The Structure of the Essential Haskell Compiler
Coping with Compiler Complexity

Atze Dijkstra, Jeroen Fokker, Doaitse Swierstra
Utrecht University

IFL, September 27–29, 2007

The structure of the Essential Haskell Compiler. . .

We are writing a Haskell compiler
that by design *evolves*

- from *lambda calculus* to *full Haskell*
- from *essential* to *syntactically sugared*
- from *common constructs* to *extensions*

. . . or: Coping with Compiler Complexity

When writing a compiler we face

- Implementation complexity
- Description/Coding complexity
- Design complexity
- Maintenance complexity

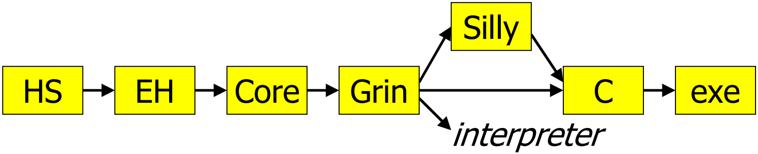
. . . or: Coping with Compiler Complexity

When writing a compiler we face

- Implementation complexity
  a compiler is a *large program*
- Description/Coding complexity
  translation involves *complicated abstract syntax trees*
- Design complexity
  a language has *many features*
- Maintenance complexity
  evolving projects must *remain consistent*

Coping with *Implementation* Complexity

Transform!

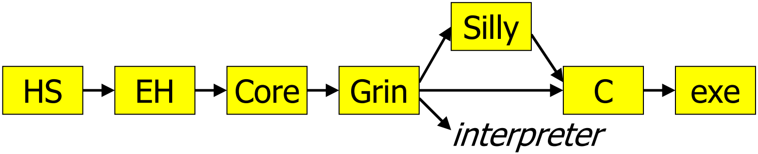


Many intermediate languages, *7 transformations*

- Haskell
- Essential Haskell
- Core
- Grin
- Silly
- C

Coping with *Implementation* Complexity

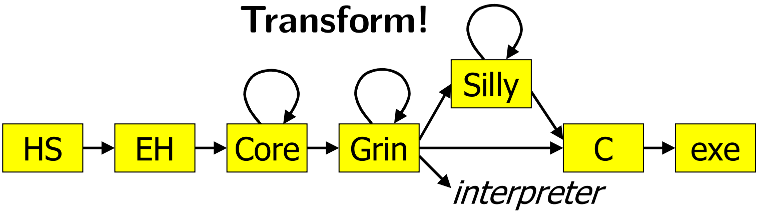
Transform!



Many intermediate languages, *7 transformations*

- Haskell
- Essential Haskell (desugared)
- Core (type erased)
- Grin (sequential)
- Silly (imperative)
- C

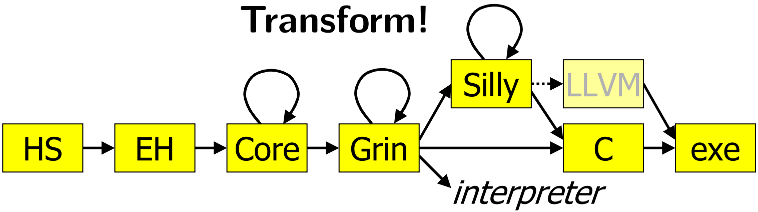
Coping with *Implementation* Complexity



Many intermediate languages, *7 transformations*

- Haskell
- Essential Haskell
- Core *12 transformations*
- Grin *17 transformations*
- Silly *3 transformations*
- C

Coping with *Implementation* Complexity

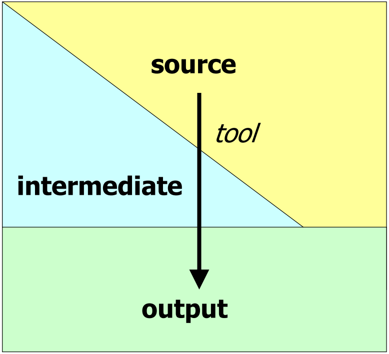


Many intermediate languages, *7 transformations*

- Haskell
- Essential Haskell
- Core *12 transformations*
- Grin *17 transformations*
- Silly *3 transformations*
- C

