Exploring Ruler and DSLs for Type Systems

John Van Schie and Mark Snyder

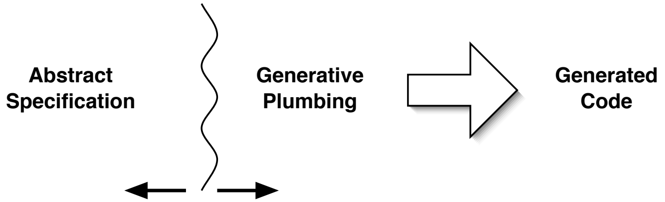
November 3, 2006

Outline

Outline:

1. Problem space
2. Ruler's Design
3. Language experimentation
4. Generating Unification code
5. Concluding Remarks

The Problem Space



Competing goals:

- we want abstract specifications
- we want generic "plumbing" code

Extra complexity:

- the problem size requires staging
- additional targets: TeX, HOL, ...

Clarifying Criteria

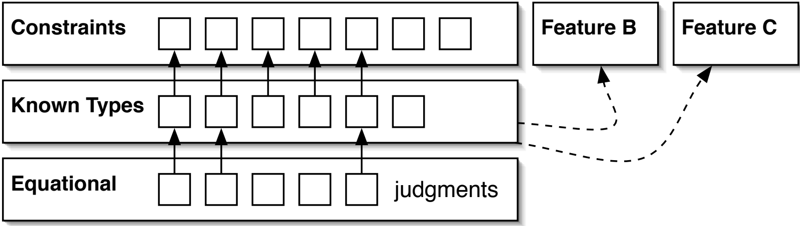
How to Miss the Mark

- DSL $\neq$ Literate Program Language
- Generative Engine $\neq$ Collection of Templates
- Software Engineering needs can't be ignored
  - support for iterative development
  - software composition mechanisms
  - literate code

So, what about TinkerType and Ruler?

Tinker Type (Vision vs. Reality)

The Vision:

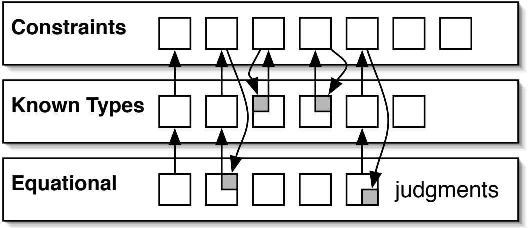


The Reality:

- used for book not full fledged type system
- composition of code and TeX fragments
    - apparently no generation
- composition safety? via feature predicates

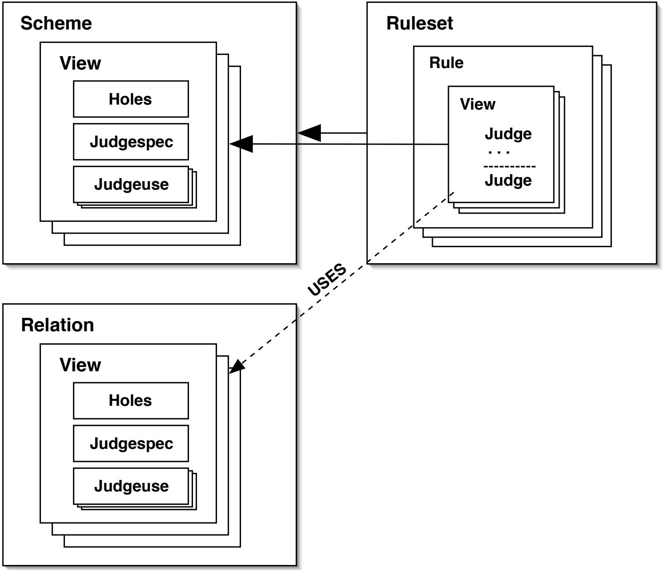
Ruler

Ruler's Layering:



- generates code and TeX
  - extensive use of special-case templates
- variability at level of holes (attributes)
  - tight coupling between multiple layers
- composition directed by explicit ordering
  - no feature/attribute usage constraints

Ruler Abstractions



Ruler Syntax (Schemes)


```
scheme expr "Expr" =
  view E =
    holes [ node e: Expr, valGam: ValGam
          , tyGam: TyGam, kiGam: KiGam | ty: Ty | ]
    judgespec kiGam ; tyGam ; valGam :- e : ty
    judgeuse tex valGam :-.."e" e : ty
  view K =
    holes [ knTy: Ty | | retain ty: Ty ]
    judgespec kiGam ; tyGam; valGam; knTy :- e : ty
    judgeuse tex valGam; knTy :-.."e" e : ty
  ...
```

