Acknowledgement:
Joost Visser, Simon-Peyton Jones.

# Generic transformations

Ralf Lämmel
Software Language Engineer
University of Koblenz-Landau
Germany

This talk essentially draws from my "old" RULE 2002 paper "Towards Generic Refactoring".

# Method extraction in Java

```
void printOwning(double amount) {
  printBanner ();
  //print details
  System.out.println("name:" + _name);
  System.out.println("amount" + amount);
}
```

⇓

```
???
```

.

.

.

⇓

```
void printDetails(double amount) {
  System.out.println("name:" + _name);
  System.out.println("amount" + amount);
}
void printOwning(double amount) {
  printBanner ();
  printDetails(amount);
}
```

# Variations on generic extraction

| *Paradigm* | *Focus* | *Abstraction* |
|---|---|---|
| OO programming | statements | method |
| OO programming | features | class |
| Functional programming | expression | function |
| Functional programming | type expression | datatype |
| Functional programming | functions | type class |
| Logic programming | literal | predicate |
| Syntax definition | EBNF phrase | nonterminal |
| Preprocessing | code fragment | macro |
| Document processing | content particle | element type |
| Cobol programming | sentences | paragraph |
| Cobol programming | sentences | subprogram |
| Cobol programming | data description entries | copy book |
| Cobol programming | data description entries | group field |

# Generic transformations other than extraction

- Inlining (inverse of extraction)
- Introduction / elimination
- Fold / unfold (similar extract / inline)
- Pull up / push down
- Add parameter
- Reorder
- ...

