Programming type rules

Arie Middelkoop, ariem@cs.uu.nl
Universiteit Utrecht, The Netherlands

(based on a presentation by Atze Dijkstra)

July 4, 2008

Our Mission

Develop languages, technologies and tools for the specification and implementation of (domain specific) languages.

- Jurriaan Hage
    - Program Analysis
    - Helium
- Johan Jeuring
    - Generics
- Doaitse Swierstra
    - Parser Combinators
    - Attribute Grammars
    - *EHC*

Use Case: EHC

Development of a complex compiler (Haskell)

- Language constructs (expressions, class system, records)
- Aspects of language construct (code, type)
- *Type rules*

Motivation

- Our experimental compiler:
  - Essential Haskell (EHC project)
- Our experiments:
  - higher-ranked types
  - impredicativity
  - existential types
  - implicit/explicit parameters
- Our desire:
  - study isolated features
  - combine them
  - *and* keep it maintainable, understandable

Motivation

Programming language research lifecycle

- Define syntax
- Define semantics
- Prove properties of semantics
- Implement
- Prove correctness of implementation
- Document

Motivation: textbook example

Syntax

$$
\begin{aligned}
e ::= {} & \mathit{int} \\
\mid {} & i \\
\mid {} & e\ e \\
\mid {} & \lambda i \rightarrow e \\
\mid {} & \textbf{let } i = e \textbf{ in } e
\end{aligned}
$$

Semantics

$$
\frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = \mathit{inst}\ (\sigma) \end{array}}{\Gamma \vdash^{e} i : \tau}\ \text{E.VAR}_{E}
$$

Implementation

```
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_
```

Motivation: real-life example

- **Semantics**

$$\frac{\begin{array}{c} v \text{ fresh} \\ o;\Gamma;\mathbb{C}^k;\mathcal{C}^k;v \to \sigma^k \vdash^e e_1 : \sigma_f;\_ \to \sigma \leadsto \mathbb{C}_f;\mathcal{C}_f \\ o_{im};\mathbb{C}_f \vdash^{\leqslant} \sigma_f \leqslant \mathbb{C}_f(v \to \sigma^k) : \_ \leadsto \mathbb{C}_F \\ o_{inst-lr};\Gamma;\mathbb{C}_F\mathbb{C}_f;\mathcal{C}_f;v \vdash^e e_2 : \sigma_a;\_ \leadsto \mathbb{C}_a;\mathcal{C}_a \\ fi^{+}_{alt}, o_{inst-l};\mathbb{C}_a \vdash^{\leqslant} \sigma_a \leqslant \mathbb{C}_a v : \_ \leadsto \mathbb{C}_A \\ \mathbb{C}_1 \equiv \mathbb{C}_A\mathbb{C}_a \end{array}}{o;\Gamma;\mathbb{C}^k;\mathcal{C}^k;\sigma^k \vdash^e e_1\ e_2 : \mathbb{C}_1\sigma^k;\sigma^k \leadsto \mathbb{C}_1;\mathcal{C}_a}\ \text{E.APP}_{I2}$$

- **Implementation**

```
sem Expr
  | App (func.gUniq, loc.uniq1, loc.uniq2, loc.uniq3)
                        = mkNewLevUID3 @lhs.gUniq
        loc  .tvarv_    = mkTyVar @uniq1
        func.knTy       = [@tvarv_] `mkArrow` @lhs.knTy
        loc  .fo_fitF_  = fitsIn o_im @fe @uniq2 @func.imprTy (@func.imprTyCnstr ⊕
        ...                    -- lots of non-obvious code
```

- **Correctness** ☹

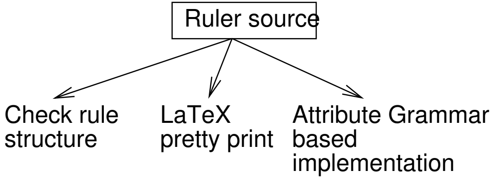
The problem

It is hard to

- Understand feature interaction
- Say something about formal properties
- Maintain consistency of semantics & implementation
- Generate implementation

Ruler

- A system for specifying type rules


Ruler source

Check rule structure

LaTeX pretty print

Attribute Grammar based implementation

Programming Type Rules

Have abstraction mechanisms and strategies to specify type rules.

