

# Ruler: programming type rules

Atze Dijkstra

October 6, 2005

# The problem

- ▶ Suppose you have a language

$e ::= int$	literals
$i$	program variable
$e e$	application
$\lambda i \rightarrow e$	abstraction
<b>let</b> $i = e$ <b>in</b> $e$	local definitions

- ▶ With a type system specification (partially given)

$$\frac{i \mapsto \sigma \in \Gamma}{\Gamma \vdash^e i : \tau} \text{E.VAR}_E \quad \frac{\Gamma \vdash^e a : \tau_a \quad \Gamma \vdash^e f : \tau_a \rightarrow \tau}{\Gamma \vdash^e f a : \tau} \text{E.APP}_E$$

## The problem

- ▶ With an implementation (partially given)

**data** *Expr*

| *Var* *i* : { *String* }

**attr** *Expr* [*g* : *Gam* | *c* : *C* | *ty* : *Ty*]

**sem** *Expr*

| *Var* (**lhs.uniq**, **loc.uniq1**)

    = *rulerMk1Uniq* @**lhs.uniq**

    (**loc.pty\_**, **loc.nmErrs**)

    = *gamLookup* @*i* @**lhs.g**

**lhs.ty** = *tyInst* @*uniq1* @*pty\_*

- ▶ That's simple, isn't it?

# The real problem

... is complexity, arising with feature combination

- ▶ EHC: impredicativity propagation algorithm (for application)

$$o; \Gamma; \mathbb{C}^k; \mathcal{C}^k; \sigma^k \vdash^e e : \sigma; \sigma \rightsquigarrow \mathbb{C}; \mathcal{C}$$

$$\frac{\begin{array}{c} v \text{ fresh} \\ o_{str}; \Gamma; \mathbb{C}^k; \mathcal{C}^k; v \rightarrow \sigma^k \vdash^e e_1 : \sigma_f; \_ \rightarrow \sigma \rightsquigarrow \mathbb{C}_f; \mathcal{C}_f \\ o_{im} \vdash^{\leq} \sigma_f \leq \mathbb{C}_f(v \rightarrow \sigma^k) : \_ \rightsquigarrow \mathbb{C}_F \\ o_{inst-lr}; \Gamma; \mathbb{C}_F \mathbb{C}_f; \mathcal{C}_f; v \vdash^e e_2 : \sigma_a; \_ \rightsquigarrow \mathbb{C}_a; \mathcal{C}_a \\ fl_{alt}^+, o_{inst-l} \vdash^{\leq} \sigma_a \leq \mathbb{C}_a v : \_ \rightsquigarrow \mathbb{C}_A \\ \mathbb{C}_1 \equiv \mathbb{C}_A \mathbb{C}_a \end{array}}{o; \Gamma; \mathbb{C}^k; \mathcal{C}^k; \sigma^k \vdash^e e_1 e_2 : \mathbb{C}_1 \sigma^k; \mathcal{C}_a \sigma \rightsquigarrow \mathbb{C}_1; \mathcal{C}_a} \text{E.APP12}$$

(Just for the idea)

sem Expr

```
| App (func.gUniq, loc.uniq1, loc.uniq2, loc.uniq3)
                                     = mkNewLevUID3 @lhs.gUniq
loc .tvarv_                          = mkTyVar @uniq1
func.fiOpts                          = o_str
  .knTy                              = [@tvarv_] 'mkArrow' @lhs.knTy
( _, loc.ty_ )                       = tyArrowArgRes @func.ty
loc .fo_fitF_                        = fitsIn o_im @fe @uniq2 @func.imprTy (@func.imprTyCnstr
arg .imprTyCnstr                    = foCnstr @fo_fitF_ ⊕ @func.imprTyCnstr
  .fiOpts                            = o_inst_lr
  .knTy                              = @tvarv_
loc .fo_fitA_                        = fitsIn (o_inst_lr { fi_alt = True }) @fe @uniq3 @arg.imprTyCnstr
  .imprTyCnstr_1_                    = foCnstr @fo_fitA_ ⊕ @arg.imprTyCnstr
lhs .imprTyCnstr                    = @imprTyCnstr_1_
loc .imprTy                          = @imprTyCnstr_1_ ⊕ @lhs.knTy
  .ty                                = @arg.tyCnstr ⊕ @ty_
```

# The question

How can you be sure the implementation really implements the type system?

