Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Program transformation: intermediate representations

Ivo Gabe de Wolff

June 7, 2022

Greek letters

- τ: Tau, types
- ρ: Rho, qualified types or qualified types
- σ: Sigma, type schemes
- Γ: Uppercase gamma, type environment
- ϕ: Phi, annotation
- α: Alpha, type variables
- β: Beta, annotation variables
- ω: Omega, more than once
- ξ: Xi, extension descriptor

Last time

Different ways to do program transformation with Algorithm W:

- Construct a map from variable names to type variables in W, and use type substitution in transformation.
- Tuple Algorithm W with transformation, return a function to do the transformation.

Both are a multipass.

Intermediate representation (IR)

$$\begin{array}{lcl} t & ::= & \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} \\ & | & \lambda x.\, t_1 \ \mid\ t_1 + t_2 \ \mid\ \cdots \\ \widehat{t} & ::= & \mathbf{let}\ x : \sigma = \widehat{t_1}\ \mathbf{in}\ \widehat{t_2}\ \mathbf{ni} \\ & | & \lambda x : \tau.\, \widehat{t_1} \ \mid\ \widehat{t_1} + \widehat{t_2} \ \mid\ \cdots \end{array}$$

Intermediate representation (IR)

$$
\begin{array}{lcl}
t & ::= & \textbf{let}\ x = t_1\ \textbf{in}\ t_2\ \textbf{ni} \\
  & \mid & \lambda x.\, t_1 \;\mid\; t_1 + t_2 \;\mid\; \cdots \\
\widehat{t} & ::= & \textbf{let}\ x : {\color{red}\sigma} = \widehat{t_1}\ \textbf{in}\ \widehat{t_2}\ \textbf{ni} \\
  & \mid & \lambda x : {\color{red}\tau}.\, \widehat{t_1} \;\mid\; \widehat{t_1} + \widehat{t_2} \;\mid\; \cdots
\end{array}
$$

```
data Expr
  = Let Name Expr Expr
  | Lam Name Expr
  | Add Expr Expr
  | ...
data TypedExpr
  = TLet Name TypeScheme TypedExpr TypedExpr
  | TLam Name Type TypedExpr
  | TAdd TypedExpr TypedExpr
  | ...
```

Duplicating data types

We could duplicate the data type, but:

- this will get out of sync and is hard to maintain.
- requires code duplication of utility functions.

In GHC

Variable names are represented differently during various stages of the compiler:

- After parsing: 'Reader names'
- After renaming: 'Name'
- After type checking: 'Id' (Name with type)

In GHC

Variable names are represented differently during various stages of the compiler:

- After parsing: 'Reader names'
- After renaming: 'Name'
- After type checking: 'Id' (Name with type)

Data type for Haskell expressions:

$$
\begin{array}{l}
\mathbf{data}\ \mathbf{HsExpr1} = \mathit{HsVar}\ \mathbf{RdrName} \mid \cdots \\
\mathbf{data}\ \mathbf{HsExpr2} = \mathit{HsVar}\ \mathbf{Name} \mid \cdots \\
\mathbf{data}\ \mathbf{HsExpr3} = \mathit{HsVar}\ \mathbf{Id} \mid \cdots
\end{array}
$$

Parameterize data type

- For lists we have one generic data type, parameterized over the type of the contents.

Parameterize data type

- For lists we have one generic data type, parameterized over the type of the contents.
- Similarly, we can paramaterize the IR over the type of variable names:

Parameterize data type

- For lists we have one generic data type, parameterized over the type of the contents.
- Similarly, we can paramaterize the IR over the type of variable names:

```
data HsExpr id
 = HsVar id
 | HsApp  (HsExpr id) (HsExpr id) | ···
```

(Simplified, actual data type also includes source mapping info.)

Pipeline

$$
\begin{aligned}
&\mathit{parse} :: \mathbf{String} \rightarrow \mathbf{HsExpr\ RdrName}\\
&\mathit{rename} :: \mathbf{HsExpr\ RdrName} \rightarrow \mathbf{HsExpr\ Name}\\
&\mathit{typecheck} :: \mathbf{HsExpr\ Name} \rightarrow \mathbf{HsExpr\ Id}
\end{aligned}
$$

Reusable functions

- We can now create generic functions such as a pretty printer,
- similar to $length$ on polymorphic lists.
- These functions can be parameterized with the semantics or behaviour of type argument id,
- similar to $filter$ on polymorphic lists.