- Abstraction mechanism example: views
    - Base case with increments
    - Declarative view, algorithmic view
    - Each view incorporates more detail
- Strategy examples:
    - Restrict type rules to be functions instead of arbitrary relations by specifying computation direction of variables
    - Restrict type rules to be syntax directed by specifying which variable determines what rule to apply

Ruler: example of multiple views

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

$$\frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = \mathit{inst}\ (\sigma) \end{array}}{\Gamma \vdash^{e} i : \tau}\ \textsc{e.var}_{E}$$

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

$$\frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = \mathit{inst}\ (\sigma) \end{array}}{\mathcal{C}^{k} ; \Gamma \vdash^{e} i : \tau \leadsto \mathcal{C}^{k}}\ \textsc{e.var}_{A}$$

  - algorithmic view A (in Hindley-Milner type system)
  - blue indicates the changed parts

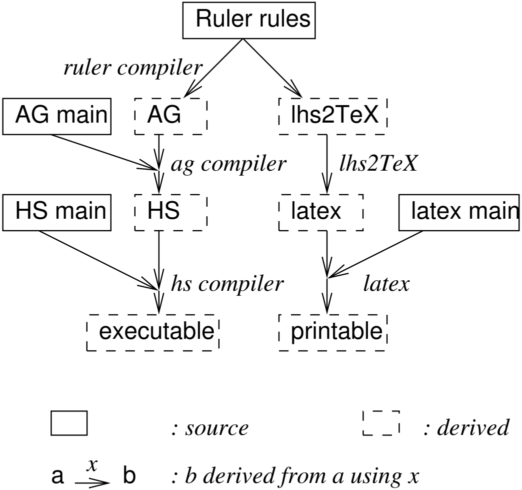
Content of remainder of talk

The tools Ruler and AG in more detail:

- Concepts of Ruler
- Case study: Hindley-Milner typing
  - Three views: E, A, AG
  - Ruler source texts and results

Omitted: feature isolation and more advanced type rule programming

Application of Ruler



Ruler rules

ruler compiler

AG main

AG

lhs2TeX

ag compiler

lhs2TeX

HS main

HS

latex

latex main

hs compiler

latex

executable

printable

☐ *: source*

⬚ *: derived*

$a \xrightarrow{x} b$ *: b derived from a using x*

Ruler concepts

- Scheme
  - judgement structure: holes + templates
  - template (or judgement shape) used to specify/output a scheme instance (a judgement)
- Views of a scheme
  - hierarchy of views, a view is built on top of previous view
  - each scheme has views, views differ in holes + templates
- Rule
  - premise judgements + conclusion judgement
  - judgement binds holes to expressions
- Views of a rule
- Rule judgement
  - each rule judgement has views, parallel to views of its scheme

Syntactic structure

```
scheme Expr =
  view E =
    holes …
    judgespec …
    judgeuse …
     …
  view A =
    holes …
    judgespec …
    judgeuse …
     …
```

```
ruleset expr_rules scheme Expr =
  rule con =
    view E =
      judge …    -- premises
       …
       –
      judge …    -- conclusion
    view A = …

  rule app =
    view E = …
    view A = …
```

Ruler 'dimensions'

- Views allow incremental extension of a language
- Schemes allow "by aspect" organisation by treating holes and associated rules together
- Ruler
    - combines views in a hierarchical, inheriting manner
    - (combines schemes into new schemes)
    - combine means overwrite of hole bindings

Case study: HM typing

- View 1: Equational (E)
  - scheme
  - rulesets
  - output
- View 2: Algorithmic (A)
  - hierarchy
  - output
  - scheme
  - rulesets
- View 3: Attribute Grammar translation (AG)

View 1: equational view E, *expr* scheme

Structure/scheme for judgements

scheme *expr* =
  view E =
    holes $[e : Expr, gam : Gam, ty : Ty]$
    judgespec $gam \vdash e : ty$
    judgeuse tex $gam \vdash$ .."e" $e : ty$

- Type $(ty : Ty)$:

$\tau ::= Int \mid Char$	literals
$\mid\ v$	variable
$\mid\ \tau \rightarrow \tau$	abstraction
$\sigma ::= \tau$	type scheme
$\mid\ \forall v.\tau$	universally quantified type, abbreviated by $\forall \overline{v}.\tau$

