First Class Rules and Generic Traversals for Program Transformation Languages

Eelco Dolstra
edolstra@students.cs.uu.nl

August 10, 2001

August 10, 2001

Introduction

- Goal: languages for writing program transformations (compilers, migration, desugarers, optimisers)
- What features?
  - First class rules (separation of rules and strategies $\Rightarrow$ 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)) -> e
comm: App(App(Var("+"), e1), e2) ->
      App(App(Var("+"), e2), e1)
beta: App(Lam(x, e1), e2) -> Let(x, e2, e1)
```

Strategy:

```
optimise = bottomup(repeat(
    plus0 + (comm; plus0) + beta
))
```

Stratego (2)

```
try(s)       = s <+ id
repeat(s)    = rec x(s; x <+ id)
bottomup(s) = rec x(all(x); s)
topdown(s)  = rec x(s; all(x))
oncetd(s)    = rec x(s <+ one(x))
```

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.
`beta (Var "foo")` $\Rightarrow$ `fail`

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

```
f = plus0 <+ (comm | plus0);
```

(| = sequential composition, left-to-right)

Distribution

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

$$\textsc{Distrib} : (e_1 \texttt{ <+ } e_2)\ e_3 \mapsto e_1\ e_3 \texttt{ <+ } e_2\ e_3$$

Alternative would be to do this manually:

```
f = x -> (plus0 x <+ (comm | plus0) x);
```

Failure / cuts

We can pattern match against failure:

```
x = (fail -> 123) fail;   ⇒ 123
x = (fail -> 123) 456;    ⇒ fail
```

Sometimes we want to let failure or a function "escape" out of a left alternative ⇒ cuts

Example: strict application

```
st = f -> ((fail -> ^fail) <+ f);
```

Pattern matching (1)

Redundant:

- Case

```
f = x -> case x of
           A       -> 123;
           B "foo" -> 456;
           _       -> 0;
```

$\Rightarrow$

```
a = A       -> 123;
b = B "foo" -> 456;
c = y       -> 0;
f = a <+ b <+ c;
```

- Equational style
- Pattern guards
- Views / transformational patterns (somewhat)

Pattern matching (2)

Pattern guards: instead of

```
f env var | isJust (lookup env var)
          = fromJust (lookup env var)
f env var = 0

f env var | Just x <- lookup env var = x
f env var = 0
```

In RhoStratego

```
f = env -> var -> (lookup env var <+ 0);
```

Pattern matching (3)

Transformational patterns:

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

In RhoStratego, given

```
snoc = reverse | ((x:xs) -> <x, xs>);
lin = [] -> <>;
```

we can write

```
f = {snoc} x xs -> x <+ {lin} -> 0;
```

Compare this with

```
f = Cons   x xs -> x <+ Nil   -> 0;
```

`f` is desugared into:

```
f =  y -> (<x, xs> -> x) (snoc y)
  <+ y -> (<>      -> 0) (lin y);
```

Generic traversals (1)

- Generic traversals are implemented using **application pattern matches**.

- Allows deconstruction of construction applications:

```
f = (c x -> c) (A B C); // = A B
g = (c x -> x) (A B C); // = C
```

- Traverse arguments linearly, e.g.:

```
termSize = c x -> termSize c + termSize x
         <+   x -> 1;
```

Generic traversals (2)

We can now write `all` and `one`:

```
all = f ->
  (c x -> ^(st (all f c) (f x)) <+ id);

one = f ->
  c x -> (st c (f x) <+ one f c x);
```

And more complex traversals:

```
topdown  = s -> s | all (topdown s);
bottomup = s -> all (bottomup s) | s;
oncetd   = s -> (s <+ one (oncetd s));
force    = all force;
```

Type system (1)

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

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

Type unifying: `e.g., collect`

```
varNames = collect (Var x -> x);
```

$$\forall\alpha.\forall\beta.(\forall\gamma.\gamma \to \beta) \to \alpha \to [\beta]$$

Type system (2)

How to type `all`?

```
all = f ->
  (c x -> ^(st (all f c) (f x)) <+ id);
```

From the assumptions

`all`	::	$\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$
`f`	::	$\forall\alpha.\alpha \to \alpha$
`c`	::	$\tau_2 \to \tau_1$
`x`	::	τ_2

we can derive:

`all f`	::	$\tau_3 \to \tau_3$
`all f c`	::	$\tau_2 \to \tau_1$
`f x`	::	τ_2
`all f c (f x)`	::	τ_1

But we should be careful; consider `c x -> x` (with type $\forall\alpha.\forall\beta.\alpha \to \beta$)

Type system (3)

GENERIC:

$$\frac{\begin{array}{l} n \geq 1 \ \wedge \ x_0 : (\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \alpha_0) \in \Gamma \ \wedge \\ x_1 : \alpha_1 \in \Gamma \ldots \ \wedge \ x_n : \alpha_n \in \Gamma \end{array}}{\Gamma \vdash_p (x_0 \, x_1 \ \ldots \ x_n) : \mathbf{Gen}(\alpha_0, \alpha_1, \ldots, \alpha_n)}$$

CONTRACT:

$$\frac{\begin{array}{l} \Gamma \vdash e : \tau[\mathbf{Gen}(\alpha_0, \alpha_1, \ldots, \alpha_n)] \wedge \\ (\forall i, 0 \leq i \leq n : \alpha_i \notin \mathsf{fv}(\Gamma)) \wedge \\ (\forall i, 1 \leq i \leq n : \alpha_i \notin \mathsf{fv}(\tau)) \end{array}}{\Gamma \vdash e : \tau[\alpha_0]}$$

Now `c x -> all f c (f x)` gets type $\mathbf{Gen}(\tau_0, \tau_1) \rightarrow \tau_0$ (using GENERIC) which becomes $\tau_0 \rightarrow \tau_0$ (using CONTRACT).

Type system (4)

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

$\Rightarrow$ **runtime mechanism**

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

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

Type system (5)

RTTC:

$$\frac{\Gamma \vdash_p p : \sigma}{\Gamma \vdash (\sigma \mathbf{?} p) : \mathbf{?}\sigma}$$

WIDEN:

$$\frac{\Gamma \vdash e : \mathbf{?}\sigma \to ([\alpha := \sigma]\tau)}{\Gamma \vdash e : \alpha \to \tau}$$

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