UHC: Coping with Compiler Complexity

Atze Dijkstra

FLOPS 2016

UHC: Coping with Compiler Complexity

UHC: Utrecht Haskell Compiler

- Haskell2010 implementation
- Primarily intended for play & experimentation: higher ranked types, partial type signatures, generic deriving, local instances, java(script) backend, ...
- Started as 'proof' of usefulness of parser combinators, attribute grammar system
- Inspiration for further tooling

UHC: Coping with Compiler *Complexity*

Complexity: dimensions of complexity

- From specification to implementation
- From deterministic to non-deterministic
- From few to many (combined) language features
- From small to large compiler input

Approach: keep specifications as simple as possible

- Compositionality as much/far as possible
- DSL (Domain Specific Language)
- Generate (e.g. boilerplate code)
- Consistency between specifications

Underlying desire

- Write compiler using reusable compiler idiom

UHC: Coping with Compiler Complexity

Assumptions

- Basic Haskell
- Type systems (Hindley-Milner in particular)

Take home: UHC as example of

- The 'art' of putting theory into practice (by implementing)
- Support for this 'art': DSLs, tooling, engineering, ...

Today's plan

- Story thread: write a smallish UHC

Today's plan

- Story thread: write a smallish UHC
- Story 'algorithm': While time left interleave
  - Write or design a compiler fragment
  - Observe/reflect upon problematic issues and solutions
  - Explore routes taken by/for UHC

Demo language

- Basic functional language

$$
\begin{array}{lll}
e ::= & i & \text{-- base: int} \\
| & n & \text{-- name reference} \\
| & e\ e & \text{-- application} \\
| & \lambda n.e & \text{-- abstraction} \\
| & \mathbf{let}\ n = e\ \mathbf{in}\ e & \text{-- let binding}
\end{array}
$$

- With Hindley-Milner type system (HM)

$$
\begin{array}{l}
\tau ::= \mathit{Int} \mid \tau \rightarrow \tau \mid \alpha \\
\sigma ::= \tau \mid \forall\ \alpha.\sigma \\
\Gamma ::= \Gamma\ [n \mapsto \sigma] \mid \varepsilon
\end{array}
$$

Demo language

- Language example

let *id* $= \lambda x.x$ **in**
let *id2* $=$ *id id id id* **in**
id id2 (*id* 5)

- Desired output and results
  - Some analysis for some semantics: type
  - Some error reporting: name errors, type errors
  - Some code generation using analysis results: pretty printing

Demo language

- Compiler output

```
-- PP
let id : forall a. a -> (a)
       = \x.x in
let id2 : forall a. a -> (a)
        = id (id) (id) (id) in
id (id2) (id (5))
-- Ty
Int
```

Demo language

- Language example with a name error

$$\textbf{let } \mathit{id} = \lambda x.y \textbf{ in}$$
$$f\ 4$$

- Compiler output

```
-- PP
let id : forall a. a -> (ERR)
       = \x.y{- Not introduced: y -} in
f{- Not introduced: f -} (4)
-- Errors
Not introduced: y
Not introduced: f
```

Demo language

- Language example with typing error (Y fixpoint combinator)

$$\lambda w.(\lambda x.w\ (x\ x))\ (\lambda x.w\ (x\ x))$$

- Compiler output

```
-- PP
\w.\x.w (x (x){- Occurs: v2
                         v2 -> (v5) -}
        ) (\x.w (x (x){- Occurs: v8
                                 v8 -> (v11) -}
                )
          )
-- Errors
Occurs: v2
        v2 -> (v5)
Occurs: v8
        v8 -> (v11)
```

Writing a compiler for the demo language

What needs to be done?

- Specify the semantics (here: only type, not operational)

Type semantics, declarative

HM type system, declarative specification for type system

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

$$\frac{n \mapsto \sigma \in \Gamma}{\Gamma \vdash n : \sigma}\ \mathrm{VAR}_D \qquad \frac{\begin{array}{c}\Gamma[n \mapsto \sigma] \vdash e_b : \tau \\ \Gamma \vdash e : \sigma\end{array}}{\Gamma \vdash \mathbf{let}\ n = e\ \mathbf{in}\ e_b : \tau}\ \mathrm{LET}_D \qquad \frac{\Gamma[n \mapsto \tau_a] \vdash e : \tau_r}{\Gamma \vdash \lambda n.e : \tau_a \rightarrow \tau_r}\ \mathrm{ABS}_D$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \tau_a \\ \Gamma \vdash f : \tau_a \rightarrow \tau_r\end{array}}{\Gamma \vdash f\ \ a : \tau_r}\ \mathrm{APP}_D$$

$+$ generalization, instantiation ...

Type semantics, declarative

Generalization, instantiation

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

$$\frac{\begin{array}{c}\alpha \notin \mathit{ftv}\ (\Gamma) \\ \Gamma \vdash e : \tau\end{array}}{\Gamma \vdash e : \forall\ \alpha.\tau}\ \mathrm{GEN}_D \qquad \frac{\begin{array}{c}\alpha \text{ fresh} \\ \Gamma \vdash e : \forall\ \beta.\tau\end{array}}{\Gamma \vdash e : [\beta \mapsto \alpha]\ \tau}\ \mathrm{INST}_D$$

Type semantics, declarative

Generalization, instantiation

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

$$\frac{\begin{array}{c}\alpha \notin \mathit{ftv}\ (\Gamma) \\ \Gamma \vdash e : \tau\end{array}}{\Gamma \vdash e : \forall\ \alpha.\tau}\ \mathrm{GEN}_D \qquad \frac{\begin{array}{c}\alpha \text{ fresh} \\ \Gamma \vdash e : \forall\ \beta.\tau\end{array}}{\Gamma \vdash e : [\beta \mapsto \alpha]\ \tau}\ \mathrm{INST}_D$$

Can we directly implement this?

- When to apply rules?
- How to solve equations implied implicitly by multiple occurrences of meta variables (like τ_a)?

Type semantics, syntax directed

For HM this is well known[1]

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

$$\frac{\begin{array}{c} n \mapsto \sigma \in \Gamma \\ \vdash^{inst} \sigma : \tau \end{array}}{\Gamma \vdash n : \tau} \text{VAR}_{SD} \qquad \frac{\begin{array}{c} \Gamma \vdash^{gen} \tau_n : \sigma \\ \Gamma \vdash e : \tau_n \\ \Gamma[n \mapsto \sigma] \vdash e_b : \tau \end{array}}{\Gamma \vdash \mathbf{let}\ n = e\ \mathbf{in}\ e_b : \tau} \text{LET}_{SD}$$

Add and allow for computational direction

- When to apply rules? Make it syntax directed
- How to solve equations? Algorithm W (for example), unification, type variables & substitution

For an implementation design, algorithm, and engineering decisions have to be made

[1] https://en.wikipedia.org/wiki/Hindley-Milner_type_system

Type semantics, algorithmic

Algorithm W

$$\boxed{\theta_i; \Gamma \vdash e : \tau \leadsto \theta_o}$$

$$\frac{\begin{array}{c} n \mapsto \sigma \in \Gamma \\ \alpha \text{ fresh} \\ \alpha \vdash^{inst} \theta\ \sigma : \tau \end{array}}{\theta; \Gamma \vdash n : \tau \leadsto \theta} \text{VAR}_A \qquad \frac{\begin{array}{c} \theta_n \Gamma \vdash^{gen} \tau_n : \sigma \\ \theta; \Gamma \vdash e : \tau_n \leadsto \theta_n \\ \theta_n; \Gamma[n \mapsto \sigma] \vdash e_b : \tau \leadsto \theta_e \end{array}}{\theta; \Gamma \vdash \textbf{let}\ n = e\ \textbf{in}\ e_b : \tau \leadsto \theta_e} \text{LET}_A$$

$$\frac{\begin{array}{c} \alpha \text{ fresh} \\ \theta; \Gamma[n \mapsto \alpha] \vdash e : \tau_r \leadsto \theta_e \end{array}}{\theta; \Gamma \vdash \lambda n.e : \theta_e \alpha \to \tau_r \leadsto \theta_e} \text{ABS}_A \qquad \frac{\begin{array}{c} \alpha \text{ fresh} \\ \theta; \Gamma \vdash f : \tau_f \leadsto \theta_f \\ \theta_a \tau_f \equiv \tau_a \to \alpha \leadsto \theta_u \\ \theta_f; \Gamma \vdash a : \tau_a \leadsto \theta_a \end{array}}{\theta; \Gamma \vdash f\ a : \theta_u \alpha \leadsto \theta_u \theta_a} \text{APP}_A$$

Algorithmic == computable

Ready to implement...