- Environment $(gam : Gam)$:

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

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 in documentation
- LaTeX rendering (via *lhs2TeX*)

LaTeX rendering

$$\boxed{\Gamma \vdash^{e} e : \tau}$$

$$\frac{}{\Gamma \vdash^{e} \mathit{int} : \mathit{Int}}\ \mathrm{E.INT}_{E} \qquad \frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = \mathit{inst}\ (\sigma) \end{array}}{\Gamma \vdash^{e} i : \tau}\ \mathrm{E.VAR}_{E}$$

$$\frac{\begin{array}{c} \Gamma \vdash^{e} a : \tau_a \\ \Gamma \vdash^{e} f : \tau_a \rightarrow \tau \end{array}}{\Gamma \vdash^{e} f\ a : \tau}\ \mathrm{E.APP}_{E} \qquad \frac{(i \mapsto \tau_i), \Gamma \vdash^{e} b : \tau_b}{\Gamma \vdash^{e} \lambda i \rightarrow b : \tau_i \rightarrow \tau_b}\ \mathrm{E.LAM}_{E}$$

$$\frac{\begin{array}{c} (i \mapsto \sigma_e), \Gamma \vdash^{e} b : \tau_b \\ \Gamma \vdash^{e} e : \tau_e \\ \sigma_e = \forall \overline{v}.\tau_e, \quad \overline{v} \notin \mathit{ftv}\ (\Gamma) \end{array}}{\Gamma \vdash^{e} \mathbf{let}\ i = e\ \mathbf{in}\ b : \tau_b}\ \mathrm{E.LET}_{E}$$

Relation

Arbitrary conditions

rule $e.var =$
view $E =$
judge $G : \mathit{gamLookupIdTy} = i \mapsto \mathit{pty} \in \mathit{gam}$
judge $I : \mathit{tyInst} = \mathit{ty}\ `{=}`\ \mathit{inst}\ (\mathit{pty})$
–
judge $R : \mathit{expr} = \mathit{gam} \vdash i : \mathit{ty}$

- Condition *gamLookupIdTy*: identifier must be bound to type in environment
- Condition *tyInst*: monotype is instantiation of polytype

Relation

Relation

$$\begin{array}{l}\textbf{relation}\ \mathit{gamLookupIdTy} =\\ \quad \textbf{view}\ E =\\ \quad\quad \textbf{holes}\ [\mathit{nm} : \mathit{Nm}, \mathit{gam} : \mathit{Gam}, \mathit{ty} : \mathit{Ty}]\\ \quad\quad \textbf{judgespec}\ \mathit{nm} \mapsto \mathit{ty} \in \mathit{gam}\end{array}$$

- LaTeX rendering when used

$$\frac{\begin{array}{c} i \mapsto \sigma \in \Gamma \\ \tau = \mathit{inst}\ (\sigma) \end{array}}{\Gamma \vdash^{e} i : \tau}\ \mathrm{E.VAR}_{E}$$

View 2: algorithmic view A

View hierarchy

$\textbf{viewhierarchy} = E < A < AG$

- View A on top of view E
- May be tree like hierarchy

View *A* on *App*

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

$$\frac{\begin{array}{c}\Gamma \vdash^{e} a : \tau_a \\ \Gamma \vdash^{e} f : \tau_a \to \tau\end{array}}{\Gamma \vdash^{e} f\ a : \tau}\ \mathrm{E.APP}_{E}$$

- New

$$\frac{\begin{array}{c}\mathcal{C}^{k}; \Gamma \vdash^{e} f : \tau_f \leadsto \mathcal{C}_f \\ \mathcal{C}_f; \Gamma \vdash^{e} a : \tau_a \leadsto \mathcal{C}_a \\ v \text{ fresh} \\ \tau_a \to v \cong \mathcal{C}_a \tau_f \leadsto \mathcal{C}\end{array}}{\mathcal{C}^{k}; \Gamma \vdash^{e} f\ a : \mathcal{C}\ \mathcal{C}_a v \leadsto \mathcal{C}\ \mathcal{C}_a}\ \mathrm{E.APP}_{A}$$

