Trees That Grow

Shayan Najd
Laboratory for Foundations of Computer Science,
The University of Edinburgh

Chalmers University, December 2016

Simon
Peyton Jones

Summer of Haskell 2016

Simon
Peyton Jones

Summer of Haskell 2016

Jacques Carette

Richard Eisenberg

Alan Zimmerman

Niklas Broberg

Summer of Haskell 2016

Haskell Community

Edward Kmett

Ryan Trinkle

GHC

GHC

- Haskell
  Abstract Syntax Tree

GHC

- AST

GHC

- AST
- Parser

GHC

- AST
- Parser
- Printer

GHC

- AST
- Parser
- Printer
- ...

Haskell Suite
(Haskell-Src-Exts)

GHC

- AST
- Parser
- Printer
- ...

Haskell Suite
(Haskell-Src-Exts)

- AST

GHC

- AST
- Parser
- Printer
- ...

Haskell Suite
(Haskell-Src-Exts)

- AST
- Parser

GHC

- AST
- Parser
- Printer
- ...

Haskell Suite
(Haskell-Src-Exts)

- AST
- Parser
- Printer

GHC

- AST
- Parser
- Printer
- ...

Haskell Suite
(Haskell-Src-Exts)

- AST
- Parser
- Printer
- ...

GHC

- AST
- Parser
- Printer
- ...

Haskell Suite
(Haskell-Src-Exts)

- AST
- Parser
- Printer
- ...

GHC

- AST
- Parser
- Printer
- ...

Template Haskell

Haskell Suite (Haskell-Src-Exts)

- AST
- Parser
- Printer
- ...

GHC

- AST
- Parser
- Printer
- ...

Template Haskell

- AST

Haskell Suite (Haskell-Src-Exts)

- AST
- Parser
- Printer
- ...

GHC

- AST
- Parser
- Printer
- ...

Template Haskell

- AST
- Parser

Haskell Suite (Haskell-Src-Exts)

- AST
- Parser
- Printer
- ...

GHC

- AST
- Parser
- Printer
- ...

Template Haskell

- AST
- Parser
- Printer

Haskell Suite (Haskell-Src-Exts)

- AST
- Parser
- Printer
- ...

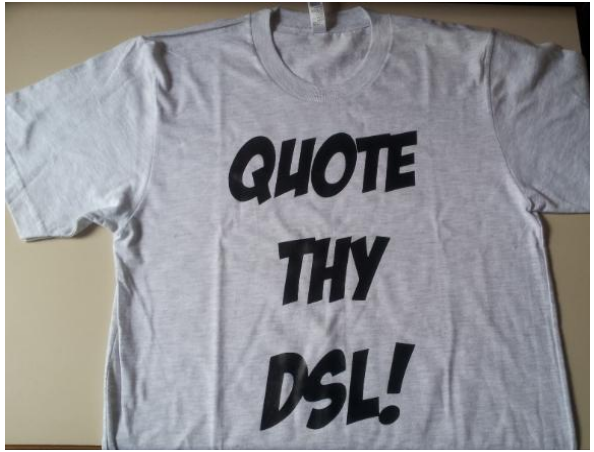
GHC

- AST
- Parser
- Printer
- ...

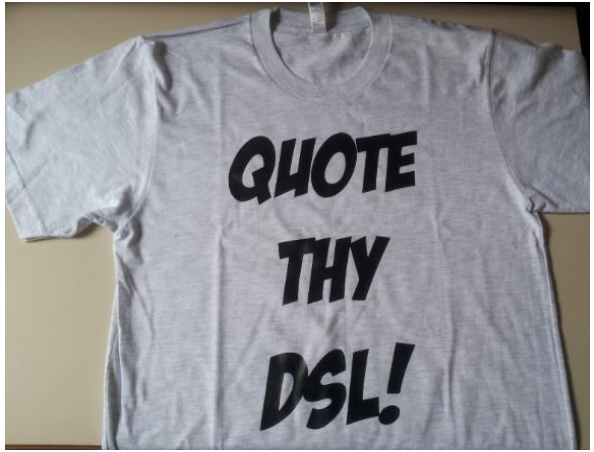
Template Haskell

- AST
- Parser
- Printer
- ...