Implementation

Haskell implementation of Algorithm W

$$\begin{array}{l} algoW :: Env \rightarrow (Subst, Int) \rightarrow Exp \rightarrow (Ty, (Subst, Int), Err) \\ \qquad\qquad\quad \Gamma \quad ; \quad \theta \qquad\qquad\quad \vdash e \quad : \quad \tau \leadsto \theta \end{array}$$

Type		
Env	Γ	scoped environment, mapping from identifiers to types
Exp	e	term AST
Ty	τ	types
Int		unique number generation, for fresh type variables
Subst	θ	substitution, mapping type variables to types
Err		errors

Syntax directed rules now allow pattern match on *Exp* to deterministically choose the right rule

Implementation, purely functional

```
algoW :: Env → (Subst, Int) → Exp → (Ty, (Subst, Int), Err)
algoW env st (Exp_App f a) =
    (s4 $θ v, (s4, uniq st3), ef ++ ea ++ eu)
  where
    (tf, st1, ef) = algoW env st   f   -- recurse for function
    (ta, st2, ea) = algoW env st1 a    -- recurse for argument
    (v,  st3)     = fresh  st2         -- new fresh type var
    s3            = subst st3          -- get current subst
    (su, eu)      = unify  (s3 $θ tf)  -- apply subst, unify
                           (TyArr ta v)
    s4            = su $θ s3

      -- helper functions
     fresh (s, u) = (mkTyVar u, (s, u + 1))
     subst = fst
     uniq = snd
```

Implementation, purely functional

Utility types (for reference)

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{TyVar} & = & \mathit{TV}\ \mathit{Int}\ ... \\
\textbf{data}\ \mathit{Ty} & = & \mathit{TyInt} \mid \mathit{TyVar}\ \mathit{TyVar} \mid \mathit{TyArr}\ \mathit{Ty}\ \mathit{Ty} \mid ... \\
\textbf{data}\ \mathit{Exp} & = & \mathit{Exp\_App}\ \mathit{Exp}\ \mathit{Exp} \mid ... \\
\textbf{type}\ \mathit{Env} & = & [(\mathit{String}, \mathit{Ty})] \\
\textbf{type}\ \mathit{Subst} & = & [(\mathit{TyVar}, \mathit{Ty})] \\
\multicolumn{3}{l}{\textbf{class}\ \mathit{Substitutable}\ x\ \textbf{where}\ \{(\$_\theta) :: \mathit{Subst} \to x \to x\}}
\end{array}
$$

Moments of reflection

What have we done so far?

- Started with concise declarative specification, which
  - Via algorithmic version
  - Led to functional implementation
  - For which Haskell itself is already a good tool
- But...
  - Many low level details crept in while still many other details are left out
  - An actual implementation must do more than just specify semantics: parser, scanner, ...
  - Implementation for our example: approx 300 LOC (without comment)
- And...
  - We are not done yet...
  - Pretty printing (as kind of code generation)

Moments of reflection

Can we *scale up*?

- The things/aspects $a_1, a_2, \ldots$ we want to specify/compute per language construct?
- The number of language features/constructs $f_1, f_2, \ldots$?

Moments of reflection

Can we *scale up*?

- The things/aspects $a_1, a_2, \ldots$ we want to specify/compute per language construct?
- The number of language features/constructs $f_1, f_2, \ldots$?

The *ideal* would be to be able to specify independently

- Aspects a_1 and a_2 and then combine them with some operator $\oplus$ into $a_1 \ \oplus \ a_2$
- Language features f_1 and f_2 and then combine them with some operator $\otimes$ into $f_1 \ \otimes \ f_2$

The *reality* is that aspects, features, and their combination usually must be 'aware' of each other to some degree.

Related to the Expression Problem (later more about that)

Implementation, pretty printing

Pretty printing: extra aspect for existing language constructs

- The ideal would be to define pretty printing independently of algoW:

$$
\begin{array}{l}
pp :: Exp \rightarrow Doc \\
pp\ (Exp\_App\ f\ a\ \ ) = pp\ f \rangle|\langle\ pp\ a \\
pp\ (Exp\_Let\ n\ e\ b) = \texttt{"let"} \rangle|\langle\ pp\ n \rangle|\langle \texttt{"="} \rangle|\langle\ pp\ e \rangle|\langle \\
\qquad\qquad\qquad\qquad\qquad \texttt{"in"}\ \ \rangle|\langle\ pp\ b \\
\textbf{data}\ Doc = \ldots \qquad\qquad\qquad \text{-- pretty print document} \\
(\rangle|\langle) :: Doc \rightarrow Doc \rightarrow Doc \qquad \text{-- combine horizontally}
\end{array}
$$

- This works well if *pp* does not use info encapsulated in *algoW*

Implementation, pretty printing

For independent aspects, this leads to nanopasses[2][3] in compiler: small, maintainable, isolates solution for single (independent) problem

- Used in UHC for (e.g.) transformations of intermediate representations
- Can be inefficient, boilerplate overhead

[2] "A Nanopass Framework for Commercial Compiler Development", 2013

[3] "Scrap your boilerplate: a practical design pattern for generic programming", 2003

Implementation, pretty printing

The reality (complexity) here is that *pp* and *algoW* are dependent: pretty printing uses the inferred type (*te*) and error messages (*eu*) of *algoW*

$$
\begin{array}{l}
pp :: Exp \rightarrow Doc \\
pp\ (Exp\_App\ f\ a\ \ ) = pp\ f \rangle|\langle\ pp\ a\ \rangle|\langle pp\ eu \\
pp\ (Exp\_Let\ n\ e\ b) = \texttt{"let"} \rangle|\langle\ pp\ n\ \rangle|\langle \texttt{":"} \rangle|\langle\ pp\ te \rangle|\langle \\
\qquad\qquad\qquad\qquad\qquad\qquad \rangle|\langle\ \texttt{"="} \rangle|\langle\ pp\ e \rangle|\langle \\
\qquad\qquad\qquad\qquad\quad \texttt{"in"}\ \ \rangle|\langle\ pp\ b
\end{array}
$$

Aspects type and pretty printing are not independent!
In general, more complex analyses are dependent.

Implementation

Solution 1: refactor in the 'wrong' direction by adding *pp* to *algoW*:

$$
\begin{array}{l}
algoW :: Env \rightarrow (Subst, Int) \rightarrow Exp \rightarrow (Ty, (Subst, Int), Err\ , Doc) \\
algoW\ env\ st\ (Exp\_App\ f\ a) = \\
\qquad (s_4\ \$_\theta\ v, (s_4, uniq\ st3), ef + \!\! + ea + \!\! + eu\ , pf\ \rangle|\langle\ pa\ \rangle|\langle\ pp\ eu) \\
\quad \textbf{where} \\
\qquad (tf, st1, ef\ , pf) = algoW\ env\ st\ \ f \\
\qquad (ta, st2, ea\ , pa) = algoW\ env\ st1\ a \\
\qquad (su, eu) \qquad\quad\ = unify\ (s_3\ \$_\theta\ tf)\ (TyArr\ ta\ v) \\
\qquad \ldots
\end{array}
$$

- Adding an aspect implies a manual overhaul of boilerplate code

Attribute grammars

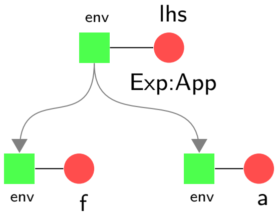
Solution 2 (taken by UHC): DSL for computations over ASTs

- Each individual computation expressed as attribute
- Specification for each attribute can be separately described, combined later
- A compiler (UUAGC) glues separate specifications, generating functional program, including boilerplate

Allows thinking in terms of attributes associated with parent and children in AST, defining data flow fragments

Attribute grammars

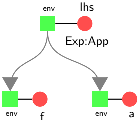
- All computation is defined in terms of attributes associated with parent and children
- Attributes are defined in terms of other attributes, thus specifying small dataflow fragments for parent and children



- Inherited attributes 'travel' downwards (from root to leaves), synthesized upwards