# Extraction of a Haskell datatype

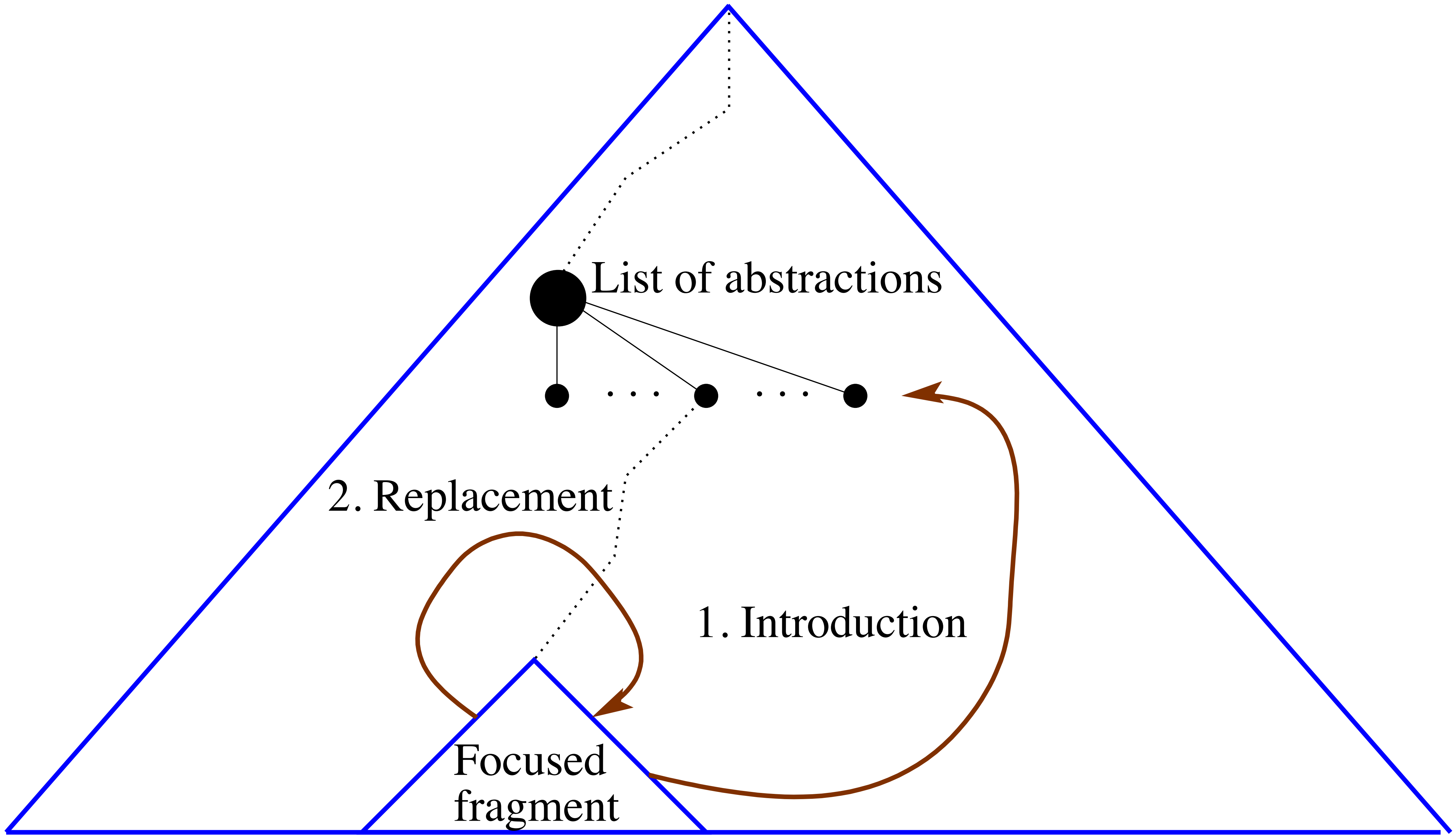
Input program with focus

$$\begin{aligned}
&\mathbf{data}\ \mathit{Prog} &&= \mathit{Prog}\ \mathit{ProgName}\ \boxed{[\mathit{Dec}]\ [\mathit{Stat}]}\\[1em]
&\mathbf{data}\ \mathit{Dec} &&= \mathit{VDec}\ \mathit{Id}\ \mathit{Type} \mid \ldots\\
&\mathbf{data}\ \mathit{Stat} &&= \mathit{Assign}\ \mathit{Id}\ \mathit{Expr} \mid \mathit{If}\ \mathit{Expr}\ \mathit{Stat}\ \mathit{Stat} \mid \ldots
\end{aligned}$$

Output program after extraction and integration

$$\begin{aligned}
&\mathbf{data}\ \mathit{Prog} &&= \mathit{Prog}\ \mathit{ProgName}\ \mathit{Block}\\
&\mathbf{data}\ \mathit{Block} &&= \mathit{Block}\ [\mathit{Dec}]\ [\mathit{Stat}]\\
&\mathbf{data}\ \mathit{Dec} &&= \ldots\\
&\mathbf{data}\ \mathit{Stat} &&= \ldots \mid \mathit{BlockStat}\ \mathit{Block}\\
&\ldots
\end{aligned}$$

# Generic extraction: THE IDEA



# Steps of generic extraction

1. Lookup focused fragment.
2. Determine free names in focused fragment.
3. Enforce language-dependent check on focus.
4. Construct abstraction.
5. Find host for new abstraction.
6. Introduce abstraction.
7. Construct application.
8. Replace focus by application.