Ruler Syntax (Rules)


```
ruleset expr.base scheme expr "Expression type rules"
 rule e.int "IConst" =
  view E =
   ---
   judge R : expr = kiGam ; tyGam ;
                    valGam :- int : tyInt
  view K =
   judge F : fit = :- tyInt <= knTy : fo : ty
   ---
   judge R : expr
       | ty = ty
 ...
```

Ruler Syntax (Relations)


```
relation fit =
 view K =
   holes [ lty: Ty, rty: Ty | | fo: FIOut, ty: Ty ]
   judgespec :-.."<=" lty <= rty : fo : ty
   judgeuse tex :-.."<=" lty <= rty : ty
   judgeuse ag (retain fo) `=` (lty) `fitsIn` (rty)
          | ty `=` foTy (fo)
 view C =
   holes [ | | cnstr: Cnstr ]
   judgespec :-.."<=" lty <= rty : fo : ty ~> cnstr
   judgeuse tex :-.."<=" lty <= rty : ty ~> cnstr
   judgeuse ag (retain fo) `=` fitsIn ^ ((lty) - (rty
          | ty `=` foTy (fo)
          | cnstr `=` foCnstr (fo)
 ...
```

Ruler Generated AG

```
SET AllTyExpr        = TyExpr

ATTR AllDecl AllExpr [ valGam: ValGam | | ]
ATTR AllExpr [ tyGam: TyGam | | ]
ATTR AllExpr [ knTy: Ty | | ]
ATTR AllExpr [ | | ty: Ty ]

SEM Expr
  | IConst  loc  .  fo_  =  tyInt `fitsIn` @lhs.knTy
                 .  ty   =  foTy @fo_
```

Our Ruler Experience

What We Liked:

- localized concrete syntax
- fine grained control via holes
  - minimizes changes from layer to layer

What We Didn't Like:

- inaccessible terminology
- presentation mixed with logic
- having to view all layers at once
  - more than 3000 lines of code

Experimenting with the Language

Synthesizing Requirements in Hindsight

EHC Goal: Open Platform for Research and Education

- minimizing ramp-up time is a priority
  - more so than conciseness
- leverage well known terminology
- decouple presentation from logic
- separate different layers
  - use oop-style inheritance
  - introduce notion of visibility
  - add hooks for extensibility

Experimenting with Language Design

Critiquing Ourselves

- why our inexperience is a liability
  - limited exposure to use-cases
  - a bias for verbose and fully explicit languages

Experimenting with Language Design

Critiquing Ourselves

- why our inexperience is a liability
  - limited exposure to use-cases
  - a bias for verbose and fully explicit languages
- why our inexperience is an asset
  - we are prototypical EHC clients
  - we haven't forgotten what is difficult or non-obvious

Experimenting with Language Design

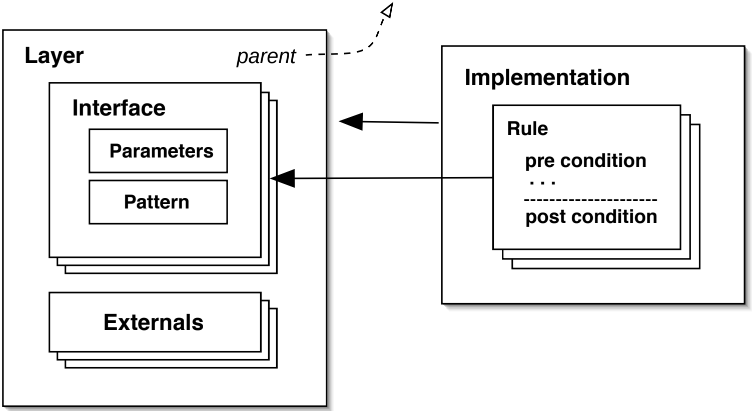
Critiquing Ourselves

- why our inexperience is a liability
  - limited exposure to use-cases
  - a bias for verbose and fully explicit languages
- why our inexperience is an asset
  - we are prototypical EHC clients
  - we haven't forgotten what is difficult or non-obvious

Our Solution

- restrict big changes to syntax/storage
  - changes must address new requirements
- preserve basic underlying Rule structure
  - only extend basic structure
  - only to solve new problems

Language Abstractions



Concrete Syntax


```
layer Equational extends Base
  interface Expr
    params
      in     e      : Expr   (node)
      in     valGam : ValGam
      in     tyGam  : TyGam
      in     kiGam  : KiGam
      inout ty      : Ty     (retain)
    uses
      out    err    : Err    (private)
    pattern "kiGam ; tyGam ; valGam :- e : ty"
```