Attribute grammars

Environment

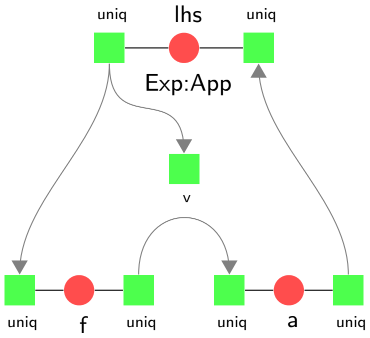



```
data Exp                          -- AST
    | App f : Exp
          a : Exp
attr Exp [env : Env ||]           -- inherited, from root to leaves
sem Exp
    | App f.env = @lhs.env        -- copied downwards
          a.env = @lhs.env
```

Boilerplate copying (for *env*) usually omitted and generated automatically

Attribute grammars

Unique number generation



$\textbf{attr}\ \mathit{Exp}\ [\,|\ \mathit{uniq} : \mathit{Int}\ |\,]$ -- synthesized + inherited = state

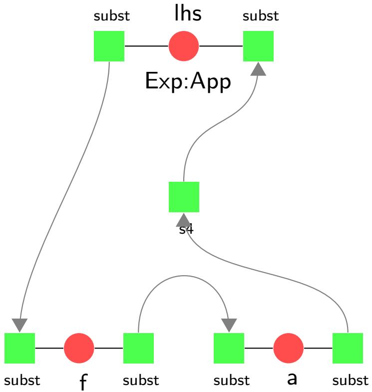
$\textbf{sem}\ \mathit{Exp}$

$\quad |\ \mathit{App}\ \textbf{loc}.v \quad = \mathit{mkTyVar}\ @\textbf{lhs}.\mathit{uniq}$

$\qquad\quad f\ \ .\mathit{uniq} = @\textbf{lhs}.\mathit{uniq} + 1$

Attribute grammars

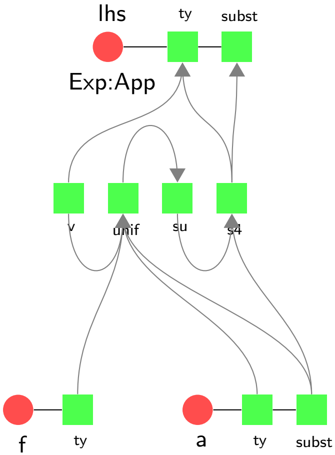
Substitution



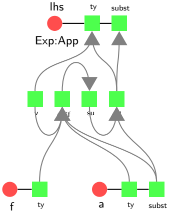
attr *Exp* [| *subst* : *Subst* |]

Attribute grammars

Substitution, type, unification



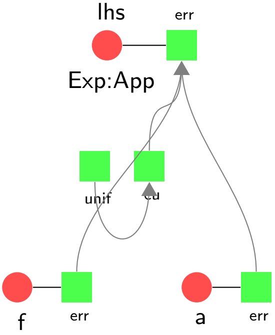
Attribute grammars



$$
\begin{array}{l}
\textbf{attr}\ \mathit{Exp}\ [\,||\ \mathit{ty} : \mathit{Ty}\,] \quad \text{-- synthesized} \\
\textbf{sem}\ \mathit{Exp} \\
\quad |\ \mathit{App}\ (\textbf{loc}.\mathit{su}, \textbf{loc}.\mathit{eu}) = \mathit{unify}\ (@a.\mathit{subst}\ \$_\theta\ \ @f.\mathit{ty}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{TyArr}\ @a.\mathit{ty}\ @v) \\
\qquad\qquad \textbf{loc}.s_4 \qquad\qquad\ = @\mathit{su}\ \$_\theta\ \ @a.\mathit{subst} \\
\qquad\qquad \textbf{lhs}.\mathit{ty} \qquad\qquad = @s_4\ \$_\theta\ \ @v \\
\qquad\qquad\quad .\mathit{subst} \qquad\quad\ = @s_4
\end{array}
$$

Attribute grammars

Error collecting



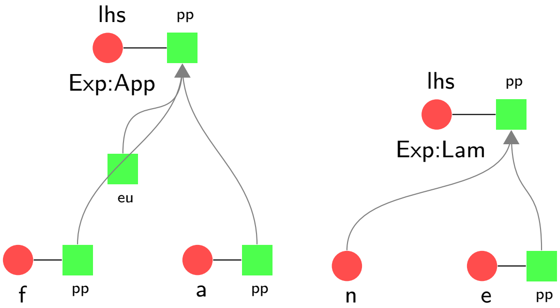
$\textbf{attr}\ \mathit{Exp}\ [\,||\ \mathit{err} : \mathit{Err}\,]$

$\textbf{sem}\ \mathit{Exp}$

$\quad |\ \mathit{App}\ \textbf{lhs}.\mathit{err} = @\mathit{eu} +\!\!+\ @f.\mathit{err} +\!\!+\ @a.\mathit{err}$

Attribute grammars

Similarly: pretty printing



attr *Exp* [|| *pp* : *Doc*]

sem *Exp*

  | *App* **lhs**.*pp* = @*f*.*pp* >|< @*a*.*pp* >|< *ppErr* @*eu*

  | *Lam* **lhs**.*pp* = "\\" >|< @*n* >|< "." >|< @*e*.*pp*

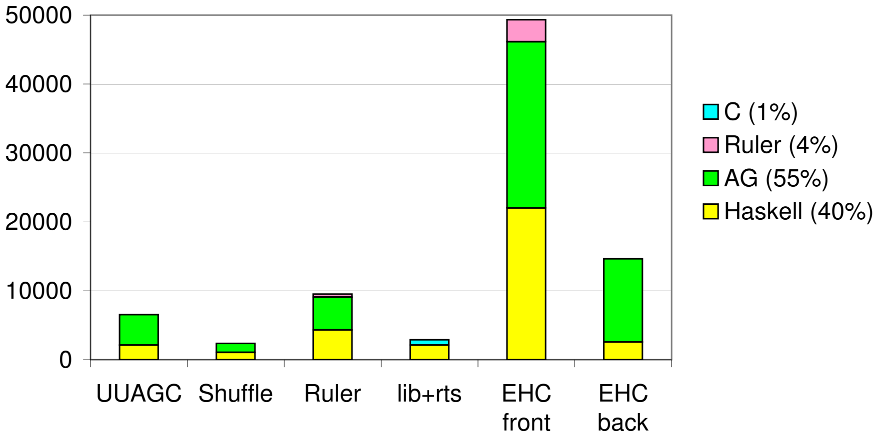
Moments of reflection

What have we done so far?

- Started with concise declarative specification, which
  - Via algorithmic version
  - Led to Attribute Grammar implementation
  - Where all aspects can be described independently even though there are dependencies
  - Generates Haskell using UUAGC (UU AG compiler)
  - Combines separate specifications, generates boilerplate code
- This works very well!
  - In UHC, almost all functionality involving trees is expressed using AGs
  - Integrates with Haskell ecosystem

Moments of reflection

Example, statistics for UHC: use of UUAGC (and other languages)



Moments of reflection

Drawbacks

- We still have to know about dependencies
  - Luckily, UUAGC gives feedback about dependency related errors
- UUAGC is a preprocessor: checks for AG specifics, but not Haskell specifics of embedded Haskell code
  - We get Haskell errors too late, only when generated code is compiled
- One has to learn a separate language
  - And the additional tooling etc.
- Implementation done manually, what guarantees do we have about consistency with type rules?
  - Maintaining two copies of the same is a nightmare

Moments of reflection

Alternative approach: plumbing via underlying implementation language

- A monadic interface for Algorithm W

```
algoW
  :: ( MonadReader Env m             -- scoped name mapping
     , MonadState (Subst, Int) m     -- global state
     , MonadError Err m              -- error/exception
     , MonadWriter Doc m             -- pretty printing
     ) ⇒ Exp → m Ty
```

- Or, more recently: extensible effects, data types a la carte
  - Monadic bind corresponds to higher order attributes

Moments of reflection

Common mechanisms (with their drawbacks)

- Layers are combined, requires scheduling overhead
  - Running overhead
  - Crossing boundaries overhead (i.e. dependencies between different aspects)
  - Layers/aspects indexed by types
- Each aspect is a separate layer of functionality indexed by a type
  - Type must be unique amongst aspects

Extensible building of languages can also be directly modelled in host language Haskell...

But...