Quoted Domain-Specific Languages (QDSLs)

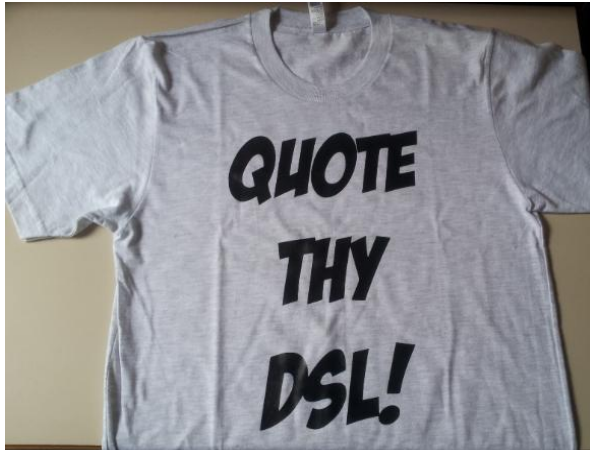


Quoted Domain-Specific Languages (QDSLs)



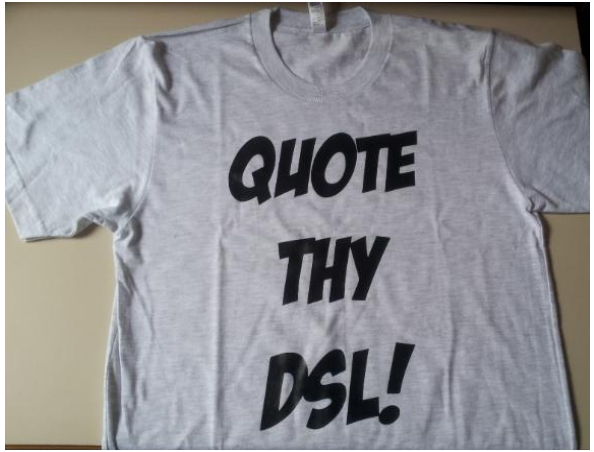
- AST

Quoted Domain-Specific Languages (QDSLs)



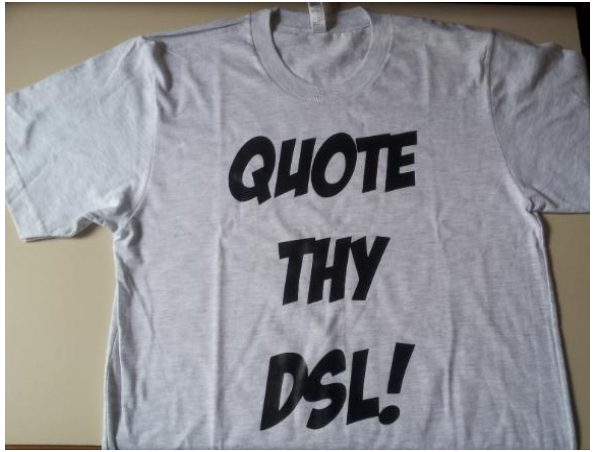
- AST
- Parser

Quoted Domain-Specific Languages (QDSLs)



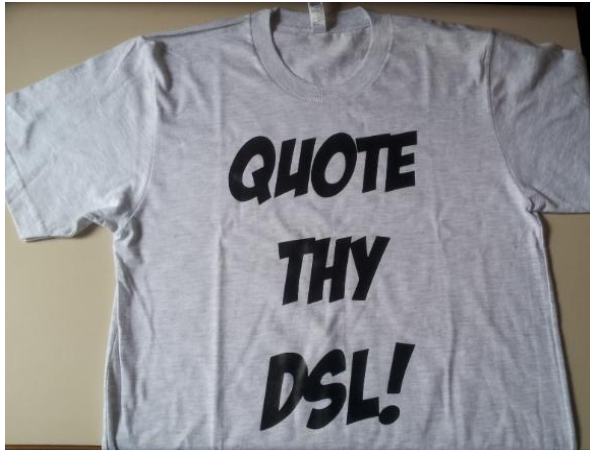
- AST
- Parser
- Printer

Quoted Domain-Specific Languages (QDSLs)



- AST
- Parser
- Printer
- ...