Concrete Syntax

```
implementation of Equational
 rule IConst implements Expr
   post Expr.R = "kiGam ; tyGam ;
                  valGam :- int : tyInt"
```

or

```
implementation of Equational
 rule IConst implements Expr
   post Expr.R
         | ty = "tyInt"
```

front-end processing

*Compiler> compile "Equational"

Processing Layers

1. check name matches file
2. recursively load layer hierarchy
3. check parameter access between layers

For each layer

1. check name matches file
2. check lhs parameter access
3. check for duplicate rule names
4. merge with implementation below us

Problems and Solutions

- How to support new domain specific information
  - identifying symmetric patterns
- How to merge layers where order is important
  - Wan't an issue for AG, but is for Haskell
- Keeping information local to its use
  - where clause in Haskell
  - keeping definitions local in documentation

Using The Language

- The front-end is functional
- Now we want to use it
- Of course, we gave up a lot of functionality...

Generating Unification...

Defining the problem

Problem definition

We want to extend Ruler so that it can generate unification code

- Until now only syntax directed rules
- A single constructor determines which rule to apply
- Ruler, like AG, can only generate syntax directed code
- Unification cannot be expressed via syntax directed rules

Unifcation rules

$$\boxed{\vdash^{\cong} \sigma_l \cong \sigma_r : \sigma}$$

$$\frac{}{\vdash^{\cong} \Box \cong \sigma : \sigma}\ \mathrm{M.ANY.L}_K \qquad \frac{}{\vdash^{\cong} \sigma \cong \Box : \sigma}\ \mathrm{M.ANY.R}_K$$

$$\frac{I_1 \equiv I_2}{\vdash^{\cong} I_1 \cong I_2 : I_2}\ \mathrm{M.CON}_K$$

$$\frac{\begin{array}{c}\vdash^{\cong} \sigma_2^a \cong \sigma_1^a : \sigma_a \\ \vdash^{\cong} \sigma_1^r \cong \sigma_2^r : \sigma_r\end{array}}{\vdash^{\cong} \sigma_1^a \to \sigma_1^r \cong \sigma_2^a \to \sigma_2^r : \sigma_a \to \sigma_r}\ \mathrm{M.ARROW}_K$$

$$\frac{\begin{array}{c}\vdash^{\cong} \sigma_1^l \cong \sigma_2^l : \sigma_l \\ \vdash^{\cong} \sigma_1^r \cong \sigma_2^r : \sigma_r\end{array}}{\vdash^{\cong} (\sigma_1^l, \sigma_1^r) \cong (\sigma_2^l, \sigma_2^r) : (\sigma_l, \sigma_r)}\ \mathrm{M.PROD}_K$$

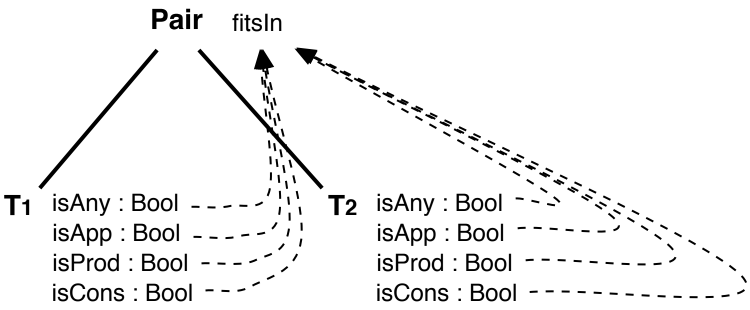
Unification code specification

Code for Unification

```
fitsIn :: Ty → Ty → Ty
fitsIn ty1 ty2
  = f ty1 ty2
    where
      f   Ty_Any          t2       = t2
      f   t1              Ty_Any   = t1
      f   t1@(Ty_Con s1)
          t2@(Ty_Con s2)
          | s1 ≡ s2                = t2
      f   t1@(Ty_App (Ty_App (Ty_Con c1) ta1) tr1)
          t2@(Ty_App (Ty_App (Ty_Con c2) ta2) tr2)
          | hsnIsArrow c1 ∧ c1 ≡ c2
            = Ty_App (Ty_App (Ty_Con c2) (f ta2 ta1) ) (f tr1 tr2)
      f   t1@(Ty_App tf1 ta1)
          t2@(Ty_App tf2 ta2)
            = Ty_App (f tf1 tf2) (f ta1 ta2)
      f   t1  t2                   = undefined
```