Moments of reflection

Common mechanisms (with their drawbacks)

- Layers are combined, requires scheduling overhead
  - Running overhead
  - Crossing boundaries overhead (i.e. dependencies between different aspects)
  - Layers/aspects indexed by types
- Each aspect is a separate layer of functionality indexed by a type
  - Type must be unique amongst aspects

Extensible building of languages can also be directly modelled in host language Haskell...

But... in both cases comes with runtime overhead and need to be aware of (type level) glueing

Language variants: *AspectAG*

Type safe embedding in Haskell of parsers, syntax macros, AST definition, Attribute Grammars[4] as knittable fragments

Comparison

- Approach 1: Explicit recursion
- Approach 2: AG preprocessor

	easy to add attributes	easy to add alternatives	easy to read	checks well formness	common patterns	compiles in GHC
1	−	−	−	+	±	+
2	+	+	+	+	+	−
3	+	+	+	+	+	+

- Approach 3: AspectAG
  - Haskell 98
  - MultiParamTypeClasses, FunctionalDependencies, FlexibleContexts, FlexibleInstances, UndecidableInstances, ExistentialQuantification, EmptyDataDecls, Rank2Types and TypeSynonymInstances.

[4] "First Class Syntax, Semantics, and Their Composition", 2013

Language variants: *AspectAG*

Example: Oberon compiler (LDTA challenge[5]) implementation demonstrates dealing with expression problem

What about efficiency?

	...	checks well formness	common patterns	compiles in GHC	efficiency
1	...	+	±	+	+
2	...	+	+	−	+
3	...	+	+	+	−

Marcos Viera, Doaitse Swierstra, Wouter Swierstra | Attribute Grammars Fly First Class

Elegant though inefficient: heavy use of type level programming, template haskell, resulting code difficult to optimize

[5] *LDTA 2011 tool challenge description and problem set*, 2011

Moments of reflection

Alternative approach: visitor pattern from the object-oriented world

- Must know about dependencies between results from visits
- (Side effects)

Moments of reflection

Core idea of Attribute Grammars, UUAGC in particular

- Restricted form of functional programming (catamorphisms)
- Declaratively specify dependent computations (via attributes)
- Is a preprocessor/compiler,
- UUAGC as a preprocessor allows global dependency analysis and tailormade codegen, avoids overhead of monad (and similar) approaches

UUAGC computes how the logistics/scheduling can be done

- No need to worry about this as a programmer
- Facilitates easy modification (which we wanted for experimentation)

Implementation: multiple visits/passes

Example: need for visits/passes

- Small change in running example: pretty print of lambda expression should include inferred type
- Input

$$((\lambda x.x)\ 5)$$

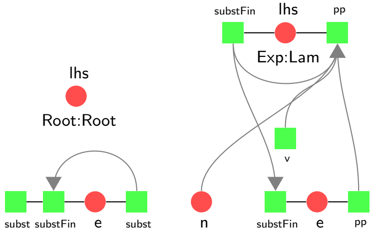
- Should pretty print to

```
(\x:Int.x) (5)
```

- Required type: type variable + substitution

Substitution *subst* only known after type inference of whole program is done: pass final *subst* (as *substFin*) from *Root* downwards as 2nd pass/visit

Implementation: multiple visits/passes



attr *Exp* [*substFin* : *Subst* ||]

sem *Root*

  | *Root e*. *substFin* = @*e*.*subst*

sem *Exp*

  | *Lam* **lhs**.*pp* = "\\" ⟩|⟨ @*n*
  ⟩|⟨ ":" ⟩|⟨ *pp* (@**lhs**.*substFin* $\$_\theta$ @*v*)
  ⟩|⟨ "." ⟩|⟨ @*e*.*pp*

Implementation: multiple visits/passes

UUAGC generates 2 visits (don't try to understand it all...)

```
type T_Exp = Env → Subst → Int → (Subst, Ty, Int, T_Exp_1)      -- 1st visit (type)
type T_Exp_1 = Subst → (Err, PP_Doc)                             -- 2nd visit (type)
sem_Exp_App :: T_Exp → T_Exp → T_Exp                             -- 1st visit
sem_Exp_App f_ a_ = λ_lhsIenv _lhsIsubst _lhsIuniq →
  let (_fOuniq, _uniq1) = rulerMk1Uniq _lhsIuniq
      (_fIsubst, _fIty, _fIuniq, f_1) = f_ _lhsIenv _lhsIsubst _fOuniq
      _tvar_ = mkTyVar _uniq1
      (_aIsubst, _aIty, _aIuniq, a_1) = a_ _lhsIenv _fIsubst _fIuniq
      (_subst_u_, _errUnify) = unify (_aIsubst $θ _fIty) (TyArr _aIty _tvar_)
     sem_Exp_1 = λ_lhsIsubstFin →                                      -- 2nd visit
      let (_aIerr, _aIpp) = a_1 _lhsIsubstFin
          (_fIerr, _fIpp) = f_1 _lhsIsubstFin
      in  (_errUnify ++ _fIerr ++ _aIerr
          , _fIpp >#< ppParens _aIpp >|< ppErr _errUnify)
  in  ( _subst_u_ $θ _aIsubst, _subst_u_ $θ _tvar_
      ,  _aIuniq
      , sem_Exp_1)
```

Implementation: multiple visits/passes

Statistics for UHC

- Analysis of expression terms: 8 visits

	deals with	nr of inh	syn	inh+syn
0	source text info			1
1	unique (fresh) identifier generation		1	1
2	type/kind env	3		1
3	kind/polarity inference/checking	6		3
4	class env, final type/kind env, datatype gathering	4		1
5	new class/instance gathering	1	4	
6	CHR env, value env, type inference	8	5	1
7	final value env, error gathering, ...	9	31	1
		31	41	9

Moments of reflection

What *allows easier change*?

- Embed DSLs, allowing better error reporting
- One DSL for both type rules and implementation of them

Moments of reflection

What *allows easier change*?

- Embed DSLs, allowing better error reporting
- One DSL for both type rules and implementation of them

The *ideal* would be to be able to specify declaratively

- The type system, with 'magic' figuring out a corresponding implementation
- An implementation using various DSLs inside one host language

The *reality* is that

- type systems soon are quite complex,
- 'magic' does not exist,
- embedding DSLs is still ongoing research leading to possibly difficult to understand error messages,
- and an implementation involves usually >1 host languages

Moments of reflection

What *allows easier change*?

- Embed DSLs, allowing better error reporting
- One DSL for both type rules and implementation of them

The *ideal* would be to be able to specify declaratively

- The type system, with 'magic' figuring out a corresponding implementation
- An implementation using various DSLs inside one host language

The *reality* is that

- type systems soon are quite complex,
- 'magic' does not exist,
- embedding DSLs is still ongoing research leading to possibly difficult to understand error messages,
- and an implementation involves usually >1 host languages

In UHC: consistency between type rules and implementation via *Ruler* system

Implementation: *Ruler*

Specification of type semantics often done in LaTeX

$$\frac{\begin{array}{c}\alpha \text{ fresh} \\ \theta; \Gamma \vdash f : \tau_f \leadsto \theta_f \\ \theta_a \tau_f \equiv \tau_a \to \alpha \leadsto \theta_u \\ \theta_f; \Gamma \vdash a : \tau_a \leadsto \theta_a\end{array}}{\theta; \Gamma \vdash f\ a : \theta_u \alpha \leadsto \theta_u \theta_a} \text{APP}_A$$

Is LaTeX (+ Lhs2TeX) a good specification language for type rules?

Implementation: *Ruler*

LaTeX + Lhs2TeX:

```
\rulerRule{app}{A}
{ | tvar | \;\mbox{fresh} | |
\\| subst ; env :- f : ty | _{ | f |} | ~> subst | _{ | f |} | |
\\| subst | _{ | a |} | ty | _{ | f |} | === ty | _{ | a |} | -> tvar ~> subst | _{ | u |} | |
\\| subst | _{ | f |} | ; env :- a : ty | _{ | a |} | ~> subst | _{ | a |} | |
 }
{ | subst ; env :- f ^^ a : subst | _{ | u |} | tvar ~> subst | _{ | u |} | subst | _{ | a |} | | }
```

Gives us pretty rendering in papers and (these) slides...
But...

- Mixup of specification & rendering
- Check for identifier introduction?
- Judgement conforms to its required structure?
- Typing of type rules?

And...