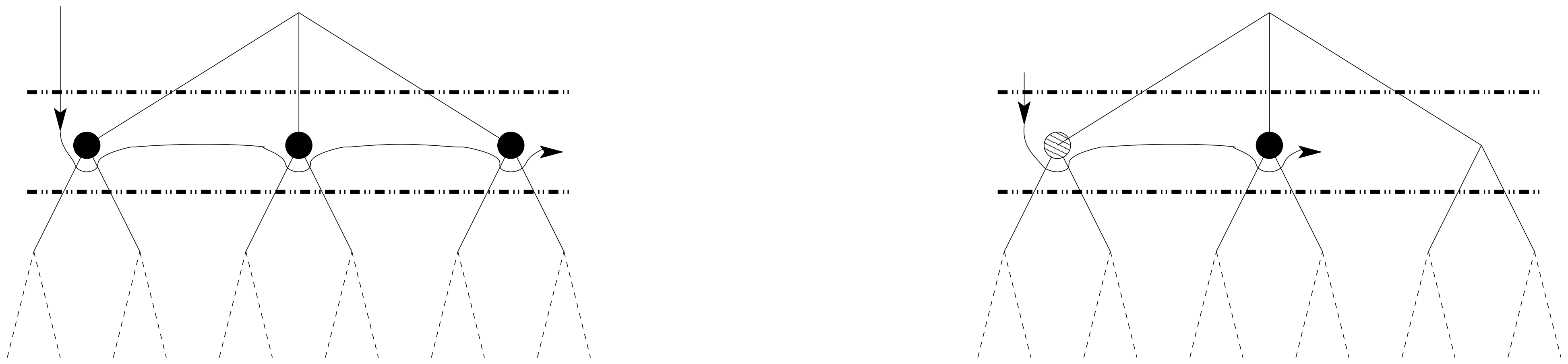
# Too much boilerplate code

## Find the focused statement in a Java fragment

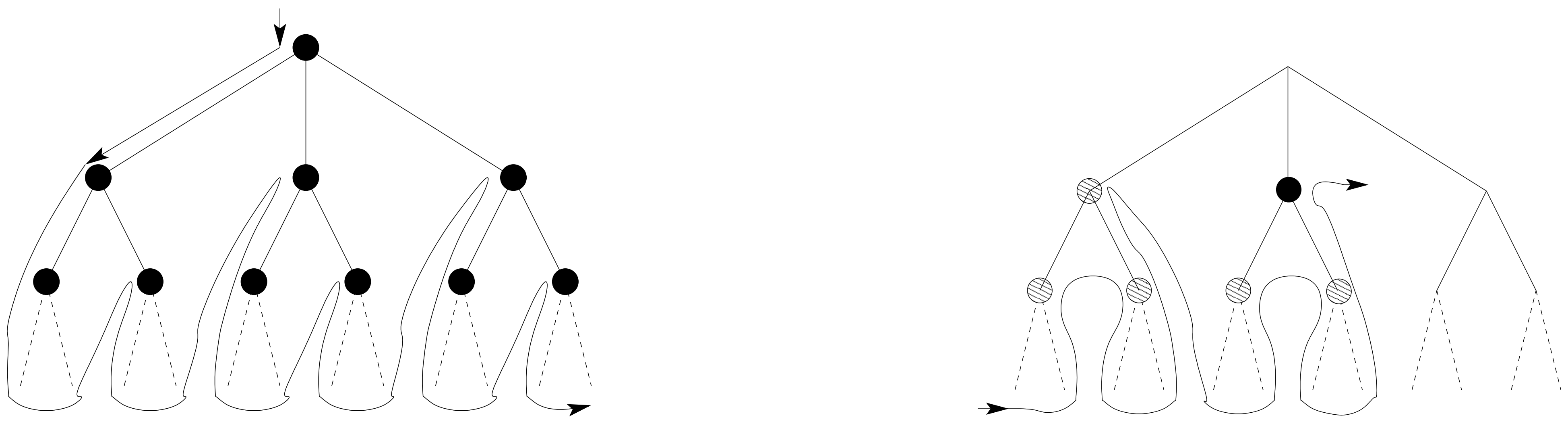
```
selectStatementInStatememt :: Statement → Maybe Statement
selectStatementInStatement (StatementFocus stat) = Just stat
selectStatementInStatement (If _ stat stat′) =
   selectStatementInStatement stat‘orElse′ selectStatementInStatement stat′
 -- continue like this for the dozens of constructors

selectStatementInClass :: Class → Maybe Statement
selectStatementInClass (Class _ _ _ _ _ methods) =
   foldr helper Nothing methods
 where
   helper method focus = focus‘orElse′ selectStatementInMethod method

 -- continue like this for the dozens of types

 -- continue like this for the dozens of components

 -- send me an email when done
```