Quoted Domain-Specific Languages (QDSLs)



- AST
- Parser
- Printer
- Type Inference Engine
- Desugaring Machinery
- ...


ENOUGH!

GHC

- AST

GHC

- AST
- Parser

GHC

- AST
- Parser
- Printer

GHC

- AST
- Parser
- Printer
- ...

GHC

- AST
- Parser
- Printer
- Type Inference Engine
- Desugaring Machinery
- ...


Oskars
SURSTRÖMMING
KYLKONSERV SKALL FÖRVARAS KALLT
(+4° till +6°)

The Problem

Haskell Suite

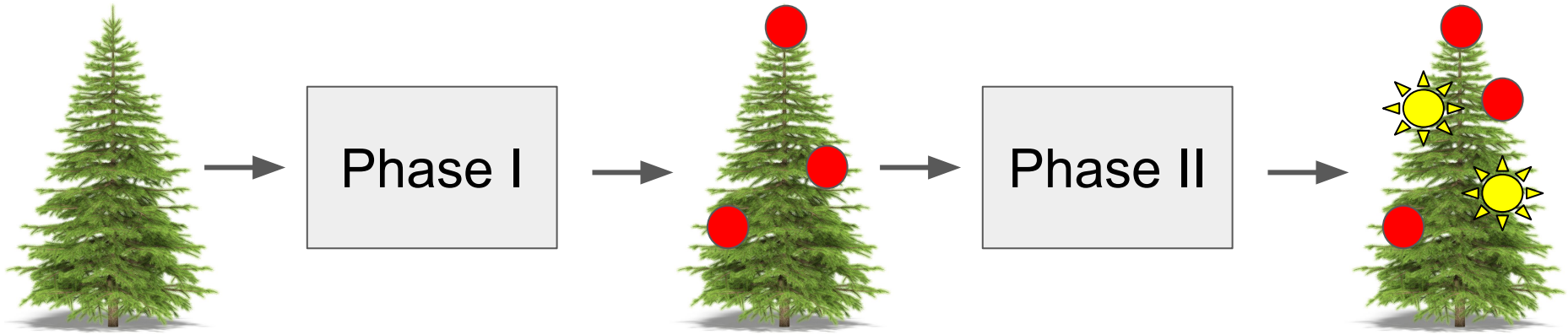
GHC

Template Haskell

Tree-Decoration Problem

V.S.

Compilers



Tree-Decoration Problem

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

Undecorated

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

Decorated

V.S.

Tree-Decoration Problem

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

Undecorated

V.S.

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

Decorated

Tree-Decoration Problem

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

Undecorated

+

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

Decorated

V.S.

Tree-Decoration Problem


```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

Undecorated

V.S.

Decorated

Tree-Decoration Problem

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

Undecorated

+

+

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

Decorated

V.S.

Common Practice

Fully Decorated

```
data Exp id
  = Lit    Integer
  | Var    id
  | Ann    (Exp id) Typ
  | Abs    id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

→ Phase I →

Fully Decorated

```
data Exp id
  = Lit    Integer
  | Var    id
  | Ann    (Exp id) Typ
  | Abs    id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

→ Phase II →

Fully Decorated

```
data Exp id
  = Lit    Integer
  | Var    id
  | Ann    (Exp id) Typ
  | Abs    id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

V.S.