Implementation: *Ruler*

How can we keep type rules and their implementation consistent?

$$\frac{\begin{array}{c}\alpha \text{ fresh} \\ \theta ; \Gamma \vdash f : \tau_f \leadsto \theta_f \\ \theta_a \tau_f \equiv \tau_a \to \alpha \leadsto \theta_u \\ \theta_f ; \Gamma \vdash a : \tau_a \leadsto \theta_a\end{array}}{\theta ; \Gamma \vdash f\ a : \theta_u \alpha \leadsto \theta_u \theta_a} \ \mathrm{APP}_A$$

```
sem Exp
    | App (f.uniq, loc.uniq1)
                      = rulerMk1Uniq @lhs.uniq
          loc.tvar_   = mkTyVar @uniq1
          (loc.subst_u_, loc.errUnify)
                      = unify (@a.subst $_θ @f.ty) (TyArr @a.ty @tvar_)
          lhs.ty      = @subst_u_ $_θ @tvar_
             .subst   = @subst_u_ $_θ @a.subst
```

Implementation: *Ruler*

In general, given a specification for some semantics, we (want to)

1. Render it for *human reading* & reasoning,
2. Feed it into *theorem proving* machinery for (automated/mechanized) reasoning,
3. Generate code for *actual execution* (of e.g. a checker)

The *ideal* would be to obtain all 3 from a single description.
The *reality* is that for a given tool we get (approx) 2 out of 3...

Implementation: *Ruler*

In general, given a specification for some semantics, we (want to)

1. Render it for *human reading* & reasoning,
2. Feed it into *theorem proving* machinery for (automated/mechanized) reasoning,
3. Generate code for *actual execution* (of e.g. a checker)

The *ideal* would be to obtain all 3 from a single description.
The *reality* is that for a given tool we get (approx) 2 out of 3...

For UHC, *Ruler* gives us 1 & 3: previous slide contains generated rendering and AG code

Implementation: *Ruler*

One specification from which everything else is generated

Not a new idea:

System	Generates for, or implements LaTeX	mechanized reasoning	verified code	verifying code
Ott[6]	✓(indirect)	✓		
Coq[7]	✓	✓	✓(extraction)	
Twelf: ML spec[8]	✓	✓		
Ruler[9]	✓			✓(AG checker)

[6] "Ott: Effective Tool Support for the Working Semanticist", 2010

[7] `https://coq.inria.fr/`

[8]

[9] "Ruler: Programming Type Rules", 2006

Implementation: *Ruler*

Generating a checker for type rules (i.e. code which verifies) turned out to be rather difficult

- Why? How did we experiment with *Ruler* in relation to UHC? What is *Ruler*?

Implementation: *Ruler*

Ruler

- Specify rules
- Generate LaTeX and/or AG code

The *good news*

- Example in these slides is generated from a single *Ruler* specification (+ additional helper code)

Implementation: *Ruler*

Ruler example

$$
\begin{array}{l}
\textbf{scheme}\ \mathit{exp} = \\
\quad \textbf{view}\ D = \\
\quad\quad \textbf{holes}\ [\textbf{node}\ \mathit{exp} : \mathit{Exp}, \mathit{env} : \mathit{Env}, \mathit{ty} : \mathit{Ty}] \\
\quad\quad \textbf{judgespec}\ \mathit{env} \vdash \mathit{exp} : \mathit{ty}
\end{array}
$$

Specifies

- for the view *D* (declarative)
- the 'type' of judgements for *exp*,
- its parsing,
- LaTeX rendering (here same as parsing spec), and
- algorithmic annotations (here **node** specifies a variable represents syntax dispatched on)

Implementation: *Ruler*

Ruler example

rule *app* "App" $=$
  view $D =$
    judge $F : exp = env \vdash f : (ty.a \rightarrow ty.r)$
    judge $A : exp = env \vdash a : ty.a$
    $-$
    judge $R : exp = env \vdash (f\ \ a) : ty.r$

Specifies a single rule instance *app* of scheme *exp*

Implementation: *Ruler*

Generates (already seen)

```
\rulerRule{app}{A}
{ | tvar | \;\mbox{fresh} | |
\\| subst ; env :- f : ty | _{ | f |} | ~> subst | _{ | f |} | |
\\| subst | _{ | a |} | ty | _{ | f |} | === ty | _{ | a |} | -> tvar ~> subst | _{ | u |} | |
\\| subst | _{ | f |} | ; env :- a : ty | _{ | a |} | ~> subst | _{ | a |} | |
 }
{ | subst ; env :- f ^^ a : subst | _{ | u |} | tvar ~> subst | _{ | u |} | subst | _{ | a |} | | }
```

which with a little help from Lhs2TeX renders as (also already seen)

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

$$\frac{\begin{array}{c}\Gamma \vdash a : \tau_a \\ \Gamma \vdash f : \tau_a \to \tau_r\end{array}}{\Gamma \vdash f\ a : \tau_r}\ \mathrm{APP}_D$$

Implementation: *Ruler*

The *bad news*

- *Ruler* can generate only when rules are algorithmic and syntax driven
- Which the declarative variant of our example is not, so
- Need to add additional info or override existing to arrive at a rule from which we can generate AG

Ruler provides mechanisms for additional or replacement specification to obtain

$$\frac{\Gamma \vdash a : \tau_a \qquad \Gamma \vdash f : \tau_a \rightarrow \tau_r}{\Gamma \vdash f\ a : \tau_r}\ \mathrm{APP}_D$$

$$\frac{\begin{array}{c}\alpha \text{ fresh} \\ \theta; \Gamma \vdash f : \tau_f \rightsquigarrow \theta_f \\ \theta_a \tau_f \equiv \tau_a \rightarrow \alpha \rightsquigarrow \theta_u \\ \theta_f; \Gamma \vdash a : \tau_a \rightsquigarrow \theta_a\end{array}}{\theta; \Gamma \vdash f\ a : \theta_u \alpha \rightsquigarrow \theta_u \theta_a}\ \mathrm{APP}_A$$

Implementation: *Ruler*

The *really bad news*

- Rules being algorithmic and syntax driven is not enough for many type systems

Implementation: *Ruler*

Why is it that rules being algorithmic and syntax driven is not enough for many type systems?

- Deterministic vs. non-deterministic!

When is non-determinism *present* and/or *required*?

- In HM type system example non-deterministism is present, but not required, because we can transform it to
- Algorithm W: deterministic, because rule choice is syntax driven, algorithmic because relationships have direction (are functions, computable)

Implementation: *Ruler*

Why is it that rules being algorithmic and syntax driven is not enough for many type systems?

- Deterministic vs. non-deterministic!

When is non-determinism *present* and/or *required*?

- In HM type system example non-deterministism is present, but not required, because we can transform it to
- Algorithm W: deterministic, because rule choice is syntax driven, algorithmic because relationships have direction (are functions, computable)
- Counterexample: Haskell type class system cannot be dealt with in a syntax driven way

Let's look at its rules and see why...

Example: Haskell type class system

Haskell type class system

- Haskell example

$$
\begin{array}{ll}
\textbf{class}\ Eq\ a\ \textbf{where} & \\
\quad (==) :: a \rightarrow a \rightarrow Bool & \\
\textbf{instance}\ Eq\ Int\ \textbf{where} \ldots & \\
\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} \ldots & \\
f :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool & \text{-- specified or inferred} \\
f\ x\ y = x == y & \text{-- just a (nonsensical) example}
\end{array}
$$

- Extension of running example with qualified types

$$
\begin{array}{ll}
\tau ::= Int \mid \tau \rightarrow \tau \mid \alpha & \\
\rho ::= \tau \mid \pi \Rightarrow \rho & \text{-- qualified} \\
\pi ::= Eq\ \tau \mid \ldots & \\
\sigma ::= \rho \mid \forall\ \alpha.\sigma &
\end{array}
$$