# (Generic) traversal

## One-layer traversal



## Deep traversal



# No boilerplate code

$$
\begin{aligned}
&\mathit{selectStatement} :: \mathit{Term}\ t \Rightarrow t \rightarrow \mathit{Maybe}\ \mathit{Statement}\\
&\mathit{selectStatement} =\\
&\quad \boxed{\mathit{adhoc}}\ (\mathit{const}\ \mathit{Nothing})\\
&\qquad\qquad\quad (\lambda \mathit{stat} \rightarrow \mathbf{case}\ \mathit{stat}\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{StatementFocus}\ \mathit{stat}' \rightarrow \mathit{Just}\ \mathit{stat}'\\
&\qquad\qquad\qquad\qquad\qquad \_ \rightarrow \mathit{Nothing})\\
&\quad \text{`}\mathit{choice}\text{`}\ \boxed{\mathit{oneTU}}\ \mathit{selectStatement}
\end{aligned}
$$

*adhoc* is for type case.
*oneTU* is for non-recursive traversal to process one child.

# Combinators for generic (strategic) programming

$$\mathit{adhoc}\ \mathit{poly}\ \mathit{mono}\ x\ = \begin{cases} \mathit{mono}\ x, & \text{if } \mathit{typeOf}(x) = \mathit{domOf}(\mathit{mono}) \\ \mathit{poly}\ x, & \text{otherwise} \end{cases}$$

$$\mathit{oneTU}\ \ \mathit{poly}\ (C\ x_1\ \cdots\ x_n)\ =\ \mathit{poly}\ x_1\ \text{`}\mathit{mplus}'\ \cdots\ \text{`}\mathit{mplus}'\ \mathit{poly}\ x_n$$

# *Language-parametric* functionality

A generic piece of refactoring functionality

$$
\begin{aligned}
&\mathit{selectFocus} :: (\mathit{Term}\ f, \mathit{Term}\ t) \\
&\qquad\qquad\quad \Rightarrow (f \to \mathit{Maybe}\ f) && \text{-- Get focus} \\
&\qquad\qquad\quad \to t && \text{-- Input term} \\
&\qquad\qquad\quad \to \mathit{Maybe}\ f && \text{-- Focused term} \\
&\mathit{selectFocus}\ \mathit{unwrap} = \boxed{\mathit{adhoc}}\ (\mathit{const}\ \mathit{Nothing})\ \mathit{unwrap} \\
&\qquad\qquad\qquad\qquad\quad `\mathit{choice}'\ \boxed{\mathit{oneTU}}\ \mathit{selectFocus}
\end{aligned}
$$

A language-specific instantiation

$$
\begin{aligned}
&\mathit{selectStatement} :: \mathit{Term}\ t \Rightarrow t \to \mathit{Maybe}\ \mathit{Statement} \\
&\mathit{selectStatement} = \mathit{selectFocus}\ \mathit{unwrapStatement} \\
&\ \ \mathbf{where} \\
&\quad \mathit{unwrapStatement}\ (\mathit{StatementFocus}\ \mathit{stat}) = \mathit{Just}\ \mathit{stat} \\
&\quad \mathit{unwrapStatement}\ \_ = \mathit{Nothing}
\end{aligned}
$$

# Aspects of generic refactorings

- Generic scope manipulation
- Generic program analyses
- Generic abstract syntax

# Generic scope manipulation

$$
\begin{aligned}
replaceFocus &:: (Term\ t, Term\ t') \\
&\Rightarrow (t \rightarrow Maybe\ t) && \text{-- Transform focus} \\
&\rightarrow t' && \text{-- Input term} \\
&\rightarrow Maybe\ t' && \text{-- Output term} \\
replaceFocus\ trafo &= once\_td\ (adhocTP\ fail\ trafo)
\end{aligned}
$$

# Generic scope manipulation

$$
\begin{aligned}
&\mathit{markHost} :: (\mathit{Term}\ f, \mathit{Term}\ h, \mathit{Term}\ t) \\
&\qquad \Rightarrow (f \rightarrow \mathit{Bool}) && \text{-- Test focus} \\
&\qquad \rightarrow (h \rightarrow h) && \text{-- Wrap host} \\
&\qquad \rightarrow t && \text{-- Input term} \\
&\qquad \rightarrow \mathit{Maybe}\ t && \text{-- Output term} \\
&\mathit{markHost}\ \mathit{testFocus}\ \mathit{wrapHost} = \mathit{host}\ `\mathit{above}`\ \mathit{focus} \\
&\ \ \mathbf{where} \\
&\qquad \mathit{host} = \mathit{adhocTP}\ \mathit{fail}\ (\mathit{Just} \circ \mathit{wrapHost}) \\
&\qquad \mathit{focus} = \mathit{adhocTU}\ \mathit{fail}\ (\mathit{guard} \circ \mathit{testFocus})
\end{aligned}
$$