Undecorated

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

→ Phase I →

Partly Decorated

```
data Exp id
  = Lit    Integer
  | Var    id
  | Ann    (Exp id) Typ
  | Abs    id       (Exp id)
  | App Typ (Exp id) (Exp id)
```

→ Phase II →

Fully Decorated

```
data Exp id
  = Lit    Integer
  | Var    id
  | Ann    (Exp id) Typ
  | Abs    id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

Common Practice

Fully Decorated

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

→ Phase I →

Fully Decorated

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

→ Phase II →

Fully Decorated

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

✗ Unnecessary Information & Dependencies

✓ No Duplication

Common Practice

✓ No Unnecessary Information & Dependencies

✗ Duplication

Undecorated

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

→ Phase I →

Partly Decorated

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
```

→ Phase II →

Fully Decorated

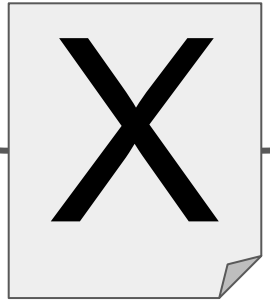
```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

Our Approach:

(1) declare ASTs using ***extensible data types***
(2) define decorations as extensions

```haskell
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```haskell
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```



Common Practice

✓ No Unnecessary Information & Dependencies

✗ Duplication

Undecorated

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

→ Phase I →

Partly Decorated

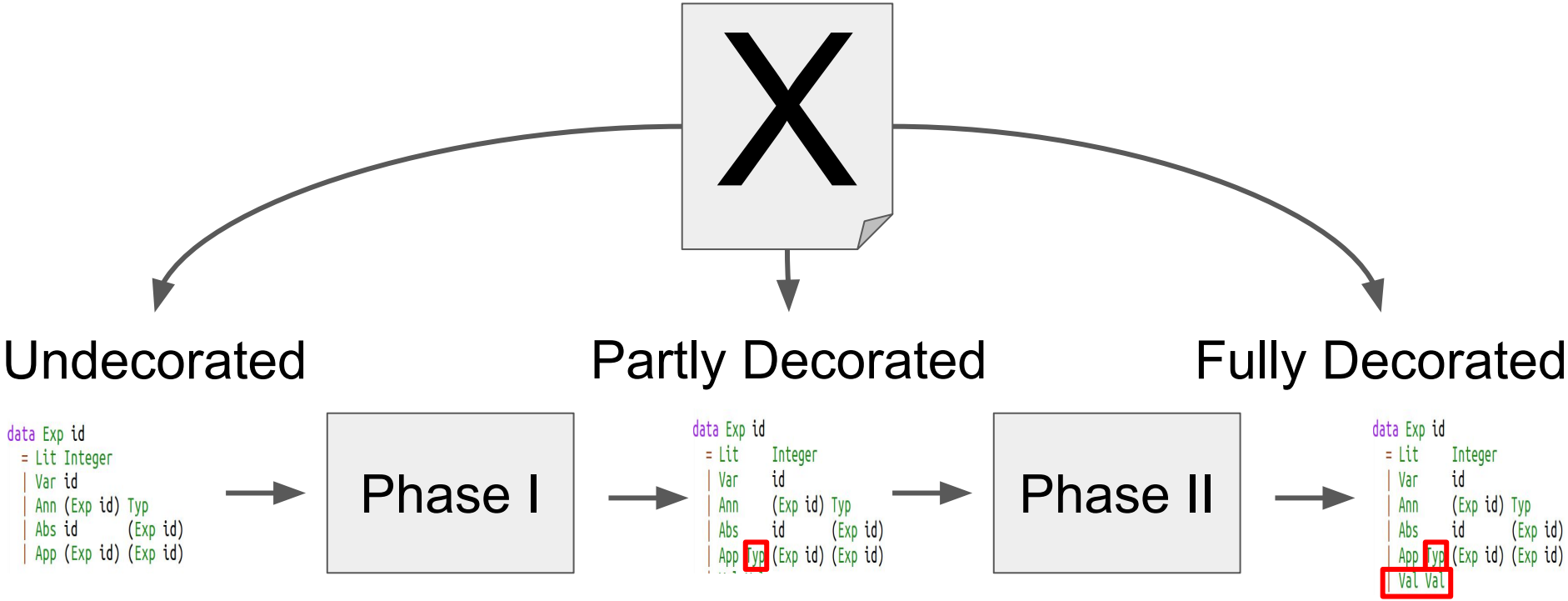
```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
```