Coping with *Description* Complexity

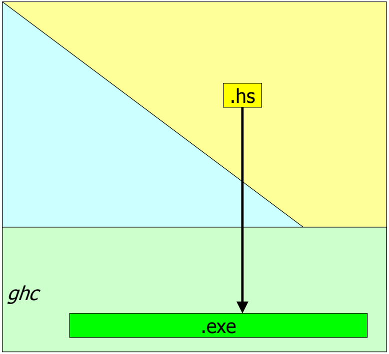
Use tools!



Coping with *Description* Complexity

Use tools!

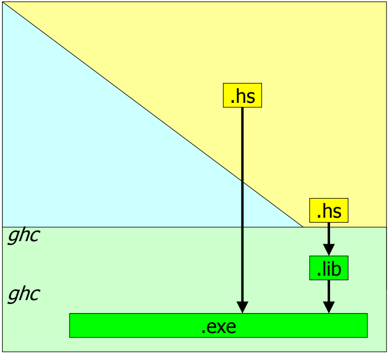
- *Glasgow* Haskell Compiler



Coping with *Description* Complexity

Use tools!

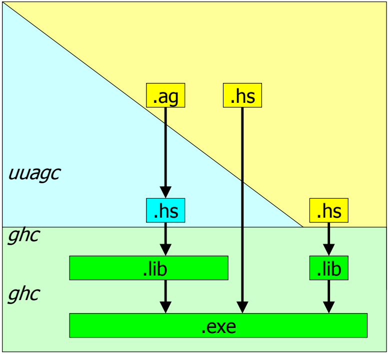
- *Glasgow* Haskell Compiler



Coping with *Description* Complexity

Use tools!

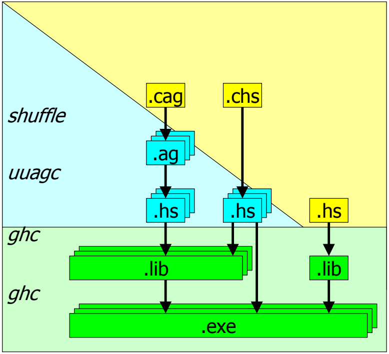
- *Utrecht University* Attribute Grammar Compiler
- *Glasgow* Haskell Compiler



Coping with *Description* Complexity

Use tools!

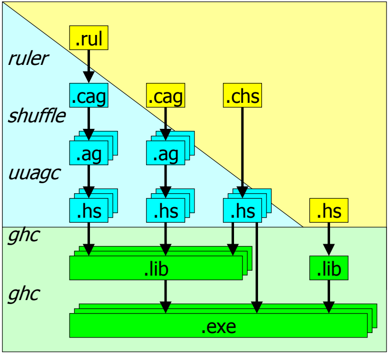
- Shuffle
- *Utrecht University* Attribute Grammar Compiler
- *Glasgow* Haskell Compiler



Coping with *Description* Complexity

Use tools!

- Ruler
- Shuffle
- *Utrecht University* Attribute Grammar Compiler
- *Glasgow* Haskell Compiler



Coping with *Design* Complexity

Grow stepwise!