- ▶ Prove it (after the construction of both)
  - ▶ Proving correctness of programs is difficult
- ▶ Prove correctness of type system against a simple model, then generate the implementation
  - ▶ Real languages are still too complex to be mechanically proven correct
  - ▶ Lack of full specification of real languages
- ▶ Describe type rules such that pretty printed version and implementation can be generated (Ruler)
  - ▶ Not proven correct, but at least consistent

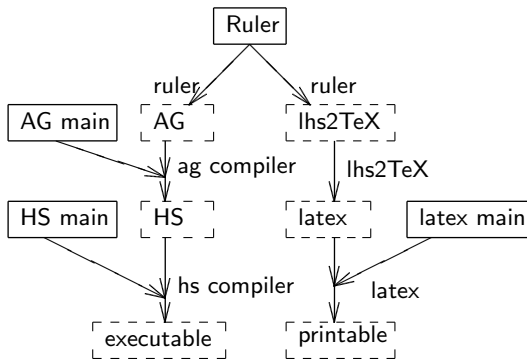
# Ruler approach

- ▶ Ruler system
  - ▶ Domain specific language for specifying type rules
- ▶ We want to check
  - ▶ Have judgements in a rule the right structure (similar to type checking for expressions)
  - ▶ Are identifiers defined before used (when specifying the algorithmic part)
  - ▶ ... more in the future
- ▶ We want to generate
  - ▶  $\LaTeX$  pretty printed version (to include in a description)
  - ▶ AG fragments (to include in an implementation)

# Content of remainder of talk

- ▶ Overall design of Ruler
- ▶ Hindley-Milner (HM) type system as a case study
- ▶ Description partitioned in three views/steps (declarative, algorithmic, AG)
- ▶ How to specify this using Ruler
- ▶ Conclusion

# Ruler: the big picture



$\square$  : source       $[ ]$  : derived

$a \xrightarrow{x} b$  : derives  $b$  from  $a$  using  $x$

# Ruler concepts

- ▶ Scheme
  - ▶ the structure: holes + template
- ▶ Views
  - ▶ hierarchy of views
  - ▶ view build on top of previous view
- ▶ View on scheme
  - ▶ each scheme has view on it, differs in holes + template
- ▶ Judgement shape
  - ▶ the shape used to specify/pretty-print an instance of a scheme (a judgement)
- ▶ Rule
  - ▶ premises + conclusion (judgements)
- ▶ View on rule
- ▶ Rule judgement
  - ▶ each (rule) judgement has a view on it, parallel to view on its scheme

# Syntactic structure

**scheme**  $X =$

**view**  $A =$

**holes** ...

**judgespec** ...

**view**  $B =$

**holes** ...

**judgespec** ...

**ruleset**  $\times$  **scheme**  $X =$

**rule**  $r =$

**view**  $A =$

**judge** ... -- premises

      ...

      —

**judge** ... -- conclusion

**view**  $B = \dots$

**rule**  $s =$

**view**  $A = \dots$

**view**  $B = \dots$

# Views

- ▶ Views are ordered
- ▶ Start with specifying the first view on a rule (say, rule E.VAR)

$$\frac{i \mapsto \sigma \in \Gamma \quad \tau = \text{inst}(\sigma)}{\Gamma \vdash^e i : \tau} \text{E.VAR}_E$$

- ▶ equational/declarative view  $E$  (in Hindley-Milner type system)
- ▶ Then specify the differences relative to previous view

$$\frac{i \mapsto \sigma \in \Gamma \quad \tau = \text{inst}(\sigma)}{\mathcal{C}^k; \Gamma \vdash^e i : \tau \rightsquigarrow \mathcal{C}^k} \text{E.VAR}_A$$

- ▶ algorithmic view  $A$  (in Hindley-Milner type system)
- ▶ colors for indicating (un)changed parts

## Step 1: equational view $E$ , $\text{expr}$ scheme

- ▶ Structure/scheme for judgements

**scheme**  $\text{expr} =$

**view**  $E =$

**holes**  $[| e : \text{Expr}, \text{gam} : \text{Gam}, \text{ty} : \text{Ty} |]$

**judgespec**  $\text{gam} \vdash e : \text{ty}$

**judgeuse**  $\text{tex } \text{gam} \vdash \dots \text{"e"} e : \text{ty}$

- ▶ Type ( $\text{ty} : \text{Ty}$ ):

$\tau ::= \text{Int}$       literals

|  $v$       variable

|  $\tau \rightarrow \tau$       abstraction

$\sigma ::= \forall \bar{v}. \tau$       universally quantified type,  $\bar{v}$  possibly empty

- ▶ Environment ( $\text{gam} : \text{Gam}$ ):