→ Phase II →

Fully Decorated

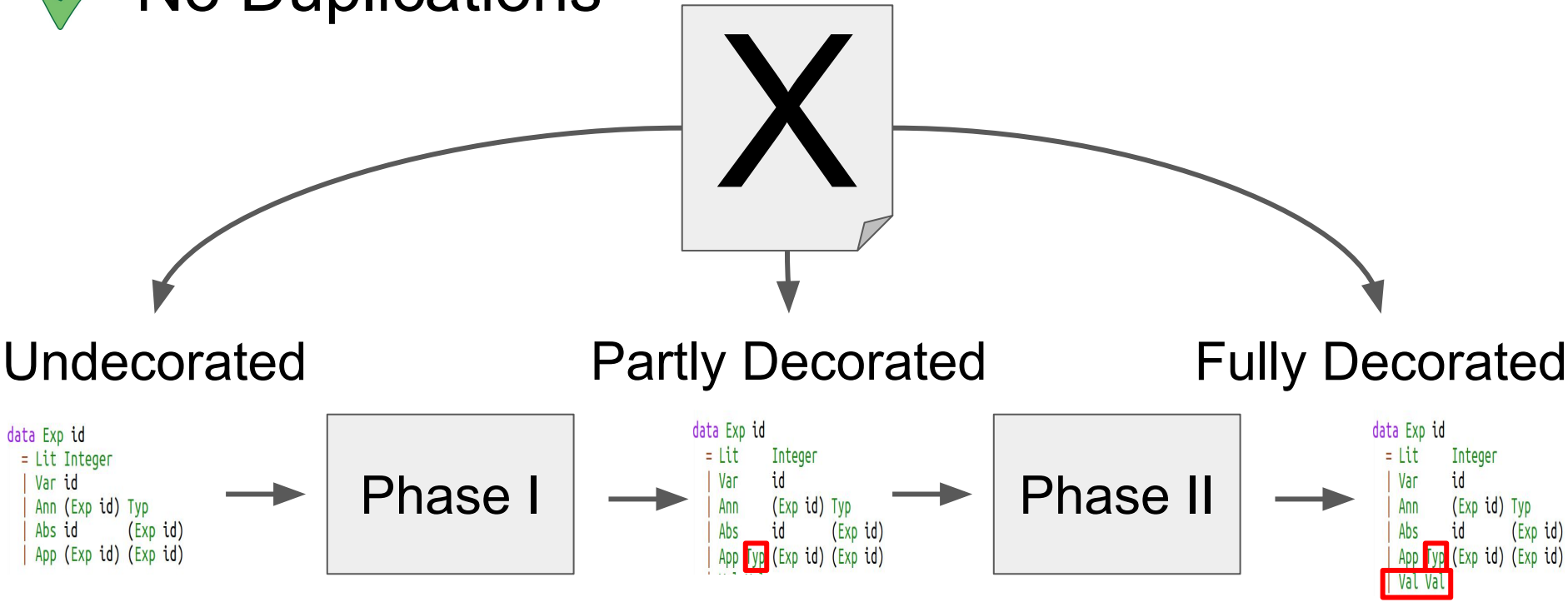
```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

Our Approach



Our Approach

- ✓ No Unnecessary Information & Dependencies
- ✓ No Duplications



Key Challenge:

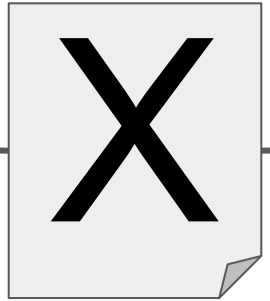
How do we declare eXtensible data types, where GHC does not support them off-the-shelf?

Key Idea:

Use same old parameterisation for extensibility!

