \sim \rho \to \rho \to \mathit{Bool} \qquad \alpha \sim \beta \to \gamma \to \delta$$
$$\alpha \sim \mathit{EntityField\ Person\ String} \qquad \beta \sim \mathit{String}$$

$$\Downarrow$$

Inconsistent!

Prune the constraint set until satisfiability

$$\alpha \sim \beta \to \gamma \to \delta \quad \alpha \sim \mathit{EntityField\ Person\ String} \quad \beta \sim \mathit{String}$$

Two stages for one example

$$((\equiv)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

No specialized type rule is applied

$$\alpha \sim \rho \to \rho \to \mathit{Bool} \qquad \alpha \sim \beta \to \gamma \to \delta$$
$$\alpha \sim \mathit{EntityField\ Person\ String} \qquad \beta \sim \mathit{String}$$

$$\Downarrow$$

Inconsistent!

Prune the constraint set until satisfiability

$$\alpha \sim \beta \to \gamma \to \delta \quad \alpha \sim \mathit{EntityField\ Person\ String} \quad \beta \sim \mathit{String}$$

$$\Downarrow$$

Now the specialized type rule kicks in

$\bot$ `Database field PersonName is being compared using (==).`

Two stages for one example

$$((\equiv)^{\alpha}\ \mathit{PersonName}^{\beta}\ \texttt{"Alejandro"}^{\gamma})^{\delta}$$

No specialized type rule is applied

$$\alpha \sim \rho \to \rho \to \mathit{Bool} \qquad \alpha \sim \beta \to \gamma \to \delta$$
$$\alpha \sim \mathit{EntityField\ Person\ String} \qquad \beta \sim \mathit{String}$$

$$\Downarrow$$

Inconsistent!

Prune the constraint set until satisfiability

$$\alpha \sim \beta \to \gamma \to \delta \quad \alpha \sim \mathit{EntityField\ Person\ String} \quad \beta \sim \mathit{String}$$

$$\Downarrow$$

Now the specialized type rule kicks in

$$\bot\ \texttt{Database field PersonName is being compared using (==).}$$

$$\Downarrow$$

The desired error message is shown to the user

Soundness and completeness

Specialized type rules should not tamper the type system

1. Generate a meta-expression which encompasses all possible instantiations of the type rule
2. Gather set of constraints S_{with} using specialized type rules
3. At the same time, recall all type preconditions $\mathcal{P}$
4. Gather set of constraints S_{none} using only default type rules
5. Prove that $\mathcal{P} \wedge S_{with} \implies S_{none}$ (soundness)
   and/or $\mathcal{P} \wedge S_{none} \implies S_{with}$ (completeness)

Meanwhile, in GHC...

```
instance TypeError (Text "Cannot 'Show' functions." :$$:
                    Text "Perhaps a missing argument?")
  => Show (a -> b) where ...
```

- Leverages the rest of type-level techniques in GHC
- Only available for type class and family resolution
- May not influence the ordering of constraints
- No specialization
  - Messages cannot depend on the function being used

- Specialized type rules enable developers to give custom error messages for their DSLs
- Rules might depend on syntactic and type-level information
  - Suggest reparations for common errors
  - Enable custom messages for concrete scenarios
- A two-stage approach enables that second possibility

- Specialized type rules enable developers to give custom error messages for their DSLs
- Rules might depend on syntactic and type-level information
  - Suggest reparations for common errors
  - Enable custom messages for concrete scenarios
- A two-stage approach enables that second possibility

Thanks for listening!