1. λ-calculus, type checking
2. type inference
3. polymorphism
4.
5. data types

Coping with *Design* Complexity

Grow stepwise!

	plain Haskell	experiments
1	λ-calculus, type checking	
2	type inference	
3	polymorphism	
4		higher ranked types, existentials
5	data types	

Coping with *Design* Complexity

Grow stepwise!

	plain Haskell	experiments
1	λ-calculus, type checking	
2	type inference	
3	polymorphism	
4		higher ranked types, existentials
5	data types	
6	kind inference	kind signatures
7	records	tuples as records
8	code generation	full program analysis
9	classes, type-synonyms	extensible records

Coping with *Design* Complexity

Grow stepwise!

	plain Haskell	experiments
1	λ-calculus, type checking	
2	type inference	
3	polymorphism	
4		higher ranked types, existentials
5	data types	
6	kind inference	kind signatures
7	records	tuples as records
8	code generation	full program analysis
9	classes, type-synonyms	extensible records
20	modules	
95	'deriving'	exceptions
99	prelude, I/O	

Coping with *Maintenance* Complexity

Generate, generate, generate...

Domain specific languages
and tools to transform them:

- Attribute Grammar Compiler
- Ruler
- Shuffle

Tools: Attribute Grammar Compiler

write recursive functions

$$
\begin{aligned}
&\mathit{sum}\ [\,] &&= 0\\
&\mathit{sum}\ (x : xs) &&= x + \mathit{sum}\ xs\\
&\mathit{concat}\ [\,] &&= [\,]\\
&\mathit{concat}\ (x : xs) &&= x \mathbin{+\!\!+} \mathit{concat}\ xs
\end{aligned}
$$

Tools: Attribute Grammar Compiler

Do not write recursive functions

$$
\begin{aligned}
&\mathit{sum}\ [\,] &&= 0\\
&\mathit{sum}\ (x : \mathit{xs}) &&= x + \mathit{sum}\ \mathit{xs}\\
&\mathit{concat}\ [\,] &&= [\,]\\
&\mathit{concat}\ (x : \mathit{xs}) &&= x \mathbin{+\!\!+} \mathit{concat}\ \mathit{xs}
\end{aligned}
$$

but generalize...

$$
\begin{aligned}
&\mathit{foldr}\ \mathit{op}\ e\ [\,] &&= e\\
&\mathit{foldr}\ \mathit{op}\ e\ (x : \mathit{xs}) &&= x\ `\mathit{op}`\ \mathit{foldr}\ \mathit{op}\ e\ \mathit{xs}
\end{aligned}
$$

Tools: Attribute Grammar Compiler

Do not write recursive functions

$$
\begin{aligned}
&\mathit{sum}\ [\,] &&= 0\\
&\mathit{sum}\ (x : xs) &&= x + \mathit{sum}\ xs\\
&\mathit{concat}\ [\,] &&= [\,]\\
&\mathit{concat}\ (x : xs) &&= x +\!\!+\ \mathit{concat}\ xs
\end{aligned}
$$

but generalize...

$$
\begin{aligned}
&\mathit{foldr}\ op\ e\ [\,] &&= e\\
&\mathit{foldr}\ op\ e\ (x : xs) &&= x\ \text{`}op\text{`}\ \mathit{foldr}\ op\ e\ xs
\end{aligned}
$$

...and specialize

$$
\begin{aligned}
&\mathit{sum} &&= \mathit{foldr}\ (+)\ 0\\
&\mathit{concat} &&= \mathit{foldr}\ (+\!\!+)\ [\,]\\
&\mathit{sort} &&= \mathit{foldr}\ \mathit{insert}\ [\,]
\end{aligned}
$$

Tools: Attribute Grammar Compiler

Do not write recursive functions

$$\begin{aligned}
&\mathit{sum} && [\,] && = 0\\
&\mathit{sum} && (x : xs) && = x + \mathit{sum}\ xs\\
&\mathit{concat} && [\,] && = [\,]\\
&\mathit{concat} && (x : xs) && = x +\!\!+\ \mathit{concat}\ xs
\end{aligned}$$

but generalize. . .

$$\begin{aligned}
&\mathit{foldr}\ op\ e\ [\,] && = e\\
&\mathit{foldr}\ op\ e\ (x : xs) && = x\ \text{`}op\text{`}\ \mathit{foldr}\ op\ e\ xs
\end{aligned}$$

. . . and specialize

$$\begin{aligned}
&\mathit{sum} && = \mathit{foldr}\ (+)\ 0\\
&\mathit{concat} && = \mathit{foldr}\ (+\!\!+)\ [\,]\\
&\mathit{sort} && = \mathit{foldr}\ \mathit{insert}\ [\,]
\end{aligned}$$

$$\mathit{catamorphism} = \mathit{foldr}_T\ \mathit{algebra}_T$$

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

- we need a custom *fold*-function for each datatype

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

- we need a custom *fold*-function for each datatype

- algebras contain *many* functions which...