So far

- A generic data type.
- Reusable utility functions.
- But they only differ on the name type.

More annotations

- We need to store more information from type checking in the IR.
- For list expressions ($[1, 2, 3]$), we must store the type of the elements.

```
data HsExpr id
    = HsVar id
    | ...
    | ExplicitList
            Type
            [HsExpr id]
```

- Now the type is always present, but it should only be present after type checking.

Type after type checking

- Type argument *id* tells us whether we are after type checking.
- We can use a type family to store a type only if *id* is **Id**.

```
type family PostTcType id
type instance PostTcType RdrName = ()
type instance PostTcType Name = ()
type instance PostTcType Id = Type
```

Type families

```
type family PostTcType id
type instance PostTcType RdrName = ()
type instance PostTcType Name = ()
type instance PostTcType Id = Type
```

- A *type family* is a type level function.
- *PostTcType* maps **RdrName** and **Name** to (), and **Id** to *Type*.

Back to the data type

Use *PostTcType id* to only store the type after type checking, when *id* is **Id**:

```
data HsExpr id
    = HsVar id
    | ...
    | ExplicitList
              (PostTcType id)
              [HsExpr id]
```

- Now the type is always present, but it should only be present after type checking.

Further generalized

PostTcType can be generalized to:

```
type family PostTc id a
type instance PostTc RdrName a = ()
type instance PostTc Name a = ()
type instance PostTc Id a = a
```

Now it can also be used for other annotations than types.

1. Trees that grow

Trees that grow

By Shayan Najd and Simon Peyton Jones

More flexibility:

- Add new fields
- Add new constructors
- Remove constructors

General idea

Type argument 'id' implies the representation of names, and which annotations are on the AST.

What if we only use the type argument for the latter?

Data type

- Type variable ξ, called the *extension descriptor*, to replace id.
- For each constructor, we have a type family for annotations.

```
data ExpX ξ
    = LitX (XLit ξ) Integer
    | VarX (XVar ξ) Var
    | AbsX (XAbs ξ) Var (ExpX ξ)    -- Abstraction/Lambda
    | AppX (XApp ξ) (ExpX ξ) (ExpX ξ)
type family XLit ξ
type family XVar ξ
type family XAbs ξ
type family XApp ξ
```

Undecorated data type

- Data type UD has no constructors, we only use it on type level.
- All annotations are Void.

```
type ExpUD = ExpX UD
data UD
type instance XLit UD = Void
type instance XVar UD = Void
type instance XAbs UD = Void
type instance XApp UD = Void
```

Void versus ()

- Void is a datatype with 0 constructors.
- It's only inhabitant is $\perp$; a computation that never completes successfully.
- This way $\mathbf{ExpX}\ UD$ is isomorphic to $\mathbf{Exp}$: there is only one annotation possible.

Void versus ()

- () has two inhabitants, but !() has only one.
- Bottom is not an inhabitant of !().
- With strict annotation fields and () as annotation, it is again isomorphic:

```
data ExpX ξ
    = LitX !(XLit ξ) Integer
    | VarX !(XVar ξ) Var
    | AbsX !(XAbs ξ) Var (ExpX ξ)    -- Abstraction/Lambda
    | AppX !(XApp ξ) (ExpX ξ) (ExpX ξ)
type instance XLit UD = ()
type instance XVar UD = ()
type instance XAbs UD = ()
type instance XApp UD = ()
```

Adding annotations

Recall the typing rule for applications:

$$\frac{\Gamma \vdash_{\mathsf{UL}} t_1 : \tau_2 \to \tau \quad \Gamma \vdash_{\mathsf{UL}} t_2 : \tau_2}{\Gamma \vdash_{\mathsf{UL}} t_1\ t_2 : \tau}\ [\textit{t-app}]$$

- We may need to store the argument type τ_2 in application nodes.

```
type ExpTC = ExpX TC
data TC
type instance XLit TC = ()
type instance XVar TC = ()
type instance XAbs TC = ()
type instance XApp TC = Type
```

Adding constructors

Partial application:

Some subtrees will be replaced with constant values

We could add an additional constructor to $\mathbf{Exp}$:

$$\mathbf{data\ Val} = \cdots$$
$$\mathbf{data\ Exp} = \cdots \mid \mathbf{Val\ Val}$$

Type family for adding constructors

Type family **XExp** ξ contains the type of the additional constructor.

```
data ExpX ξ
  = LitX (XLit ξ) Integer
  | VarX (XVar ξ) Var
  | AbsX (XAbs ξ) Var (ExpX ξ)    -- Abstraction/Lambda
  | AppX (XApp ξ) (ExpX ξ) (ExpX ξ)
  | ExpX (XExp ξ)
```

Undecorated data type

All annotations are Void:

```
type ExpUD = ExpX UD
data UD
type instance XLit UD = Void
type instance XVar UD = Void
type instance XAbs UD = Void
type instance XApp UD = Void
type instance XExp UD = Void
```

Data type for partial evaluation

```
type ExpPE = ExpX PE
data PE
type instance XLit PE = Void
type instance XVar PE = Void
type instance XAbs PE = Void
type instance XApp PE = Void
type instance XExp PE = Val
```

Isomorphic

Is $\mathbf{ExpX}\ UD$ still isomorphic to the original $\mathbf{Exp}$?

- The additional constructor was implemented as:
  $\mid ExpX\ (\mathbf{XExp}\ \xi)$
- In $\mathbf{ExpX}\ UD$, the constructor $ExpX$ can still be used:
- $ExpX\ \perp$ is a value of type $\mathbf{XExp}\ UD$.
- So $\mathbf{ExpX}\ UD$ and $\mathbf{Exp}$ are not isomorphic any more.

Isomorphic with strictness

Is $\mathbf{ExpX}\ \mathit{UD}$ still isomorphic to the original $\mathbf{Exp}$?

- By making the field strict, we can prevent this:
  $\mid \mathit{ExpX}\ !\,(\mathbf{XExp}\ \xi)$
- $\perp$ is not a value of $!\mathit{Void}$, so this constructor can now really not be used.
- $\mathbf{ExpX}\ \mathit{UD}$ and $\mathbf{Exp}$ are now isomoprhic.
- The paper doens't talk about strictness, but it does solve some issues they had.

Replacing constructors

A compiler often replaces a chain of applications with a single application, with a list of arguments:

$$\begin{aligned}&\mathbf{data\ Val} = \cdots\\&\mathbf{data\ Exp} = \cdots \mid \mathit{App}\ \mathbf{Exp}\ [\mathbf{Exp}]\end{aligned}$$

Now we need to:

- Remove the old App constructor.
- Add a new constructor.

Add the new constructor

```
type ExpSA = ExpX SA
data SA
type instance XLit SA = Void
type instance XVar SA = Void
type instance XAbs SA = Void
type instance XApp SA = Void
type instance XExp SA = (ExpSA, [ExpSA])
```

Remove the old constructor

- Paper: use module system to hide this constructor.
- Not in the paper, but strictness helps here again!
- We then use () for annotation-less constructors, and *Void* for inaccessible constructors.

Remove the new constructor

```
data ExpX ξ
    = LitX ! (XLit ξ) Integer
    | VarX ! (XVar ξ) Var
    | AbsX ! (XAbs ξ) Var (ExpX ξ)   -- Abstraction/Lambda
    | AppX ! (XApp ξ) (ExpX ξ) (ExpX ξ)
    | ExpX ! (XExp ξ)
type ExpSA = ExpX SA
data SA
type instance XLit SA = ()
type instance XVar SA = ()
type instance XAbs SA = ()
type instance XApp SA = Void
type instance XExp SA = (ExpSA, [ExpSA])
```

Conclusion

- A generic data type with one type argument ξ.
- Type family per constructor for annotations.
- Type family to add an additional constructor.
- We then use () for annotation-less constructors, and $Void$ for inaccessible constructors.

More

Read the paper, if you want to know more about: (not part of the exam)

- Pattern synonyms.
- GADTs for typed expressions.
- Generic functions.
- Use of module systems.

Remaining lectures

- Guest lecture Marco Vassena: type systems for constant time cryptography
- Abstract interpretation
- Fusion for high-level GPGPU programming (Accelerate)
- ?

Guest lectures will be on the exam.