$\Gamma ::= \overline{j \mapsto \sigma}$

# Ruleset

- ▶ Set of rules of a scheme

**ruleset** *expr.base* **scheme** *expr* "Expression type rules" =

**rule** *e.app* =

**view** *E* =

**judge** *A* : *expr* = *gam* ⊢ *a* : *ty.a*

**judge** *F* : *expr* = *gam* ⊢ *f* : (*ty.a* → *ty*)

—

**judge** *R* : *expr* = *gam* ⊢ (*f a*) : *ty*

- ▶ ruleset displays as a figure (not shown here)

- ▶ L<sup>A</sup>T<sub>E</sub>X rendering (via *lhs2TeX*)

$$\frac{\Gamma \vdash^e a : \tau_a \quad \Gamma \vdash^e f : \tau_a \rightarrow \tau}{\Gamma \vdash^e f a : \tau} \text{E.APPE}$$

# Relation

- ▶ Arbitrary conditions

**rule**  $e.var =$

**view**  $E =$

**judge**  $G : gamLookupIdTy = i \mapsto pty \in gam$

**judge**  $I : tyInst = ty '=' inst (pty)$

—

**judge**  $R : expr = gam \vdash i : ty$

- ▶ L<sup>A</sup>T<sub>E</sub>X rendering

$$\frac{i \mapsto \sigma \in \Gamma \quad \tau = inst(\sigma)}{\Gamma \vdash^e i : \tau} \text{E.VAR}_E$$

- ▶ Relation

**relation**  $gamLookupIdTy =$

**view**  $E =$

**holes**  $[ | nm : Nm, gam : Gam, ty : Ty | ]$

**judgespec**  $nm \mapsto ty \in gam$

## Step 2: algorithmic view $A$

- ▶ On top of view  $E$
- ▶ View hierarchy  
**viewhierarchy** =  $E < A < AG$ 
  - ▶ may be tree like hierarchy
- ▶ Algorithmic view
  - ▶ use of constraints/substitution  
$$\mathcal{C} ::= \overline{v \mapsto \tau}$$
  - ▶ computation has direction

# Direction of computation

- ▶ Holes

- ▶ specify direction (similar to AG's attributes)

**scheme**  $expr =$

**view**  $A =$

**holes**  $[e : Expr, gam : Gam \mid \mathbf{thread} \text{ } cnstr : \mathcal{C} \mid ty : Ty]$

**judgespec**  $cnstr.inh; gam \vdash \dots "e" \ e : ty \rightsquigarrow cnstr.syn$

**judgeuse** – **tex**

- ▶ may be tree like hierarchy

- ▶ Algorithmic view

- ▶ use of constraints/substitution

$\mathcal{C} ::= \overline{v \mapsto \tau}$

- ▶ computation has direction

## View A on App

- ▶ Specify the differences (for rule e.app)
- ▶ Previous

$$\frac{\Gamma \vdash^e a : \tau_a \quad \Gamma \vdash^e f : \tau_a \rightarrow \tau}{\Gamma \vdash^e f a : \tau} \text{E.APP}_E$$

- ▶ New

$$\frac{\begin{array}{l} \mathcal{C}^k; \Gamma \vdash^e f : \tau_f \rightsquigarrow \mathcal{C}_f \\ \mathcal{C}_f; \Gamma \vdash^e a : \tau_a \rightsquigarrow \mathcal{C}_a \\ v \text{ fresh} \\ \tau_a \rightarrow v \cong \mathcal{C}_a \tau_f \rightsquigarrow \mathcal{C} \end{array}}{\mathcal{C}^k; \Gamma \vdash^e f a : \mathcal{C} \mathcal{C}_a v \rightsquigarrow \mathcal{C} \mathcal{C}_a} \text{E.APP}_A$$

## View A on App

► New:

**view**  $A =$

**judge**  $V : tvFresh = tv$

**judge**  $M : match = (ty.a \rightarrow tv) \cong (cnstr.a \ ty.f)$   
 $\rightsquigarrow cnstr$

**judge**  $F : expr$

|  $ty = ty.f$

|  $cnstr.syn = cnstr.f$

**judge**  $A : expr$

|  $cnstr.inh = cnstr.f$

|  $cnstr.syn = cnstr.a$

—

**judge**  $R : expr$

|  $ty = cnstr \ cnstr.a \ tv$

|  $cnstr.syn = cnstr \ cnstr.a$

## Step 3: AG translation view AG

- ▶ Building on top of view  $A$
- ▶ Mapping holes to attributes
  - ▶ either value construction or deconstruction
- ▶ Fresh type variables
  - ▶ threading unique value
- ▶ Error handling
  - ▶ 'side effect': error messages in hidden attribute
- ▶ The rest
  - ▶ parsing, pretty printing, ...