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

- we need a custom *fold*-function for each datatype
- algebras contain *many* functions which...
- ...all return a tuple
- ...take extra parameters

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

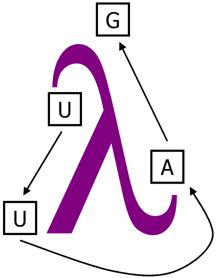
- we need a custom *fold*-function for each datatype
- algebras contain *many* functions which. . .
- . . . all return a tuple
- . . . take extra parameters
- . . . and mostly just pass values up or down

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing why does nobody do it when processing parse trees?

- we need a custom *fold*-function for each datatype
- algebras contain *many* functions which...
- ...all return a tuple
- ...take extra parameters
- ...and mostly just pass values up or down

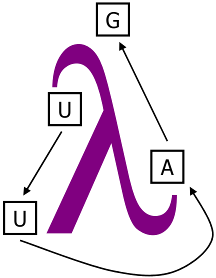
our *Attribute Grammar Compiler* makes this easy



Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

- we need a custom *fold*-function for each datatype
  automatically generated
- algebras contain *many* functions which...
- ...all return a tuple
- ...take extra parameters
- ...and mostly just pass values up or down

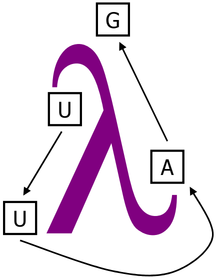


our *Attribute Grammar Compiler* makes this easy

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

- we need a custom *fold*-function for each datatype
  automatically generated
- algebras contain *many* functions which...
  distributedly definable
- ...all return a tuple
- ...take extra parameters
- ...and mostly just pass values up or down

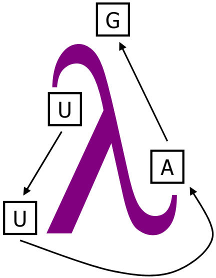


our *Attribute Grammar Compiler* makes this easy

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

- we need a custom *fold*-function for each datatype
  automatically generated
- algebras contain *many* functions which…
  distributedly definable
- …all return a tuple
  synthesized attributes
- …take extra parameters
  inherited attributes
- …and mostly just pass values up or down

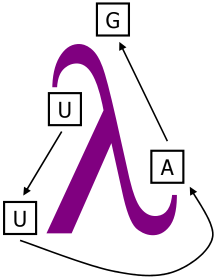


our *Attribute Grammar Compiler* makes this easy

Tools: Attribute Grammar Compiler

If *programming by writing algebras* is a Good Thing
why does nobody do it when processing parse trees?

- we need a custom *fold*-function for each datatype
  automatically generated
- algebras contain *many* functions which. . .
  distributedly definable
- . . . all return a tuple
  synthesized attributes
- . . . take extra parameters
  inherited attributes
- . . . and mostly just pass values up or down
  defaulting mechanism