# Generic program analysis for free names

$$
\begin{array}{ll}
\mathit{gfreeNames} :: (\mathit{Eq}\ n, \mathit{Term}\ \alpha) & \\
\quad \Rightarrow (\forall \alpha.\ \mathit{Term}\ \alpha \Rightarrow \alpha \rightarrow [\,n\,]) & \text{-- Identify declarations} \\
\quad \rightarrow (\forall \alpha.\ \mathit{Term}\ \alpha \Rightarrow \alpha \rightarrow [\,n\,]) & \text{-- Identify references} \\
\quad \rightarrow \alpha & \text{-- Input term} \\
\quad \rightarrow [\,n\,] & \text{-- Free names}
\end{array}
$$

$$
\begin{array}{l}
\mathit{gfreeNames}\ \mathit{declared}\ \mathit{referenced}\ x = \\
\qquad ((\mathit{referenced}\ x) \\
\qquad\ \ `\mathit{union}` \\
\qquad\ \ (\mathit{allTU}\ \ \mathit{union}\ [\,]\ (\mathit{gfreeNames}\ \mathit{declared}\ \mathit{referenced}))\ x \\
\qquad )\ \backslash\backslash\ \mathit{declared}\ x
\end{array}
$$

# Generic abstract syntax for abstractions and applications

**class**

Syntactical domains

$$\begin{array}{ll}
( & \\
\quad \mathit{Term\ abstr}, & \text{-- Term type for abstraction} \\
\quad \mathit{Eq\ name}, & \text{-- Names of abstractions} \\
\quad \mathit{Term\ [abstr]}, & \text{-- Lists of abstractions} \\
\quad \mathit{Term\ apply} & \text{-- Term type for applications} \\
) & \\
\quad \Rightarrow \mathit{Abstraction\ abstr\ name\ tpe\ apply} &
\end{array}$$

# Dependencies between syntactical domains

$$
\begin{array}{lll}
|\; abstr & \longrightarrow & name, \\
abstr & \longrightarrow & tpe, \\
abstr & \longrightarrow & apply, \\
apply & \longrightarrow & name, \\
apply & \longrightarrow & abstr
\end{array}
$$

**where**

## Observers

$$\begin{aligned}
\mathit{getAbstrName} &:: \mathit{abstr} \rightarrow \mathit{Maybe}\ \mathit{name}\\
\mathit{getAbstrParas} &:: \mathit{abstr} \rightarrow \mathit{Maybe}\ [(\mathit{name}, \mathit{tpe})]\\
\mathit{getAbstrBody} &:: \mathit{abstr} \rightarrow \mathit{Maybe}\ \mathit{apply}\\
\mathit{getApplyName} &:: \mathit{apply} \rightarrow \mathit{Maybe}\ \mathit{name}\\
\mathit{getApplyParas} &:: \mathit{apply} \rightarrow \mathit{Maybe}\ [(\mathit{name}, \mathit{tpe})]
\end{aligned}$$

## Constructors

$$\mathit{constrAbstr} :: \mathit{name} \rightarrow [(\mathit{name}, \mathit{tpe})] \rightarrow \mathit{apply} \rightarrow \mathit{Maybe}\ \mathit{abstr}$$
$$\mathit{constrApply} :: \mathit{name} \rightarrow [(\mathit{name}, \mathit{tpe})] \rightarrow \mathit{Maybe}\ \mathit{apply}$$