Example: Haskell type class system

Almost independent extension of HM type system[10], declaratively

$$\boxed{\mathcal{P} \mid \Gamma \vdash e : \tau}$$

$$\frac{n \mapsto \sigma \in \Gamma}{\mathcal{P} \mid \Gamma \vdash n : \sigma}\ \mathrm{VAR}_Q \qquad \frac{\begin{array}{c}\mathcal{P} \mid \Gamma[n \mapsto \sigma] \vdash e_b : \tau \\ \mathcal{P} \mid \Gamma \vdash e : \sigma\end{array}}{\mathcal{P} \mid \Gamma \vdash \mathbf{let}\ n = e\ \mathbf{in}\ e_b : \tau}\ \mathrm{LET}_Q$$

$$\frac{\mathcal{P} \mid \Gamma[n \mapsto \tau_a] \vdash e : \tau_r}{\mathcal{P} \mid \Gamma \vdash \lambda n.e : \tau_a \to \tau_r}\ \mathrm{ABS}_Q \qquad \frac{\begin{array}{c}\mathcal{P} \mid \Gamma \vdash a : \tau_a \\ \mathcal{P} \mid \Gamma \vdash f : \tau_a \to \tau_r\end{array}}{\mathcal{P} \mid \Gamma \vdash f\ a : \tau_r}\ \mathrm{APP}_Q$$

- $\mathcal{P}$: assumed (i.e. given, true, ...) type class predicates
  - E.g. $Eq\ Int, Eq\ a \Rightarrow Eq\ [a]$

[10] "Qualified Types, Theory and Practice", 1994

Example: Haskell type class system

Tweak generalization

$$\boxed{\mathcal{P} \mid \Gamma \vdash e : \tau}$$

$$\frac{\alpha \notin (\mathit{ftv}\ (\Gamma) \cup \mathit{ftv}\ (\mathcal{P})) \qquad \mathcal{P} \mid \Gamma \vdash e : \tau}{\mathcal{P} \mid \Gamma \vdash e : \forall\ \alpha.\tau}\ \mathrm{GEN}_Q \qquad \frac{\alpha \text{ fresh} \qquad \mathcal{P} \mid \Gamma \vdash e : \forall\ \beta.\tau}{\mathcal{P} \mid \Gamma \vdash e : [\beta \mapsto \alpha]\ \tau}\ \mathrm{INST}_Q$$

Example: Haskell type class system

Additional rules for predicate introduction and elimination

$$\boxed{\mathcal{P} \mid \Gamma \vdash e : \tau}$$

$$\frac{\mathcal{P}, \pi \mid \Gamma \vdash e : \rho}{\mathcal{P} \mid \Gamma \vdash e : \pi \Rightarrow \rho}\,\text{INTROP}_Q \qquad \frac{\begin{array}{c}\mathcal{P} \mid \Gamma \vdash e : \pi \Rightarrow \rho \\ \mathcal{P} \vDash \pi\end{array}}{\mathcal{P} \mid \Gamma \vdash e : \rho}\,\text{ELIMP}_Q$$

Entailment $\mathcal{P} \vDash \pi$

Example: Haskell type class system

Additional rules for predicate introduction and elimination

$$\boxed{\mathcal{P} \mid \Gamma \vdash e : \tau}$$

$$\frac{\mathcal{P}, \pi \mid \Gamma \vdash e : \rho}{\mathcal{P} \mid \Gamma \vdash e : \pi \Rightarrow \rho}\ \text{INTROP}_Q \qquad \frac{\begin{array}{c}\mathcal{P} \mid \Gamma \vdash e : \pi \Rightarrow \rho \\ \mathcal{P} \vDash \pi\end{array}}{\mathcal{P} \mid \Gamma \vdash e : \rho}\ \text{ELIMP}_Q$$

Entailment $\mathcal{P} \vDash \pi$

- Usual rules: transitivity, ...
- Rules derived from class/instance definitions
- Requires a small theorem prover
  - E.g. to derive $Eq\ [Int]$ from $Eq\ Int, Eq\ a \Rightarrow Eq\ [a]$

Example: Haskell type class system

Additional rules for predicate introduction and elimination

$$\boxed{\mathcal{P} \mid \Gamma \vdash e : \tau}$$

$$\frac{\mathcal{P}, \pi \mid \Gamma \vdash e : \rho}{\mathcal{P} \mid \Gamma \vdash e : \pi \Rightarrow \rho}\ \mathrm{INTROP}_Q \qquad \frac{\begin{array}{c}\mathcal{P} \mid \Gamma \vdash e : \pi \Rightarrow \rho \\ \mathcal{P} \vDash \pi\end{array}}{\mathcal{P} \mid \Gamma \vdash e : \rho}\ \mathrm{ELIMP}_Q$$

Entailment $\mathcal{P} \vDash \pi$

- Usual rules: transitivity, ...
- Rules derived from class/instance definitions
- Requires a small theorem prover
  - E.g. to derive $Eq\ [Int]$ from $Eq\ Int, Eq\ a \Rightarrow Eq\ [a]$

So, why is this a problem?

Example: Haskell type class system

Can we make the rules syntax directed and algorithmic?

- To some degree, but...
  - Direction of $\mathcal{P}$ is inherited: classes and instances are given
  - Direction of $\mathcal{P}$ is synthesized: during type inference occurrences of identifiers give rise to πs
  - Predicates π may involve type variables for which we 'later' find substitutions
- Theorem proving (context reduction) usually ends up being done in **let** expression, before generalization
  - Gather constraints syntax directed, prove at **let** because *ftv* (..) determines satisfiability
  - Need results still at location where constraint arose, for inserting evidence,
  - Requires representations which can be 'patched' later

Example: Haskell type class system

Can we make the rules syntax directed and algorithmic?

- To some degree, but...
  - Direction of $\mathcal{P}$ is inherited: classes and instances are given
  - Direction of $\mathcal{P}$ is synthesized: during type inference occurrences of identifiers give rise to πs
  - Predicates π may involve type variables for which we 'later' find substitutions
- Theorem proving (context reduction) usually ends up being done in **let** expression, before generalization
  - Gather constraints syntax directed, prove at **let** because *ftv* (..) determines satisfiability
  - Need results still at location where constraint arose, for inserting evidence,
  - Requires representations which can be 'patched' later

In general

- More complex type system, less syntax directedness, more via constraint solving (possibly involving backtracking)

Implementation: *Ruler*

Back to the *really bad news*

- Rules being algorithmic and syntax driven is not enough for many type systems

Solutions:

Implementation: *Ruler*

Back to the *really bad news*

- Rules being algorithmic and syntax driven is not enough for many type systems

Solutions:

- Just a Prolog program!
  - Do we really want logic programming (overhead) for all compiler tasks?
- Paradigm mix: deterministic (syntax directed) AG-like and non-deterministic (logic programming, constraint solving, backtracking) Prolog-like programming
  - Given a declarative set of (type) rules, can we figure out what can be done using which paradigm?

Implementation: multiple visits/passes

Exploration in context of UUAGC, inspired by UHC

Implementation mechanisms[11] for which a combi of AG and constraint solving solutions can be generated

- Explicit scheduling for attribute evaluation: sequence of visits/passes
  - Each visit/pass a coroutine
- Can be invoked syntax directed or uncoupled from syntax (as part of constraint solving)
- Can backtrack; can give partial results

[11] "Inference of Program Properties with Attribute Grammars, Revisited", 2012

Implementation: multiple visits/passes

Exploration in context of UUAGC, inspired by UHC

Implementation mechanisms[11] for which a combi of AG and constraint solving solutions can be generated

- Explicit scheduling for attribute evaluation: sequence of visits/passes
  - Each visit/pass a coroutine
- Can be invoked syntax directed or uncoupled from syntax (as part of constraint solving)
- Can backtrack; can give partial results
- Left unexplored: design of a *Ruler* successor which allows declarative and algorithmic part to be independently specified, thus avoiding 'pollution' of declarative part

[11] "Inference of Program Properties with Attribute Grammars, Revisited", 2012

Moments of reflection: UUAGC, *Ruler*, (non)determinism

UUAGC

- Great for: 'tree-oriented' programming: syntax directed, deterministic
- Great for: independent specifications for attributes, combined later on
- Dependency analysis gives efficient (strict) visit based code
- Generates boilerplate code
- Manages *complexity* of aspects

Used a lot in UHC, and in other tools as well

- Still have to look at: combining language features (not just aspects per feature)

Moments of reflection: UUAGC, *Ruler*, (non)determinism

Ruler

- In addition to UUAGC: pretty (LaTeX) printing of type rules
- Inspired exploration of visits as target machine model for checkers using syntax directedness, constraint solving, and backtracking

But, as it is, *Ruler* not used anymore in UHC as it offers too little on top of UUAGC

Management of *complexity* (of choice) of implementation mechanisms: still unresolved

