Functional Stratego*

Eelco Dolstra
edolstra@students.cs.uu.nl

February 6, 2001

Abstract

Stratego is a domain-specific language intended for the construction of program transformation systems. To that end, the language has a number of interesting features; notably, first class pattern matching and generic traversal mechanisms. The question arises whether and how such features can be implemented in functional programming languages and Haskell in particular. In a rewrite system it is essential that one can easily recover from pattern match failure, so that alternative rewrite rules can be tried. Typical functional languages do not support this very well since local pattern match failure leads to global divergence. This article shows how to solve this problem by adding a choice operator to a simple functional language. It seems that this approach is not only useful to strategic programming, but is also more powerful than previous proposals to extend Haskell's pattern matching, such as views, patterns guards, and transformational patterns. Furthermore, we discuss how the generic programming techniques can be employed to implement generic traversals.

1 Introduction

Program transformation systems transform a computer program, typically represented as an abstract syntax tree, from one language to (possibly) another language. Examples of such systems are code generators, optimizers, and application generators (translating a high-level specification into a program in a general purpose language).

There are a number of features that seem to be especially important to the implementation of program transformations systems:

- Pattern matching, so that we can easily deconstruct and inspect values. More importantly, we need to have *first class* pattern matching: if a pattern fails to match, it should be easy to try alternatives.
- Traversal mechanisms. Transformations often need to applied at many points in the abstract syntax tree. We should have generic operations that can traverse arbitrary data structures, applying arbitrary transformations.

*Talk given at the Second Stratego Users Day, February 8, 2001.

Stratego [7] is a domain-specific language intended for the construction of program transformation systems, and as such has extensive matching and traversal features.

The question arises whether and how such features can be implemented in functional programming languages. After all, functional languages like ML and Haskell [4] are well-appreciated for their pattern matching facilities, and recursion makes it relatively easy to traverse a structured value. Unfortunately, pattern matching is not first class: pattern match failure leads to divergence (i.e., a crash) of the entire program. Also, Stratego is untyped, which makes generic traversal primitives like `all` possible; but in typed functional languages, this becomes problematic.

This article proposes a solution to the first problem by means of a choice operator which evaluates to its left argument, unless it fails; otherwise it evaluates to its right argument.

Outline. Section 2 explores the problem space of implementing Stratego-like constructs in Haskell. The addition of a choice operator is discussed in section 3, which also compares this to other proposals to extend pattern matching. Finally, section 4 briefly discusses generic traversals. The reader should be familiar with Stratego and Haskell.

2 Strategic programming in Haskell

It is not difficult to implement Stratego-like functionality. It would be nice if we could write a Stratego rewrite rule like:

```
PlusZero:  Plus(x, Const(0)) → x
```

as

```
plusZero = λ(Plus x (Const 0)) → x                    (1)
```

Of course, this will not work because we cannot handle pattern match failure in λ-abstractions. So we wrap the result in a `Maybe` type:

```
plusZero (Plus x (Const 0)) = Just x
plusZero _ = Nothing
```

In general, then, a strategy has the following type:

```
type Strat α β = α → Maybe β
```

Stratego's left choice and composition operators (here written as `<*`) are trivial:

```
(<+) ::  Strat α β → Strat α β → Strat α β
s1 <+ s2 = λt → maybe (s2 t) Just (s1 t)

(<*) ::  Strat α β → Strat β γ → Strat α γ
s1 <* s2 = λt → maybe Nothing s2 (s1 t)
```

Combining strategies is just as easy as in Stratego. Writing the basic rewrite rules, however, is tiresome because we spend a lot of time packing and unpacking the `Maybe` wrapper. For example, the congruence over a constructor `Add Expr Expr` would look like this:

```
cgrAdd s1 s2 (Add e1 e2) = maybe Nothing (λe1' →
  maybe Nothing (λe2' → Just $ Add e1' e2') (s2 e2)) (s1 e1)
```

Another issue is how to implement generic traversals. We have to describe explicitly for each data type how to traverse over it. Type classes can be used to make this generic:

```
class Trav α where
  allS ::  Strat α α → Strat α α
```

Functions like `bottomup` can then be defined in the usual way:

```
bottomupS ::  Trav α ⇒ Strat α α → Strat α α
bottomupS s = allS (bottomupS s) <* s
```

An instance might look like this:

```
instance Trav Expr where
  allS s (Const n) = Just $ Const n
  allS s (Add e1 e2) =
    case s e1 of
      Just e1' →
        case s e2 of
          Just e2' → Just $ Add e1' e2'
          Nothing → Nothing
      Nothing → Nothing
```

There are more refined (and complex) approaches (see [5]), but these also suffer from the fact that a lot of code must be written for each data type over which we want to traverse.