## View AG on App

$$\frac{\begin{array}{l} \mathcal{C}^k; \Gamma \vdash^e f : \tau_f \rightsquigarrow \mathcal{C}_f \\ \mathcal{C}_f; \Gamma \vdash^e a : \tau_a \rightsquigarrow \mathcal{C}_a \\ v \text{ fresh} \\ \tau_a \rightarrow v \cong \mathcal{C}_a \tau_f \rightsquigarrow \mathcal{C} \end{array}}{\mathcal{C}^k; \Gamma \vdash^e f a : \mathcal{C} \mathcal{C}_a v \rightsquigarrow \mathcal{C} \mathcal{C}_a} \text{E.APP}_A$$

**sem Expr**

| *App* (*f.uniq*, **loc.uniq1**)

    = *rulerMk1Uniq* @**lhs.uniq**

**loc.tv\_** = *Ty\_Var* @*uniq1*

(**loc.c\_**, **loc.mtErrs**)

    = (@*a.ty* 'Ty\_Arr' @*tv\_*)  $\cong$  (@*a.c*  $\oplus$  @*f.ty*)

**lhs.c** = @*c\_*  $\oplus$  @*a.c*

    .*ty* = @*c\_*  $\oplus$  @*a.c*  $\oplus$  @*tv\_*

## Fresh type variable

- ▶ Relation is inlined

```
relation tvFresh =
```

```
view A =
```

```
holes [||| tv : Ty]
```

```
judgespec tv
```

```
judgeuse tex tv (text "fresh")
```

```
judgeuse ag tv '=' Ty_Var unique
```

- ▶ Keyword **unique**

- ▶ insertion of *rulerMk1Uniq*

- ▶ translated to *uniq1*

- ▶ Type structure (supporting code)

```
type TvId = UID
```

```
data Ty = Ty_Any | Ty_Int | Ty_Var TvId
```

```
| Ty_Arr Ty Ty
```

```
| Ty_All [TvId] Ty
```

```
deriving (Eq, Ord)
```

# Rewriting *Ruler* expressions

- ▶ *Ruler* expression
  - ▶  $ty.a \rightarrow ty$  pretty prints as  $\tau_a \rightarrow \tau$
  - ▶ but requires rewriting for AG
- ▶ Rewrite rule
  - rewrite ag def**  $a \rightarrow r = (a) 'Ty\_Arr' (r)$
  - ▶ target: **ag**
  - ▶ when value is **defined** (constructed) for further use
- ▶ Constraining rewrite rule using *Ruler* (sort of) types
  - rewrite ag def**  $(a | Ty) \rightarrow (r | Ty) = ((a) 'Ty\_Arr' (r) | Ty)$
- ▶ Formatting identifiers (for target **ag**)
  - format ag**  $cnstr = c$
  - format ag**  $gam = g$

## Error handling

- ▶ Relation *match* is inlined

**relation** *match* =

**view** *A* =

**holes** [*ty.l* : *Ty*, *ty.r* : *Ty* || *cnstr* : *C*]

**judgespec** *ty.l*  $\cong$  *ty.r*  $\rightsquigarrow$  *cnstr*

**judgeuse** **ag** (*cnstr*, *mtErrs*) '=*(ty.l)*  $\cong$  (*ty.r*)

- ▶ Returns extra value: error messages (if any)
- ▶ Type matching (supporting code)

$(\cong) :: Ty \rightarrow Ty \rightarrow (C, [PP\_Doc])$

*Ty\_Any*  $\cong t_2 = ([], [])$

*t*<sub>1</sub>  $\cong Ty\_Any = ([], [])$

*Ty\_Int*  $\cong Ty\_Int = ([], [])$

*Ty\_Var* *v*<sub>1</sub>  $\cong Ty\_Var *v*<sub>2</sub>$

| *v*<sub>1</sub> == *v*<sub>2</sub> = ([], [])

*Ty\_Var* *v*<sub>1</sub>  $\cong t_2$

| *v*<sub>1</sub>  $\notin$  *ftv* *t*<sub>2</sub> =  $([(v_1, t_2)], [])$

...

# Conclusion

- ▶ Lightweight solution to two problems
  - ▶ Consistency between type rules and (AG) implementation
  - ▶ Understandability by stepwise construction
- ▶ Related work
  - ▶ TinkerType, TWELF, AG alike systems