Direction of computation

New for scheme expr: holes with direction

$$
\begin{array}{l}
\textbf{scheme}\ \mathit{expr} = \\
\quad \textbf{view}\ A = \\
\quad\quad \textbf{holes}\ [\textbf{inh}\ \mathit{gam} : \mathit{Gam}, \textbf{thread}\ \mathit{cnstr} : \mathcal{C}, \textbf{syn}\ \mathit{ty} : \mathit{Ty}] \\
\quad\quad \textbf{judgespec}\ \mathit{cnstr}.\textbf{inh}; \mathit{gam} \vdash e : \mathit{ty} \leadsto \mathit{cnstr}.\textbf{syn} \\
\quad\quad \textbf{judgeuse}\ \textbf{tex}\ \mathit{cnstr}.\textbf{inh}; \mathit{gam} \vdash ..\texttt{"e"}\ e : \mathit{ty} \leadsto \mathit{cnstr}.\textbf{syn}
\end{array}
$$

- Algorithmic view
  - use of constraints/substitution
    $\mathcal{C} ::= \overline{v \mapsto \tau}$
  - computation has direction

Specify the differences

New for rule e.app in ruleset expr

```
view A =
  judge V : tvFresh = tv
  judge M : match  = (ty.a → tv) ≅ (cnstr.a ty.f)
                     ⇝ 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
```

Attribute Grammars

- With Attribute Grammars you can define tree walks using intuitive concepts of inherited and synthesized attributes
- Concepts:
  - Abstract Syntax Tree
  - Attributes (inherited and synthesized)
  - Definitions
- UUAGC is a preprocessor for Haskell that generates efficient tree walks

Specification of the AST

```
data TypedExpr
    | Con
        x  : String
    | Var
        x  : String
    | App
        f  : TypedExpr
        a  : TypedExpr
    | Lam
        x  : String
        tp : Ty
        e  : TypedExpr
        b  : TypedExpr
```

- Terminology: Nonterminals, Terminals, Productions/alternatives

Attributes

```
attr Expr
  inh gam : Gam
  syn ty : Ty

sem Expr
    | Con Var
        lhs.ty = lookup @x @lhs.gam
    | App
        lhs.ty = if argPart @f.ty ==  @a.ty
                 then resPart @f.ty
                 else error "arg and res do not match."
    | Lam
        e.gam = insert @x @tp @lhs.gam
        lhs.ty = if @tp ==  @e.ty
                 then @b.ty
                 else error "type for x does not match"
```

Interface

```
typecheck :: TypedExpr → Gam → Tp
typecheck e initialEnv
   = let i = Inh_Expr{gam_Inh_Expr = initialEnv}
         s = wrap_Expr (sem_Expr e) i
     in ty_Syn_Expr s
```

View 3: AG translation view *AG*

- Built on top of view *A*
- Mapping rules to data type alternatives
- 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*

- Binding an AST to rules
- **data** definition (similar to Haskell/AG)

```
data Expr [expr]
  view E
    | App [e.app] f   :: Expr
                  a   :: Expr
    | Int [e.int] int :: Int
    | Var [e.var] i   :: String
    | Lam [e.lam] i   :: String
                  b   :: Expr
    | Let [e.let ] i  :: String
                  e   :: Expr
                  b   :: Expr
```

View *AG* on *App*

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

```
sem Expr
  | App (f.uniq, loc.uniq1)
                   = rulerMk1Uniq @lhs.uniq
         loc.tv_ = Ty_Var @uniq1
         (loc.c_, loc.mtErrs)
                   = (@a.ty ‘Ty_Arr‘ @tv_) ≅ (@a.c ⊕ @f.ty)
         lhs.ty    = @c_ ⊕ @a.c ⊕ @tv_
            .c     = @c_ ⊕ @a.c
```

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

  $$\textbf{rewrite ag def}\ a \rightarrow r = (a)\ \text{`}Ty\_Arr\text{`}\ (r)$$

  - target: **ag**
  - when value is **def**ined (constructed) for further use
- Formatting identifiers (for target **ag**)

  $$\textbf{format ag}\ cnstr = c$$
  $$\textbf{format ag}\ gam \ \ = g$$

Conclusion

- Lightweight solution to two problems
  - consistency between type rules and (AG) implementation
  - understandability & manageability by stepwise (& aspectwise) construction
- Current state
  - major part of EHC type rules described by *Ruler*
  - focus of my research
- See `http://www.cs.uu.nl/wiki/Ehc/WebHome`