3 Extending the λ-calculus with a choice operator

3.1 Semantics

In the previous section, example 1 did not work because it is not possible to deal with pattern match failure in the left-hand side of λ-abstractions. In this section I shall give the semantics of a simple (untyped) functional language extended with a choice operator which first tries to evaluate its left argument, and if that fails, evaluates its right argument.

The syntax is the λ-calculus extended with constructed values, pattern matching λ-abstraction, and a choice operator:

$$
\begin{aligned}
&e ::= x \mid ee \mid C \mid \lambda p \rightarrow e \mid e \texttt{ <+ } e \mid \delta \\
&p ::= x \mid pp \mid C
\end{aligned}
$$

where x ranges over the variables, and C over the constructors. The constant δ denotes pattern match failure. The choice operator `<+` binds weaker than λ-abstraction and function application.

The semantics is given by the following rewrite rules:

$$
\begin{array}{rcll}
(\lambda x \rightarrow e_1)e_2 & \Rightarrow & [x := e_2]e_1 & (\beta\text{-reduction}) \\
(\lambda C \rightarrow e_1)C & \Rightarrow & e_1 & (\text{pos. constr. match}) \\
(\lambda C_1 \rightarrow e_1)C_2 & \Rightarrow & \delta \ (\text{if } (C_1 \neq C_2)) & (\text{neg. constr. match}) \\
(\lambda(p_1 p_2) \rightarrow e_1)(e_2 e_3) & \Rightarrow & (\lambda p_1 \rightarrow (\lambda p_2 . e_1)e_3))e_2 & (\text{constr. application}^1) \\
e_1 \texttt{ <+ } e_2 & \Rightarrow & e_1 \ (\text{if NF}(e_1) \neq \delta) & (\text{left choice}) \\
\delta \texttt{ <+ } e & \Rightarrow & e & (\text{right choice}) \\
\delta e & \Rightarrow & \delta & (\text{failure propagation})
\end{array}
$$

where NF(e) denotes the normal form of term e and x' is a fresh variable. These rules suffice to implement a lazy evaluator (we have done so in Stratego for a system extended with let-bindings).

The semantics is given as a set of rewrite rules from the language to the language, i.e., as source transformations. The advantage is that no additional syntax or notation is required. Furthermore, such rules can be employed directly in e.g. an optimizer.

Unfortunately, something seems to be missing. For example, we would like to define the `if` function as follows:

```
if = λTrue e1 e2 → e1  <+  λFalse e1 e2 → e2
```
(2)

expecting that `if False 1 2` $\Rightarrow$ `2`. This is, after all, the behaviour expected from Stratego. However, this obviously will not work, since in the evaluation of `if` the choice operator will immediately pick the *left* side, since `λTrue e1 e2` $\rightarrow$ `e1` $\neq \delta$.

We can get around this by writing instead:

```
if = λc e1 e2 → (
    (λTrue e1 e2 → e1) c e1 e2  <+
    (λFalse e1 e2 → e2) c e1 e2)
```
(3)

This will enforce that `c` is matched against the pattern `True`, and so the left side will fail and the right side will be evaluated instead.

A nicer solution (which is admittedly somewhat *ad hoc*) is to add the following rule:

$$
(\lambda p \rightarrow e_1) \texttt{ <+ } e_2 \quad \Rightarrow \quad \lambda x \rightarrow ((\lambda p \rightarrow e_1)x \texttt{ <+ } e_2 x) \quad (\text{distr. arguments})
$$

and to add the restriction to the left choice rule that e_1 must not be a λ-abstraction.

In other words, this rule pushes arguments into the choice operands until we have a base type (a constructed value). It is easy to see that this rule automatically transforms example 2 into example 3.