our *Attribute Grammar Compiler* makes this easy

Tools: Ruler

```
sem_Expr_App func_ arg_ =
  (λ_lhsIknTy
    _lhsIopts
    _lhsItyGam
    _lhsIvalGam →
      (case (_lhsIvalGam) of
        { _argOvalGam →
        (case (_lhsItyGam) of
          { _argOtyGam →
          (case (_lhsIopts) of
            { _argOopts →
            (case (_lhsIvalGam) of
              { _funcOvalGam →
              (case (_lhsItyGam) of
                { _funcOtyGam →
                (case (_lhsIopts) of
                  { _funcOopts →
                  (case ([Ty_Any] `mkArrow` _lhsIknTy) of
                    { _funcOknTy →
                    (case ((func_ _funcOknTy _funcOopts _funcOtyGam _funcOvalGam)) of
                      { (_funcIappArgPPL, _funcIappFunNm, _funcIappFunPP, _funcIerrSq, _funcIpp, _funcIppAST, _f
                      (case (tyArrowArgRes _funcIty) of
                        { __tup2 →
                        (case (__tup2) of
                          { (_ty_a_, _) →
                          (case (_ty_a_) of
                            { _argOknTy →
                            (case ((arg_ _argOknTy _argOopts _argOtyGam _argOvalGam)) of
                              { (_argIappArgPPL, _argIappFunNm, _argIappFunPP, _argIerrSq, _argIpp, _argIppAST
                              (case (_funcIappArgPPL ++ [_argIpp]) of
                                { _lhsOappArgPPL →
                                { _lhsOappFunNm →
```

Tools: Ruler

```
sem_Expr_App func_ arg_ =
  (λ _lhsIknTy
    _lhsIopts
    _lhsItyGam
    _lhsIvalGam →
      (case (_lhsIvalGam) of
        { _argOvalGam →
        (case (_lhsItyGam) of
          { _argOtyGam →
          (case (_lhsIopts) of
            { _argOopts →
            (case (_lhsIvalGam) of
```

```
data Expr
    | App func : Expr
          arg  : Expr
attr AllExpr [knTy : Ty || ty : Ty]
sem Expr
    | App func.knTy             = [Ty_Any] `mkArrow` @lhs.knTy
          (loc.ty_a_, loc.ty_)  = tyArrowArgRes @func.ty
          arg .knTy             = @ty_a_
          loc .ty               = @ty_
```

```
pAST , _f
{ (_argIappArgPPL, _argIappFunNm, _argIappFunPP, _argIerrSq, _argIpp, _argIppAST ,
(case (_funcIappArgPPL ++ [_argIpp]) of
   { _lhsOappArgPPL →
     { _lhsOappFunNm →
```

Tools: Ruler

```
sem_Expr_App func_ arg_ =
  (λ_lhsIknTy
    _lhsIopts
    _lhsItyGam
    _lhsIvalGam →
      (case (_lhsIvalGam) of
        { _argOvalGam →
        (case (_lhsItyGam) of
          { _argOtyGam →
          (case (_lhsIopts) of
            { _argOopts →
            (case (_lhsIvalGam) of
```

$$\frac{\Gamma; \square \rightarrow \sigma^k \vdash^e e_1 : \sigma_a \rightarrow \sigma \qquad \Gamma; \sigma_a \vdash^e e_2 : \_}{\Gamma; \sigma^k \vdash^e e_1\ e_2 : \sigma} \mathrm{E.APP}_K$$

```
data Expr
  | App func : Expr
        arg  : Expr
attr AllExpr [knTy : Ty || ty : Ty]
sem Expr
  | App func.knTy           = [Ty_Any] `mkArrow` @lhs.knTy
        (loc.ty_a_, loc.ty_) = tyArrowArgRes @func.ty
        arg  .knTy          = @ty_a_
        loc  .ty            = @ty_
```

```
pAST, _f
{ (_argIappArgPPL, _argIappFunNm, _argIappFunPP, _argIerrSq, _argIpp, _argIppAST,
(case (_funcIappArgPPL ++ [_argIpp]) of
  { _lhsOappArgPPL →
  { _lhsOappFunNm →
  (case (_funcIappFunPP) of
```

Tools: Ruler

Now that things are acceptably simple...

$$\frac{\begin{array}{c}\Gamma; \Box \rightarrow \sigma^{k} \vdash^{e} e_1 : \sigma_a \rightarrow \sigma \\ \Gamma; \sigma_a \vdash^{e} e_2 : \_\end{array}}{\Gamma; \sigma^{k} \vdash^{e} e_1\ e_2 : \sigma}\ \text{E.APP}_K$$