Possible targets for generation

1. Try to fit non-syntax directed code in AG
   - Combine the the types that have to be unified.
   - Use synthesized attributes to determine the synthesized type
   - Direct integration with Ruler
2. Generate Haskell directly
   - A more 'natural fit'
   - A new target language for Ruler



- An attribute to recognize each constructor
- Attributes for values of each constructor
- Attributes for each nested pattern

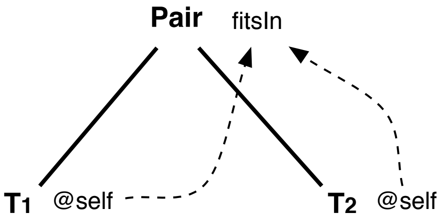
Unification in AG - Pattern attributes


```
DATA Pair
  | Pair   lty : Ty rty : Ty

DATA Ty
  | Ty_Any
  | Ty_Con    x : {String}
  | Ty_App lty : Ty rty : Ty

ATTR Ty [ | | self   : SELF ]
ATTR Ty [ | | isAny : Bool ]
ATTR Ty [ | | isCon : Bool conName : {String} ]
ATTR Ty [ | | isApp : Bool appLeft : Ty appRight : Ty ]
ATTR Pair [ | | ty : Ty ]
SEM Pair
  | Pair   lhs·ty =    let match  | @lty·isAny   = @rty·self
                                  | @rty·isAny   = @lty·self
                                  | @lty·isCon ∧ @rty·isCon ∧
                                    @lty·conName ≡ @rty·conName
                                                 = @rty·self
                                  | otherwise    = undefined
                       in  match
```

Unification in AG - Matching on SELF attributes



- Only need to add one self attribute
- Pattern match on the child nodes
- Basicly just a normal haskell function

Unification in AG - Matching on SELF attributes


```
ATTR Pair [ | | ty : Ty ]
SEM Pair
  | Pair   lhs·ty =    let match  (Ty_Ty_Any) rty = rty
                           match  lty (Ty_Ty_Any) = lty
                           match  (Ty_Ty_Con x) rty@(Ty_Ty_Con y)
                                    | x ≡ y        = rty
                           match  lty rty          = undefined
                         in  match @lty·self @rty·self

ATTR Ty [ | | self   : SELF ]
```

But this is just 'ugly' Haskell!

Mapping rules to Haskell (1)

$$\frac{I_1 \equiv I_2}{\vdash^{\cong} I_1 \cong I_2 : I_2}\ \mathrm{M.CON}_K$$

```
implementation of Base
  rule unifyCon implements unify
    post unify·R
      | guard = "lty == rty"
      | lty   = "Ty_Con x"
      | rty   = "Ty_Con y"
      | res   = "rty"

unify lty@(Ty_Con x) rty@(Ty_Con y)
  | lty ≡ rty  = rty
```

Mapping rules to Haskell (2)

$$\frac{\begin{array}{c}\vdash^{\cong} \sigma_1^l \cong \sigma_2^l : \sigma_l \\ \vdash^{\cong} \sigma_1^r \cong \sigma_2^r : \sigma_r\end{array}}{\vdash^{\cong} (\sigma_1^l, \sigma_1^r) \cong (\sigma_2^l, \sigma_2^r) : (\sigma_l, \sigma_r)} \ \text{M.PROD}_K$$

```
rule unifyProd implements unify
  pre unify·FST
    | lty = "lty_1"  | rty = "lty_2"  | res = "lty_res"
  pre unify·SND
    | lty = "rty_1"  | rty = "rty_2"  | res = "rty_res"
  post unify·R
    | lty = "Ty_App lty_1 rty_1"
    | rty = "Ty_App lty_2 rty_2"
    | res = "Ty_App lty_res rty_res"
```

```
unify lty@(Ty_App lty_1 rty_1) rty@(Ty_App lty_2 rty_2)  =
  let
    ( lty_res ) = unify (lty_1) (lty_2)
    ( rty_res ) = unify (rty_1) (rty_2)
  in ( Ty_App lty_res rty_res )
```

Mapping rules to Haskell (3)

1. *node*, *in* and *inout* parameters of the post-judgment map to input of the function
2. *inout* and *out* parameters of the post-judgment map to the output tuple of the function
3. pre-judgments map to let bindings
    1. *node*, *in* and *inout* map to arguments of the function call
    2. *inout* and *out* form the resulting binding
4. the special parameter *guard* maps to a guard expression

Extending the rules - Error handling (1)

- The rules abstract away from details like error handling
- These 'details' are thus not generated
- Should we extend our rules with error handling, or do these things do not belong in our type rules?
- Error matches 'all other' cases, so now order matters.

