First Class Rules and Generic Traversals for Program Transformation Languages

Eelco Dolstra
edolstra@students.cs.uu.nl

August 10, 2001
Introduction

- Goal: languages for writing program transformations (compilers, migration, desugarers, optimisers)

- What features?
  - First class rules (separation of rules and strategies ⇒ strategic programming)
  - Generic traversals

- How to implement strategic features in functional languages?
Stratego (1)

Stratego: program transformation language based on separation of rules and strategies

Rules:

plus0: App(App(Var("+"), e), Num(0)) \rightarrow e
comm: App(App(Var("+"), e1), e2) \rightarrow App(App(Var("+"), e2), e1)
beta: App(Lam(x, e1), e2) \rightarrow Let(x, e2, e1)

Strategy:

optimise = bottomup(repeat(
    plus0 + (comm; plus0) + beta
))
Stratego (2)

\[
\begin{align*}
\text{try}(s) &= s \leftarrow \text{id} \\
\text{repeat}(s) &= \text{rec } x(s; x \leftarrow \text{id}) \\
\text{bottomup}(s) &= \text{rec } x(\text{all}(x); s) \\
\text{topdown}(s) &= \text{rec } x(s; \text{all}(x)) \\
\text{oncetd}(s) &= \text{rec } x(s \leftarrow \text{one}(x))
\end{align*}
\]

Generic traversal primitives:

- **all(s)**: Apply \(s\) successfully to all subterms
- **one(s)**: Apply \(s\) successfully to exactly one subterm
RhoStratego

RhoStratego is a non-strict pure functional language with:

- Constructors and pattern matching, e.g.

  plus0 = App (App (Var "+") e) (Num 0) -> e;
  comm = App (App (Var "+") e1) e2 ->
         App (App (Var "+") e2) e1;
  beta = App (Lam x e1) e2 -> Let x e2 e1;

- Failure and a choice operator

- A generic traversal primitive
Choice

If a pattern match fails, the result is fail, e.g.
\( \text{beta (Var "foo") } \Rightarrow \text{fail} \)

The choice operator \( <+ \) first tries its left alternative, and then its right alternative if the left one fails.

\[ f = \text{plus0} <+ (\text{comm } \mid \text{plus0}); \]

\( (\mid = \text{sequential composition, left-to-right}) \)
Distribution

We choose between functions **applied to values**. The distribution rule pushes arguments into choice alternatives:

\[
\text{DISTRIB} : (e_1 <+ e_2) \ e_3 \mapsto e_1 \ e_3 <+ e_2 \ e_3
\]

Alternative would be to do this manually:

\[
f = x \rightarrow (\text{plus0} \ x <+ (\text{comm} \mid \text{plus0}) \ x);
\]
Failure / cuts

We can pattern match against failure:

\[
x = (\text{fail} \rightarrow 123) \text{fail}; \quad \Rightarrow 123
\]
\[
x = (\text{fail} \rightarrow 123) \text{456}; \quad \Rightarrow \text{fail}
\]

Sometimes we want to let failure or a function “escape” out of a left alternative \( \Rightarrow \) cuts

Example: strict application

\[
st = f \rightarrow ((\text{fail} \rightarrow \text{fail}) <+ f);
\]
Pattern matching (1)

Redundant:

- Case

\[
f = x \rightarrow \text{case } x \text{ of } \\
\quad A \rightarrow 123; \\
\quad \text{B } "\text{foo}" \rightarrow 456; \\
\quad \_ \rightarrow 0;
\]

⇒

\[
a = A \rightarrow 123; \\
b = \text{B } "\text{foo}" \rightarrow 456; \\
c = y \rightarrow 0; \\
f = a <+ b <+ c;
\]

- Equational style

- Pattern guards

- Views / transformational patterns (somewhat)
Pattern matching (2)

Pattern guards: instead of

\[
\begin{align*}
  f \text{ env var } | \text{isJust (lookup env var)} &= \text{fromJust (lookup env var)} \\
  f \text{ env var} &= 0
\end{align*}
\]

\[
\begin{align*}
  f \text{ env var } | \text{Just x <- lookup env var} &= x \\
  f \text{ env var} &= 0
\end{align*}
\]

In RhoStratego

\[
f = \text{env } \rightarrow \text{var } \rightarrow (\text{lookup env var <+ 0});
\]
Pattern matching (3)

Transformational patterns:

\[
f(x:xs)!reverse = x \\
f[]!reverse = 0
\]

In RhoStratego, given

\[
\text{snoc} = \text{reverse} \mid ((x:xs) \rightarrow <x, xs>); \\
\text{lin} = [] \rightarrow <>;
\]

we can write

\[
f = \{\text{snoc}\} x xs \rightarrow x <+ \{\text{lin}\} \rightarrow 0;
\]

Compare this with

\[
f = \text{Cons} x xs \rightarrow x <+ \text{Nil} \rightarrow 0;
\]

\(f\) is desugared into:

\[
f = y \rightarrow (<x, xs> \rightarrow x)(\text{snoc } y) \\
<+ y \rightarrow (<> \rightarrow 0)(\text{lin } y);
\]
Generic traversals (1)

- Generic traversals are implemented using application pattern matches.