# Generic extraction, finally

$$
\begin{array}{ll}
\mathit{extract} :: (\mathit{Term}\ \mathit{prog}, \mathit{Abstraction}\ \mathit{abstr}\ \mathit{name}\ \mathit{tpe}\ \mathit{apply}) & \\
\quad \Rightarrow (\forall \alpha.\ \mathit{Term}\ \alpha \Rightarrow \alpha \rightarrow [(\mathit{name}, \mathit{tpe})]) & \text{-- Identify declarations} \\
\quad \rightarrow (\forall \alpha.\ \mathit{Term}\ \alpha \Rightarrow \alpha \rightarrow [\mathit{name}]) & \text{-- Identify references} \\
\quad \rightarrow (\mathit{apply} \rightarrow \mathit{Maybe}\ \mathit{apply}) & \text{-- Unwrap focus} \\
\quad \rightarrow ([\mathit{abstr}] \rightarrow [\mathit{abstr}]) & \text{-- Wrap host} \\
\quad \rightarrow ([\mathit{abstr}] \rightarrow \mathit{Maybe}\ [\mathit{abstr}]) & \text{-- Unwrap host} \\
\quad \rightarrow ([(\mathit{name}, \mathit{tpe})] \rightarrow \mathit{apply} \rightarrow \mathit{Bool}) & \text{-- Check focus} \\
\quad \rightarrow \mathit{name} & \text{-- Name for abstraction} \\
\quad \rightarrow \mathit{prog} & \text{-- Input program} \\
\quad \rightarrow \mathit{Maybe}\ \mathit{prog} & \text{-- Output program}
\end{array}
$$

```
extract declared referenced unwrap wrap unwrap′ check name prog
 = do
    -- Operate on focus
   (bound, focus)   ← boundTypedNames declared unwrap prog
   free             ← return (freeTypedNames declared referenced bound focus)
   guard (check bound focus)

    -- Construct abstraction
   abstr            ← constrAbstr name free focus

    -- Insert abstraction
   prog′            ← markHost (maybe False (const True) ∘ unwrap) wrap prog
   prog″            ← introduce declared referenced unwrap′ abstr prog′

    -- Construct application
   apply            ← constrApply name free

    -- Replace focus by application
   replaceFocus (maybe Nothing (const (Just apply)) ∘ unwrap) prog″
```

# Framework instantiation

- Ingredients of generic algorithms
- Instantiation of *Abstraction* class
- Language-specific checks

Exemplified for Java subset JOOS

# Focus processing for JOOS

Syntax extensions

$$\mathbf{data}\ \mathit{Statement} = \cdots \mid \mathit{StatementFocus}\ \mathit{Statement}$$
$$\mathbf{data}\ \mathit{MethodDecl} = \cdots \mid \mathit{MethodDeclFocus}\ [\mathit{MethodDecl}]$$

Focus on statements and lists of method declarations

$$\mathit{wrapStatement} = \mathit{StatementFocus}$$
$$\mathit{unwrapStatement}\ (\mathit{StatementFocus}\ x) = \mathit{return}\ x$$
$$\mathit{unwrapStatement}\ \_ = \mathit{mzero}$$

$$\mathit{wrapMethods}\ \mathit{xs} = [\mathit{MethodDeclFocus}\ \mathit{xs}]$$
$$\mathit{unwrapMethods}\ [\mathit{MethodDeclFocus}\ \mathit{xs}] = \mathit{return}\ \mathit{xs}$$
$$\mathit{unwrapMethods}\ \_ = \mathit{mzero}$$

# Free-name analysis for JOOS

Datatype for kinds of relevant JOOS identifiers

$$\mathbf{data}\ \mathit{TypeJoos} = \mathit{ExprType}\ \mathit{Type}$$

Declared names (with type)