Moments of reflection: UUAGC, *Ruler*, (non)determinism

Nondeterminism

- In declarative specification (of rules)
  - Convert to algorithmic specification
  - The (practical attainable) *ideal* would be to annotate (in *Ruler*) which mechanism ($\in$ {syntax directedness, unification, constraint solving, ...}) should be used
- In the language (for which we specify rules)
  - Language specification may allow ambiguity
  - In Haskell: higher ranked types, overlapping instances, (in UHC) local instances
  - Language mechanisms for explicitly disambiguating: type signatures, functional dependencies, (in UHC) named instances
  - The *ideal* programming language should offer for each implicit mechanism possibly leading to ambiguity an explicit mechanism for disambiguation

Moments of reflection: UUAGC, *Ruler*, (non)determinism

Presence of non-determinism ultimately leads to 'choosing' mechanisms

```
class Eq a where
  (==) :: a -> a -> Bool
instance Eq Int where                              -- (I1)
  x == y = primEqInt x y
e1 = 3 == 5                                        -- result: False
e2 = let instance Eq Int where                     -- (I2)
           x == y = primEqInt (x `mod` 2)
                              (y `mod` 2)
     in  3 == 5                                    -- result: True
```

In UHC: local instances

Moments of reflection: UUAGC, *Ruler*, (non)determinism

In UHC

- Choosing via
  - Scope of instance
  - Naming instances (by the UHC programmer)
- Delaying choice in the implementation by generating all possible solutions
  - Uses DSL for rule based programming: Constraint Handling Rules (CHR)[12]
  - Rules deal with scope explicitly
  - Left unexplored: programmer specifiable strategies for choosing (allowing policies for mechanism)
  - Limitation: CHR variant too limited to be able to deal with (e.g. functional dependencies, ...)

[12] "*Modelling Scoped Instances with Constraint Handling Rules*", 2007

Compositionality of language features

Language features versus implementation aspects

- UUAGC takes care about compositionality of aspects (type system, pretty printing, ...) of language features

How does UHC deal with compositionality of language features?

Compositionality of language features

Desired compositionality:

- Informally: need not modify individual specifications when composing
  - *ideal*: concatenate textually
  - *reality*: concatenate textually + additional glue/override

Compositionality of language features

Desired compositionality:

- Informally: need not modify individual specifications when composing
  - *ideal*: concatenate textually
  - *reality*: concatenate textually + additional glue/override
- More formally:
  - Given (already theoretical/formally well defined) language features f_i ($i \in \{1, 2\}$),
  - (textual) specifications s_i for f_i for which we have already some notion $\sim$ of correctness $s_i \sim f_i$,
  - feature combination '$+$' and textual specification concatenation '$+\!\!+$',
  - specific additional specification $s_{1,2}$ to make hold $(s_1 +\!\!+ s_2 +\!\!+ s_{1,2}) \sim (f_1 + f_2)$,
  - if $s_{1,2} = \emptyset$ then specifications s_1, s_2 are compositional

Compositionality of language features

Desired compositionality:

- Informally: need not modify individual specifications when composing
  - *ideal*: concatenate textually
  - *reality*: concatenate textually + additional glue/override
- More formally:
  - Given (already theoretical/formally well defined) language features f_i $(i \in \{1, 2\})$,
  - (textual) specifications s_i for f_i for which we have already some notion $\sim$ of correctness $s_i \sim f_i$,
  - feature combination '$+$' and textual specification concatenation '$+\!\!+$',
  - specific additional specification $s_{1,2}$ to make hold $(s_1 +\!\!+ s_2 +\!\!+ s_{1,2}) \sim (f_1 + f_2)$,
  - if $s_{1,2} = \emptyset$ then specifications s_1, s_2 are compositional
- What about compositionality for $f_1 + f_2$?
  - Observation: usually many published papers p_i for individual f_i, few papers $p_1 + p_2$?

Compositionality of language features

Not a new idea:

System	$i =$ operational semantics	verifying code
TinkerType[13]	✓	✓(unchecked)
PlanCompS[14]	✓	✓
Shuffle(UHC)		textual combining

[13] *TinkerType: A Language for Playing with Formal Systems*, 1999
[14] "Reusable Components of Semantic Specifications", 2015

Compositionality of language features

Is *composition* of specifications (for language features) possible?

- Observation: usually $s_{i,j} \neq \emptyset$ for arbitrary s_i, s_j
- For n features $> O(n^2)$ combinations, s_k might require $s_{i,j,k}$ specifics

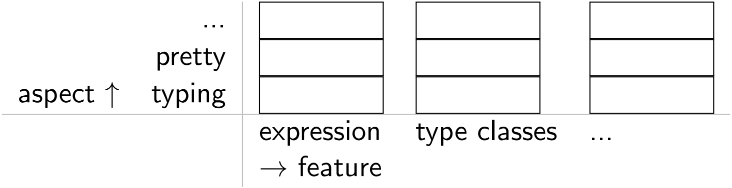
In context of UHC: *no*

- A language feature may rely on another language feature: Haskell type class system requires expressions and/or type system
  - Ordering between language features, 'later' features have no meaning on their own
- A language feature may have to override the specification of another feature when combined: Haskell type class system changes (e.g.) code generation
  - Composing can potentially require (ad-hoc, implementation specific) changes on individual specifications of features

Compositionality of language features

Expression Problem[15]

- Feature: structure, data type, AST
- Aspect: computation, semantics



- UUAGC compositionality: able to independently fill in the squares
- Square ordering
  - Square assumes presence of other square
  - Square overrides (part of) other square (difficult to do with UUAGC)

[15] *The Expression Problem*, 1998

Language variants: *Shuffle*

For UHC:

- Composing requires ad-hoc composition specific overriding
  - Restrict to ordering of features building on top of each other
  - Aspects as independent of features as possible
- Features and aspects are relevant for all artefacts of a compiler
  - AG files, Haskell files, C (runtime system), ...
  - Only manipulation of file content, semantics agnostic

Tool *Shuffle* selects and shuffles around text fragments annotated with aspect and variant (language feature)

Language variants: *Shuffle*

Example from UHC:

```
%%[(2 hmtyinfer || hmtyast).mkTyVar hs
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv
%%]
```

Haskell (hs) fragment to be included in the compiler for variant 2, when incorporating type AST (hmtyast) or type inference (hmtyinfer)

Language variants: *Shuffle*

Example from UHC:

```
%%[(3 hmtyinfer || hmtyast).mkTyVar -2.mkTyVar hs
mkTyVar :: TyVarId -> Ty
mkTyVar tv = Ty_Var tv TyVarCateg_Plain
%%]
```

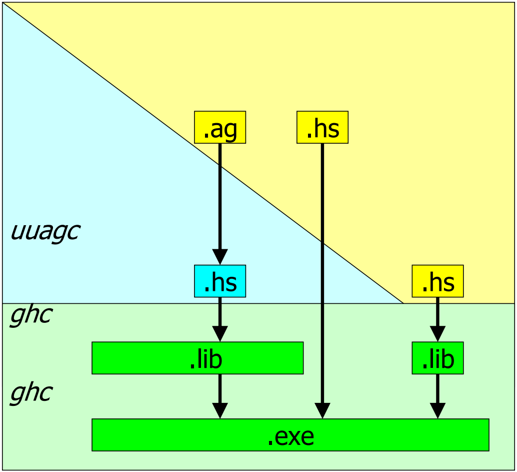
Override a fragment for variant 3

Language variants: *Shuffle*

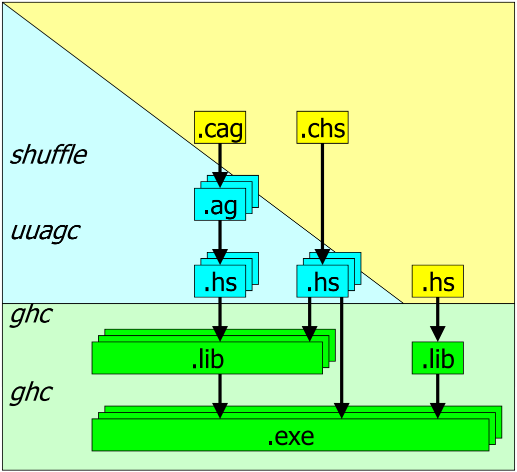
In UHC

	Plain Haskell	Experiments
1	λ-calculus, type checking	
2	type inference	partial type signature
3	polymorphism	
4		higher ranked types, existentials
5	data types	
6	kind inference	kind signatures
7	records	tuples as records
8	code generation	whole program analysis
9	classes, type-synonyms	extensible records
..	modules	
..	deriving	
..	prelude, I/O	
..	Haskell	