```haskell
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```haskell
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```



```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

```
data ExpX   id
  = LitX                Integer
  | VarX                id
  | AnnX                (ExpX   id) Typ
  | AbsX                id          (ExpX   id)
  | AppX                (ExpX   id) (ExpX   id)
  | NewX
```

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```
data ExpX   id
  = LitX  ∅   Integer
  | VarX  ∅   id
  | AnnX  ∅   (ExpX   id) Typ
  | AbsX  ∅   id          (ExpX   id)
  | AppX  ∅   (ExpX   id) (ExpX   id)
  | NewX  ∅*
```

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

```
data ExpX   id
  = LitX  ∅   Integer
  | VarX  ∅   id
  | AnnX  ∅   (ExpX   id) Typ
  | AbsX  ∅   id          (ExpX   id)
  | AppX  Typ (ExpX   id) (ExpX   id)
  | NewX  Val
```

Our Encoding

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

```
data ExpX   id
  = LitX                Integer
  | VarX                id
  | AnnX                (ExpX   id) Typ
  | AbsX                id          (ExpX   id)
  | AppX                (ExpX   id) (ExpX   id)
  | NewX
```

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

```
data ExpX x id
  = LitX (XLit x id) Integer
  | VarX (XVar x id) id
  | AnnX (XAnn x id) (ExpX x id) Typ
  | AbsX (XAbs x id) id          (ExpX x id)
  | AppX (XApp x id) (ExpX x id) (ExpX x id)
  | NewX (XNew x id)
```

```haskell
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```haskell
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

```haskell
data ExpX x id
  = LitX (XLit x id) Integer
  | VarX (XVar x id) id
  | AnnX (XAnn x id) (ExpX x id) Typ
  | AbsX (XAbs x id) id          (ExpX x id)
  | AppX (XApp x id) (ExpX x id) (ExpX x id)
  | NewX (XNew x id)
```

```
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

```
data ExpX x id
  = LitX (XLit x id) Integer
  | VarX (XVar x id) id
  | AnnX (XAnn x id) (ExpX x id) Typ
  | AbsX (XAbs x id) id          (ExpX x id)
  | AppX (XApp x id) (ExpX x id) (ExpX x id)
  | NewX (XNew x id)
```

```haskell
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```haskell
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

```haskell
data ExpX x id
  = LitX (XLit x id) Integer
  | VarX (XVar x id) id
  | AnnX (XAnn x id) (ExpX x id) Typ
  | AbsX (XAbs x id) id          (ExpX x id)
  | AppX (XApp x id) (ExpX x id) (ExpX x id)
  | NewX (XNew x id)
```

```haskell
data Exp id
  = Lit Integer
  | Var id
  | Ann (Exp id) Typ
  | Abs id       (Exp id)
  | App (Exp id) (Exp id)
```

```haskell
data Exp id
  = Lit     Integer
  | Var     id
  | Ann     (Exp id) Typ
  | Abs     id       (Exp id)
  | App Typ (Exp id) (Exp id)
  | Val Val
```

```haskell
data ExpX x id
  = LitX (XLit x id) Integer
  | VarX (XVar x id) id
  | AnnX (XAnn x id) (ExpX x id) Typ
  | AbsX (XAbs x id) id          (ExpX x id)
  | AppX (XApp x id) (ExpX x id) (ExpX x id)
  | NewX (XNew x id)
```

Demo

Extensions? I’ve heard that before!

- Nominal vs Structural
- Syntactic Completeness:
  rows **and** columns (and …)
- Within Haskell **now**
- Generic programming is a plus,
  not a must

Current Status

- Extensible HsSyn AST
- Liberated Parser from GHC
- Automation Using Template Haskell
- Syntax

Next Steps

- Performance Tests
- Splitting GHC into packages
- Replacing TH AST
- Direct Reflection in TH
- Extensible data types as language features