Tools: Ruler

Now that things are acceptably simple...

$$\frac{\Gamma; \Box \rightarrow \sigma^k \vdash^e e_1 : \sigma_a \rightarrow \sigma \qquad \Gamma; \sigma_a \vdash^e e_2 : \_}{\Gamma; \sigma^k \vdash^e e_1\ e_2 : \sigma} \ \mathrm{E.APP}_K$$

...we can start to introduce new ideas:

$$\frac{v \text{ fresh} \qquad \Gamma; \mathcal{C}^k; v \rightarrow \sigma^k \vdash^e e_1 : \sigma_a \rightarrow \sigma \leadsto \mathcal{C}_f \qquad \Gamma; \mathcal{C}_f; \sigma_a \vdash^e e_2 : \_ \leadsto \mathcal{C}_a}{\Gamma; \mathcal{C}^k; \sigma^k \vdash^e e_1\ e_2 : \mathcal{C}_a\sigma \leadsto \mathcal{C}_a} \ \mathrm{E.APP}_{HM}$$

Tools: Ruler

Now that things are acceptably simple. . .

$$\frac{\begin{array}{c}\Gamma; \Box \to \sigma^k \vdash^e e_1 : \sigma_a \to \sigma \\ \Gamma; \sigma_a \vdash^e e_2 : \_\end{array}}{\Gamma; \sigma^k \vdash^e e_1\ e_2 : \sigma}\ \text{E.APP}_K$$

. . . we can start to introduce new ideas:

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

APP_{HM}

Tools: Shuffle

How to ensure consistency in:

- 20 language variants of increasing complexity
- code, documentation, test sets, publications, presentations

Tools: Shuffle

How to ensure consistency in:

- 20 language variants of increasing complexity
- code, documentation, test sets, publications, presentations

Shuffle

- source files divided in *chunks*
- each chunk is tagged with
  - variant number
  - name

Tools: Shuffle

How to ensure consistency in:

- 20 language variants of increasing complexity
- code, documentation, test sets, publications, presentations

Shuffle

- source files divided in *chunks*
- each chunk is tagged with
  - variant number
  - name

Shuffle shuffles the chunks,
to extract the input for the compiler and text formatter

Tools: Shuffle

Shuffle combines the best of:

- *#define / #ifdef*
- *#include*
- Literate programming
- Version management

Tools: Shuffle

Shuffle combines the best of:

- *#define* / *#ifdef*
  Shuffle has *hierarchical* variants
- *#include*

- Literate programming

- Version management

Tools: Shuffle

Shuffle combines the best of:

- *#define / #ifdef*
  Shuffle has *hierarchical* variants
- *#include*
  Shuffle can include *parts of* a file
- Literate programming
- Version management

Tools: Shuffle

Shuffle combines the best of:

- *#define* / *#ifdef*
  Shuffle has *hierarchical* variants
- *#include*
  Shuffle can include *parts of* a file
- Literate programming
  Shuffle can combine *multiple sources* and *re-use* code
- Version management

Tools: Shuffle

Shuffle combines the best of:

- *#define / #ifdef*
  Shuffle has *hierarchical* variants
- *#include*
  Shuffle can include *parts of* a file
- Literate programming
  Shuffle can combine *multiple sources* and *re-use* code
- Version management
  Shuffle does *variant* management
  versions are historically grown
  variants are *didactically chosen*

Project status

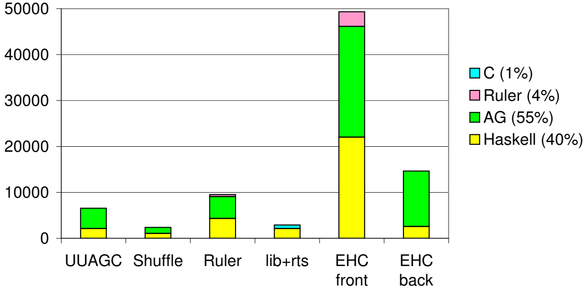
Status of the Essential Haskell Compiler