$$
\begin{array}{l}
\mathit{declaredJoos} :: \mathit{TU}\ [(\mathit{Identifier}, \mathit{TypeJoos})]\ \mathit{Identity} \\
\mathit{declaredJoos} = \mathit{adhocTU}\ (\mathit{adhocTU}\ (\mathit{constTU}\ [\,])\\
\qquad\qquad\qquad\qquad (\mathit{Identity} \circ \mathit{declaredBlock}))\\
\qquad\qquad\qquad\qquad (\mathit{Identity} \circ \mathit{declaredMeth})\\
\quad \mathbf{where}\\
\qquad \mathit{declaredBlock}\ (\mathit{Block}\ \mathit{vds}\ \_)\\
\qquad\ = \mathit{map}\ (\lambda(\mathit{VarDecl}\ t\ i) \rightarrow (i, \mathit{ExprType}\ t))\ \mathit{vds}\\
\qquad \mathit{declaredMeth}\ (\mathit{MethodDecl}\ \_\ \_\ (\mathit{Formals}\ \mathit{fps})\ \_)\\
\qquad\ = \mathit{map}\ (\lambda(\mathit{Formal}\ t\ i) \rightarrow (i, \mathit{ExprType}\ t))\ \mathit{fps}
\end{array}
$$

Further ingredients omitted

# Instance of Abstraction class for JOOS

```
instance Abstraction
           MethodDecl       -- abstr
           Identifier       -- name
           TypeJoos         -- tpe
           Statement        -- apply
```

**where**

-- Observers

$$\mathit{getAbstrName}\ (\mathit{MethodDecl}\ \_\ i\ \_\ \_) = \mathit{Just}\ i$$

$$\mathit{getAbstrParas}\ (\mathit{MethodDecl}\ \_\ \_\ (\mathit{Formals}\ \mathit{fps})\ \_)$$

$$= \mathit{Just}\ (\mathit{map}\ (\lambda(\mathit{Formal}\ t\ i) \rightarrow (i, \mathit{ExprType}\ t))\ \mathit{fps})$$

$$\mathit{getAbstrBody}\ (\mathit{MethodDecl}\ \_\ \_\ \_\ b)$$

$$= \mathit{Just}\ (\mathit{BlockStat}\ b)$$

$$\ldots$$

```
-- Constructors
  constrAbstr n l a
   = maybe Nothing
           (λfps → Just (MethodDecl Nothing n
                           (Formals fps)
                           (toBlock a)))
           (mapM toFormal l)
   where
     ...
```

# Language-specific transformations by parameter passing

Type of transformation on JOOS programs

$$\mathbf{type}\ \mathit{TrafoJoos} = \mathit{Program} \to \mathit{Maybe}\ \mathit{Program}$$

Extraction of a statement to constitute a new method declaration

$$\begin{aligned}
&\mathit{extractJoos} :: \mathit{Identifier} \to \mathit{TrafoJoos}\\
&\mathit{extractJoos} = \mathit{extract}\\
&\qquad\qquad\quad \mathit{declaredJoos}\\
&\qquad\qquad\quad \mathit{referencedJoos}\\
&\qquad\qquad\quad \mathit{unwrapStatement}\\
&\qquad\qquad\quad \mathit{wrapMethods}\\
&\qquad\qquad\quad \mathit{unwrapMethods}\\
&\qquad\qquad\quad \mathit{check}
\end{aligned}$$

$$
\begin{array}{l}
\textbf{where} \\
\quad \mathit{check}\ \_\ f = \mathit{and}\ [\mathit{noReturns}\ f, \mathit{noFrees}\ f] \\
\quad \mathit{noReturns} = \mathit{maybe}\ \mathit{True}\ (\mathit{const}\ \mathit{False}) \circ \\
\qquad\qquad\qquad\qquad \mathit{applyTU}\ (\mathit{oncetd}\ (\mathit{adhocTU}\ \mathit{fail} \\
\qquad\qquad\qquad\qquad\quad (\lambda s \rightarrow \textbf{case}\ s\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{ReturnStat}\ \_ \rightarrow \mathit{Just}\ () \\
\qquad\qquad\qquad\qquad\qquad\quad \_ \rightarrow \mathit{Nothing}))) \\
\quad \mathit{noFrees} = (\equiv)\ [\,] \circ \mathit{freeNames}\ \mathit{declaredJoos}\ \mathit{definedJoos}
\end{array}
$$

# Generic functional programming in Haskell does matter!

- Functions and types
- Higher-order functions
- Generic traversal needed
- Type-class polymorphism
- Mix of genericity and specificity
- ...

Thanks!

Questions?