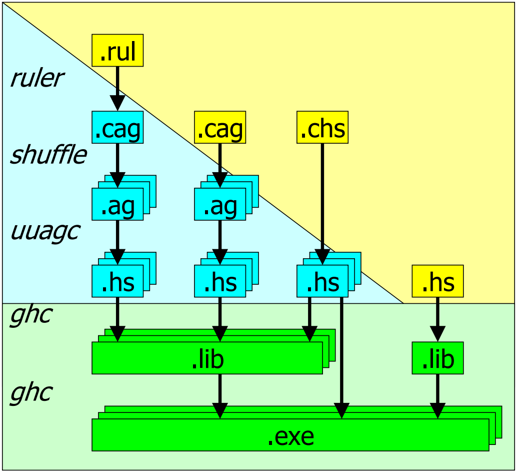
Language variants: *Shuffle*, *Ruler*



Language variants: *Shuffle*, *Ruler*



Language variants: *Shuffle*, *Ruler*



Language variants: *Shuffle*

Shuffle good news

- Can build compiler per variant
- Can turn on/off aspects (configuration management)
- Shuffle fragments can be referred to by name for text inclusion (literate programming)
- Helps with debugging

Language variants: *Shuffle*

Shuffle good news

- Can build compiler per variant
- Can turn on/off aspects (configuration management)
- Shuffle fragments can be referred to by name for text inclusion (literate programming)
- Helps with debugging

Shuffle bad news

- Rather crude/simple 'textual only' tool
- Makes build system complex, adds another language to be learned
- Keeping all combinations working takes effort, not all combinations can work

Moments of reflection: compositionality

Approach/desire: use compositionality to simplify specification of UHC

Does it work? Can it be done?

- Yes, when aspects/features are independent

When restricted to feature ordening in the presence of dependency between features

- Features become tightly coupled
- Subsequent feature may need redesign of implementation of previous feature
  - Either must override a lot, or design for the last feature,
  - making the first feature also depend on later variants

Moments of reflection: compositionality

Conjecture

- Compositionality probably feasible for simple languages, but not for the complex ones (like Haskell)

What is known about feature composition and their (in)dependence?

- *ideal*: for every paper p_i, p_j (describing theory, semantics, etc) for feature f_i, f_j there is a paper $p_{i,j}$ for $f_i + f_j$
- *reality*: there are few $p_{i,j}$'s
  - Left to the implementer (to figure something out)

But still compositionality is the major mechanism to keep complexity manageable

Engineering: the devil is in the detail (1)

(The sting is in the tail, before summarizing)
The small seemingly innocent can bite

$$\frac{\begin{array}{c}\alpha \notin \mathit{ftv}\ (\Gamma) \\ \Gamma \vdash e : \tau\end{array}}{\Gamma \vdash e : \forall\ \alpha.\tau}\ \mathrm{GEN}_D$$

ftv traverses Γ

- Γ may be large, possibly 'naively' holding all imported modules
- Does not scale

Need for incrementality

Engineering: the devil is in the detail (1)

Observations:

- *ftv* (Γ) usually is small, empty for each module
- *ftv* (Γ) can be incrementally computed

In either case: manual tweaking/optimization required

- Led to exploration of (generation of) incrementality for AG[16]

[16] "On the Incremental Evaluation of Higher-Order Attribute Grammars", 2015

Engineering: the devil is in the detail (2)

Encoding of substitutions

$$\frac{\begin{array}{c}\alpha \text{ fresh} \\ \theta; \Gamma \vdash f : \tau_f \leadsto \theta_f \\ \theta_a \tau_f \equiv \tau_a \rightarrow \alpha \leadsto \theta_u \\ \theta_f; \Gamma \vdash a : \tau_a \leadsto \theta_a\end{array}}{\theta; \Gamma \vdash f\ a : \theta_u \alpha \leadsto \theta_u \theta_a} \text{APP}_A$$

Naive encoding eagerly propagates substitution, rebuilding structures

Engineering: the devil is in the detail (2)

Invariant: substitution maps in 1 lookup (no indirections)

$$\textbf{class}\ \mathit{Substitutable}\ x\ \textbf{where}\ \{(\$_\theta) :: \mathit{Subst} \to x \to x\}$$

$$\textbf{instance}\ \mathit{Substitutable}\ \mathit{Subst}\ \textbf{where}$$
$$\quad s_1\ \$_\theta\ s_2 = s_1 +\!\!+\ \mathit{map}\ (\lambda(v,t) \to (v, s_1\ \$_\theta\ t))\ s_2 \quad \text{-- } O(n^2)$$

$$\textbf{instance}\ \mathit{Substitutable}\ \mathit{Ty}\ \textbf{where}$$
$$\quad s\ \$_\theta\ t = \textbf{case}\ t\ \textbf{of}$$
$$\qquad \mathit{TyVar}\ v \to \mathit{maybe}\ t\ (s\$_\theta)\ \$\ \mathit{lookup}\ v\ s$$
$$\qquad \mathit{TyArr}\ a\ r \to \mathit{TyArr}\ (s\ \$_\theta\ a)\ (s\ \$_\theta\ r)$$

$$\textbf{data}\ \mathit{TyVar} = \ldots$$
$$\textbf{data}\ \mathit{Ty} = \mathit{TyInt} \mid \mathit{TyVar}\ \mathit{TyVar} \mid \mathit{TyArr}\ \mathit{Ty}\ \mathit{Ty} \mid \ldots$$
$$\textbf{type}\ \mathit{Subst} = [(\mathit{TyVar}, \mathit{Ty})]$$

Does *not scale*: $O(n^2)$ complexity, size of structures increases (duplication when variables occur >1 times)

Engineering: the devil is in the detail (2)

Solution: delay substitution[17]

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Substitutable}\ \mathit{Subst}\ \textbf{where} \\
\quad s_1 \mathbin{\$_\theta} s_2 = s_1 \mathbin{+\!\!+} s_2 \\
\textbf{instance}\ \mathit{Substitutable}\ \mathit{Ty}\ \textbf{where} \\
\quad s \mathbin{\$_\theta} t = \textbf{case}\ t\ \textbf{of} \\
\quad\quad \mathit{TyVar}\ v \rightarrow \mathit{maybe}\ t\ (s\ \$_\theta)\ \$\ \mathit{substLookup}\ v\ s \\
\quad\quad \ldots \\
\mathit{substLookup}\ v\ s = \mathit{lookup}\ v\ s \ggg\!= \lambda t \rightarrow \\
\quad (\mathit{tyIsVar}\ t \ggg\!= \mathit{flip}\ \mathit{substLookup}\ s)\ \langle|\rangle\ \mathit{return}\ t
\end{array}
$$

Price: all code assuming fully substituted types now can no longer assume this, must be passed substitution as additional parameter

[17] *Efficient Functional Unification and Substitution*, 2008

Moments of reflection

The *ideal*

- Implementation follows directly from declarative/algorithmic specification

The *reality*

- Engineering issues (scaling up) require (manual) tweaking

In UHC

- Manual tweaking, making code less readable/understandable
- Left unexplored: generated implementation for these idioms (e.g. as part of *Ruler*)

Summary

What did we learn from building UHC?

- Use tools, engineering, (non embedded) dsl, exploit compositionality
- Some complexity can be engineered away, some complexity is introduced by engineering other complexity away
- Even though we use tools (etc) building a (non toy) compiler is complex, requires (human) resources
- Compositionality and language design are intertwined, perhaps too difficult to disentangle
- Not discussed: for the sake of compositionality no *unsafePerformIO* was abused
- Not discussed: performance, some solutions perform worse but tend to be negligable relative to the rest of what compiler does
- Still wishing: compiler writers "design idiom" toolbox allowing easy construction of (reasonably) efficient compilers

Web locations & contributors

This talk & running example demos

- Built using *Shuffle*, *Ruler*, UUAGC, GHC, Lhs2TeX, LaTeX

URLs

- This talk & running example demos:
  Part of `https://github.com/atzedijkstra/uhc-doc`
- UHC, UUAGC, and other tools:
  `https://github.com/UU-ComputerScience/`
  Installable from `http://hackage.haskell.org/`

Contributors

- Doaitse Swierstra, Jeroen Fokker, Arie Middelkoop, Marcos Viera, Jeroen Bransen, students ...