- Available on `www.cs.uu.nl/wiki/Ehc`

Project status

Status of the Essential Haskell Compiler

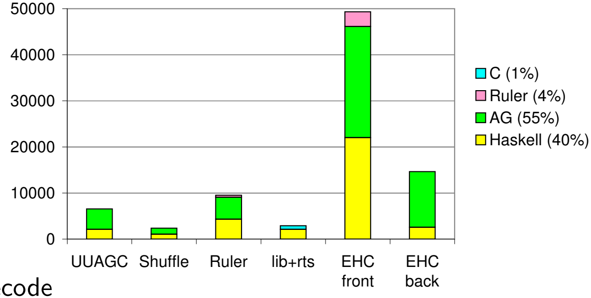
- Available on `www.cs.uu.nl/wiki/Ehc`
- 85000 lines of code, half of which in AG



Project status

Status of the Essential Haskell Compiler

- Available on `www.cs.uu.nl/wiki/Ehc`
- 85000 lines of code, half of which in AG
- Working towards full Haskell with full prelude
- Simple programs compile and run
  - as interpreted bytecode
  - as compiled code



Summary

Coping with Compiler Complexity

in the Essential Haskell Compiler

Summary

Coping with Compiler Complexity

- Implementation complexity
- Description complexity
- Design complexity
- Maintenance complexity

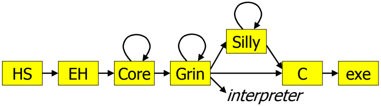
in the Essential Haskell Compiler

Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
- Design complexity
- Maintenance complexity

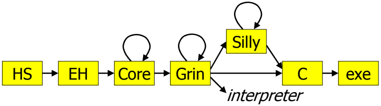
in the Essential Haskell Compiler

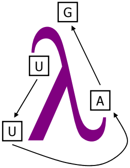


Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
  Use tools!
- Design complexity
- Maintenance complexity



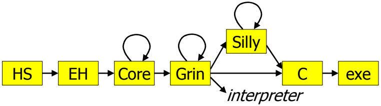


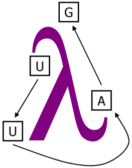
in the Essential Haskell Compiler

Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
  Use tools!
- Design complexity
  Grow stepwise!
- Maintenance complexity





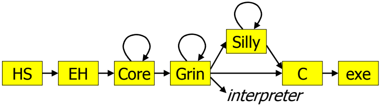
in the Essential Haskell Compiler

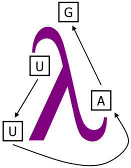
Summary

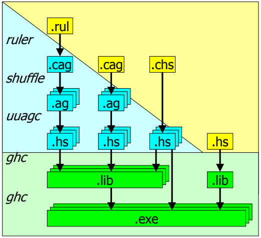
Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
  Use tools!
- Design complexity
  Grow stepwise!
- Maintenance complexity
  Generate, generate, generate!

in the Essential Haskell Compiler



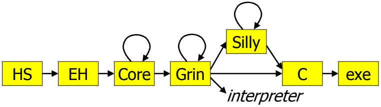


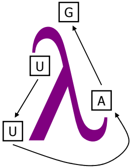


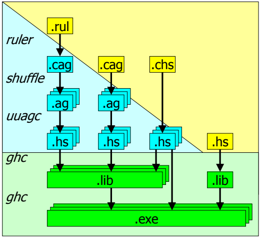
Summary

Coping with Compiler Complexity

- Implementation complexity
  Transform!
- Description complexity
  Use tools!
- Design complexity
  Grow stepwise!
- Maintenance complexity
  Generate, generate, generate!







in the Essential Haskell Compiler

`www.cs.uu.nl/wiki/Ehc`