$$\frac{}{\vdash^{\cong} \sigma_l \cong \sigma_r : \mathit{error}}\ \mathrm{M.ERR}_K$$

Extending the rules - Error handling (2)


```
implementation of Base
  rule unifyCon implements unify
    post unify·R
      | guard = "lty == rty"
      | lty   = "Ty_Con x"   | rty   = "Ty_Con y"
      | res   = "emptyFO {foTy = rty}"
  rule unifyProd implements unify
    pre unify·FST
      | lty = "lty_1"  | rty = "lty_2"  | res = "lty_res"
    pre unify·SND
      | lty = "rty_1"  | rty = "rty_2"  | res = "rty_res"
    post unify·R
      | lty = "Ty_App lty_1 rty_1"
      | rty = "Ty_App lty_2 rty_2"
      | res = "foldr1 (\fo1 fo2 -> if foHasErrs fo1 then fo1 else fo2)
                [lty_res,rty_res,emptyFO {foTy = Ty_App lty_res rty_res}]"
  rule unifyErr implements unify
    post unify·R
      | lty = "lty"  | rty = "rty"
      | res = "emptyFO {foErrL = [Err_UnifyClash lty rty]}"
```

Extending the rules - Error handling (3)

- Judgments not very abstract anymore; broken up Haskell functions
- Language should provide options to hide these details
  - Functions can be hidden behind other layers or local declarations
  - Functions can be hidden inside templates, specific for the output

Extending the rules - New versions (1)

We have only described the generation of the EH2 unification algorithm. The later EH unification algorithms are much more complex (e.g. Quantified types, records)

```
fitsIn :: FIOpts → FIEnv → UID → Ty → Ty → FIOut
fitsIn opts env uniq ty1 ty2
  =  f (emptyFI {fiUniq = uniq, fiFIOpts = opts}) ty1 ty2
  where
    res fi t
      =  emptyFO  { foUniq = fiUniq fi, foTy = t
                  , foAppSpineL = asGamLookup appSpineGam (tyConNm t)}
    f fi t1@(Ty_Con s1)         t2@(Ty_Con s2)
         | s1 ≡ s2             = res fi t2
    f fi t1@(Ty_App tf1 ta1)    t2@(Ty_App tf2 ta2)
      = manyFO [ffo,afo,foCmbApp ffo afo]
      where  ffo   = f fi tf1 tf2
             fs    = foCnstr ffo
             (as:_) = foAppSpineL ffo
             fi'   = fi  { fiFIOpts  = asFIO as · fioSwapCoCo (asCoCo as)
                                                · fiFIOpts $ fi
                         , fiUniq    = foUniq ffo }
             afo   = f fi' (fs |=> ta1) (fs |=> ta2)
```

Extending the rules - New versions (2)

Same problem as extending with error handling:

- Helper functions (`appSpineGam`, `fioSwapCoCo`, `foCmbApp`, ...) clutter; appear all over the rules
- We can 'hide' them behind local definitions or relations
- But still references need to be made to this hidden definition in the rules and thus our rendered documentation will be cluttered with it too.
- We are only re-arraning Haskell to comply with the ruler structure. Is this enough abstraction?

Output control

- Decoupled output from domain language
- Currently Haskell is generated from an AG implementation, but this is not very flexible for a user of the DSL
- We would like to generate output by using a template language for flexibility of output
  - Possbile targets can be: LaTeX, AG, Haskell, HOL ...

Integrating in Ruler

- Language does not differ much compared to ruler
    - Language is layered in a different way and view-logic is removed
    - Language adds local definitions; translate to relations
    - Language adds private parameters; not present anymore in Ruler AST
- Same techniques can be used to generate Haskell from Ruler AST
- Defining Haskell output by `judgeuse`, to comply to the ruler framework, requires some extra thought.

Conclusion

Conclusion 1/3

- Reflections on the project:
  - problem space very big and difficult
  - specific task (generating unification code)
  - not much middle ground for meaningful contributions
- On Generating Haskell via Ruler
  - tedious, but eminently doable
  - you can generate just about anything with Ruler
  - but what does that get you? Is result concise? maintainable?

Conclusion 2/3

Question: What is the relative value of code vs. documentation

- are you willing to give up documentation quality in order to achieve more concise generative logic?
- in order to move squiggly line, you must sacrifice one or the other

You can't have your cake and eat it to

Conclusion 3/3

concrete suggestions for Ruler

1. move language away from AG
2. explicit extensibility hooks
3. local definitions
4. address software engineering concerns
   - decomposition of source layers
   - explicit constraints declared in layers
   - separate out rendering logic
     - maybe use css approach to achieve?

Questions