We don't lose any expressive power here. Even if we actually want to choose between functions and not results of functions (e.g. in `(if b then f else g) <+ h`), where `b` may fail, we can do this by wrapping the function in a constructor (e.g. `(if b then F f else F g) <+ (F h)`) and unpacking the result later, outside of the scope of the choice operator.

A problem with the choice operator is that we do not have control over the scope of the failure handling, e.g. in `(λFoo → e) <+ ...` we might want the choice operator to catch failure in the `Foo`-match but not in `e`. This problem also confronts Stratego programmers. For example, if we want to rewrite `Foo`-terms and nothing else, we might write in Stratego `try(?Foo; s)`. Unfortunately, any rewrite failure in `s` (including those due to programming errors) will be caught indiscriminately by `try`.

3.2 Comparison to other approaches

The question is what the choice operator gains us, apart from nicer strategic programming. It is well known that regular pattern matching is not perfect [6, 2]. It turns out that the choice operator eliminates the need for many of the proposals to extends Haskell's pattern matching, such as *views*, *pattern guards*, and *transformational patterns*. Furthermore, it makes Haskell's "equational style" (writing a function definition as a number of pattern-guarded equations which must be tried one after another) unnecessary, as well as `case`-expressions.

Equational style unnecessary. Haskell allows us to write patterns not just in λ-abstractors but also in function definitions:

$$
\begin{array}{l}
\mathtt{f}\ p_{11}\ \ldots p_{1n}\ \mathtt{=}\ e_1 \\
\mathtt{f}\ p_{21}\ \ldots p_{2n}\ \mathtt{=}\ e_2 \\
\ldots \\
\mathtt{f}\ p_{m1}\ \ldots p_{mn}\ \mathtt{=}\ e_m
\end{array}
$$

The semantics here are different than in λ-abstractions: if a pattern match fails, the program does not diverge, but instead the next equation is tried. This is an *ad hoc* mechanism that becomes redundant if we have a choice operator:

$$
\begin{array}{l}
\mathtt{f\ =} \\
\quad \lambda p_{11}\ \ldots p_{1n} \to e_1 \quad \mathtt{<+} \\
\quad \lambda p_{21}\ \ldots p_{2n} \to e_2 \quad \mathtt{<+} \\
\quad \ldots \\
\quad \lambda p_{m1}\ \ldots p_{mn} \to e_m
\end{array}
$$

In fact, a major advantage is that pattern matching λ-abstractors are now first class, so we can write:

$$
\begin{aligned}
&\mathtt{f}_1 = \lambda p_{11} \ \ldots p_{1n} \ \rightarrow \ e_1 \\
&\mathtt{f}_2 = \lambda p_{21} \ \ldots p_{2n} \ \rightarrow \ e_2 \\
&\ldots \\
&\mathtt{f}_m = \lambda p_{m1} \ \ldots p_{mn} \ \rightarrow \ e_m \\
&\mathtt{f} = \mathtt{f}_1 \ \ \texttt{<+} \ \ \mathtt{f}_2 \ \ \texttt{<+} \ \ \ldots \ \texttt{<+} \ \ \mathtt{f}_m
\end{aligned}
$$

and we can combine the f_i arbitrarily.

Case-expressions can be eliminated from the language in a similar way.

Views. Views [8] were proposed to address the problem that regular pattern matching is rather limited since we can only match with actual constructors. As a consequence we cannot match against e.g. the end of a list instead of the head, nor can we match against abstract data types since there is simply nothing to match against. For example, using the proposed views for Haskell [1] we can write the following view to match against the end of a list:

```
view Tsil a of [a] = Lin | Snoc y ys where
  tsil xs =
    case reverse xs of
      [] → Lin
      (y:ys) → Snoc y ys
```

where matching against a `Snoc`-constructor causes the function `tsil` to be applied to the value:

```
f (Snoc y _) = y
f Lin = 0
```

It is worth pointing out why views (and transformational patterns) are useful. The reason is that the equational style can only be used if the non-applicability of an equation can be discovered in the pattern. When that is not possible, the equational style falls apart, and we have to explicitly write the traversal through the alternatives (the equations) as a series of ever deeper nested `case`-expressions. The choice operator liberates us from this regime, hence the main motivation for views and transformational patterns disappears. With it, the previous example becomes:

```
f = (λ(y:_) → y  <+  λ[] → 0) .  reverse
```

Views still have the advantage that the transformation to be applied (e.g. `tsil`) is implicit in the name of the patterns (e.g. `Snoc`), but this seems only a minor advantage.