- Allows deconstruction of construction applications:

  \[ f = (c \ x \to\ c) \ (A \ B \ C); \quad // = A \ B \]
  \[ g = (c \ x \to\ x) \ (A \ B \ C); \quad // = C \]

- Traverse arguments linearly, e.g.:

  \[ \text{termSize} = c \ x \to\ \text{termSize} \ c + \text{termSize} \ x \]
  \[ \quad <+ \quad x \to\ 1; \]
We can now write all and one:

\[
all = f \rightarrow (c \ x \rightarrow \ ^{\text{st}}(\text{all} \ f \ c) \ (f \ x)) \leftarrow \text{id});
\]

\[
one = f \rightarrow c \ x \rightarrow (\text{st} \ c \ (f \ x) \leftarrow \text{one} \ f \ c \ x);
\]

And more complex traversals:

\[
topdown = s \rightarrow s \mid \text{all} \ (\text{topdown} \ s);
\]
\[
bottomup = s \rightarrow \text{all} \ (\text{bottomup} \ s) \mid s;
\]
\[
oncetd = s \rightarrow (s \leftarrow \text{one} \ (\text{oncetd} \ s));
\]
\[
force = \text{all} \ force;
\]
Type system (1)

Type preserving: e.g., all, one, topdown

$$\forall \beta. (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$

Type unifying: e.g., collect

$$\forall \alpha. \forall \beta. (\forall \gamma. \gamma \rightarrow \beta) \rightarrow \alpha \rightarrow [\beta]$$
Type system (2)

How to type all?

all = f →
(c x → ^(st (all f c) (f x)) <+ id);

From the assumptions

all :: \(\forall \beta. (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta\)
f :: \(\forall \alpha. \alpha \rightarrow \alpha\)
c :: \(\tau_2 \rightarrow \tau_1\)
x :: \(\tau_2\)

we can derive:

all f :: \(\tau_3 \rightarrow \tau_3\)
all f c :: \(\tau_2 \rightarrow \tau_1\)
f x :: \(\tau_2\)
all f c (f x) :: \(\tau_1\)

But we should be careful; consider c x → x (with type \(\forall \alpha. \forall \beta. \alpha \rightarrow \beta\))
Type system (3)

**Generic:**

\[
\begin{align*}
n & \geq 1 \land x_0 : (\alpha_1 \to \ldots \to \alpha_n \to \alpha_0) \in \Gamma \land \\
x_1 : \alpha_1 & \in \Gamma \ldots \land x_n : \alpha_n \in \Gamma \\
\Gamma \vdash_p (x_0 \ x_1 \ldots \ x_n) : \text{Gen}(\alpha_0, \alpha_1, \ldots, \alpha_n)
\end{align*}
\]

**Contract:**

\[
\begin{align*}
\Gamma \vdash e : \tau[\text{Gen}(\alpha_0, \alpha_1, \ldots, \alpha_n)] \land \\
(\forall i, 0 \leq i \leq n : \alpha_i \notin \text{fv}(\Gamma)) \land \\
(\forall i, 1 \leq i \leq n : \alpha_i \notin \text{fv}(\tau))
\end{align*}
\]

\[
\Gamma \vdash e : \tau[\alpha_0]
\]

Now \( c \ x \rightarrow \text{all} \ f \ c \ (f \ x) \) gets type \( \text{Gen}(\tau_0, \tau_1) \rightarrow \tau_0 \) (using \text{GENERIC}) which becomes \( \tau_0 \rightarrow \tau_0 \) (using \text{CONTRACT}).
Type system (4)

How do we use all et al.? Argument has type $\forall \alpha. \alpha \to \alpha$.

$\Rightarrow$ runtime mechanism

rename =
  todown (try (Exp?Var "x" -> Var "y"));

Type of Exp?Var "x" -> Var "y" is (?Exp) -> Exp, which can be widened into $\alpha \to \alpha$ (and then generalised into $\forall \alpha. \alpha \to \alpha$).

varNames = collect (Exp?Var x -> x);

Type of Exp?Var x -> x is (?Exp) -> String, which becomes $\forall \gamma. \gamma \to String$. 
Type system (5)

**RTTC:**

\[
\Gamma \vdash_p p : \sigma \\
\frac{}{\Gamma \vdash \sigma ? p : ? \sigma}
\]

**Widen:**

\[
\Gamma \vdash e : ? \sigma \rightarrow ([\alpha := \sigma] \tau) \\
\frac{}{\Gamma \vdash e : \alpha \rightarrow \tau}
\]
Implementation

• Interpreter (lazy and strict variants)

• Compiler (to C)

• Type inferencer

• Standard library; reads and writes ATerms, and so can be easily interfaced with XT
Conclusion

- Application pattern matches are a simple but quite powerful primitive for constructing generic traversals

- Application pattern matches can be typed; type safety of type unifying and type preserving functions is guaranteed

- Allows notation very similar to Stratego (and rewriting)

- Choices liberate pattern matching