Pattern guards. In Haskell's equational notation, we can use boolean guards to further restrict the applicability of an equation, e.g. `f x | x > 3 = 123`. However, there is a disparity between patterns and guards: patterns can bind variables, whereas guards cannot. For example, if we want to return a variable from an environment, or 0 if it is undefined, we would write:

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

where `lookup` has type `[(`α`, `β`)]` $\to$ α $\to$ `Maybe` β. This is awkward because we now inspect the result of `lookup` twice. Pattern guards [2] redefine a guard as a list of qualifiers, just like in a list comprehension, so that binding can occur:

```
f env var | Just x ← lookup env var = x
f env var = 0                                    (4)
```

But when we have a choice operator, we can just write:

```
f env var = let (Just x) = lookup env var in x  <+  0
```

Alternatively, we could just get rid of the `Maybe` result of `lookup` altogether, making it of type `[(`α`, `β`)]` $\to$ α $\to$ β, and then we get:

```
f env var = lookup env var  <+  0
```

Transformational patterns. Transformational patterns [2] provide a cheap alternative to views, allowing us to write example 4 as:

```
f env (Just x)!(lookup env) = x
f env var = 0
```

Hence, transformational patterns are just view transformations made explicit. The choice operator allows a similar notation:

```
f env = (λ(Just x) → x) .  (lookup env)
   <+  λ_ → 0
```

In general, a definition `f pat!fun = e` can be written as `f = (λpat → e) . fun`.

4 Generic traversals

Given the untyped calculus given in the previous section, it is not hard to define the `all` primitive (which applies a function to each subterm of a term):

```
all  = λf →
     ( λ(c x) → (all f c) (f x)
     <+ λc → c
     )
```

But it would be hard to type this function. Previous work on generic programming, such as Generic Haskell [3], seems not to help us here. For example, in Generic Haskell, we write a generic function by giving a clause for binary sums, binary products, and primitive types, since all data types can be decomposed into a combination of these. Unfortunately, this moves us below the level of constructors. For example, it is not hard to give the primitive clause for an implementation of `bottomup`:

```
bottomup(t) ::  (t → t) → t → t
bottomup(1) f x = f x
```

But we cannot give a clause for e.g. binary products: it is clear that we should apply `bottomup` to the arguments, but then what? At some point `f` should be applied, but only if the binary product is in fact of type `t`, and not just part of something of type `t`.

```
bottomup(a * b) f (x, y) =
  ???  $ (bottomup(a) f x, bottomup(b) f y)
```

It also seems impossible to implement `all`, since a generic function traverses the entire value, not just the top-level subterms.

5 Conclusion

We have discussed two features that seem to be useful to strategic programming in a functional language. First, there is the proposed mechanism to handle pattern match failure, which has the additional advantage of making some proposed extensions to pattern matching redundant. Second, generic traversals are important, but it is not currently clear how these can be implemented elegantly.

References

[1] Warren Burton, Erik Meijer, Patrick Sansom, Simon Thompson, and Phil Wadler. Views: An extension to haskell pattern matching. `http://www.haskell.org/development/views.html`.

[2] Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop*, 2000.

[3] Ralf Hinze. A generic programming extension for Haskell. In *Haskell Workshop*, 1999.

[4] Simon Peyton Jones, John Hughes, et al. Report on the programming language Haskell 98, 1999.

[5] Ralf Lämmel and Joost Visser. Type-safe functional strategies. In *Scottish Functional Programming Workshop*, July 2000.

[6] Mark Tullsen. First class patterns. In *2nd International Workshop on Practial Aspects of Declarative Languages*, number 1753 in LNCS, pages 1–15, 2000.

[7] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, ACM SIGPLAN, pages 13–26, September 1998.

[8] Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *14th ACM Symp. on Principles of Programming Languages*, pages 307–313, 1987.