A strategy language

Johan Jeuring
Joint work with Alex Gerdes and Bastiaan Heeren

Utrecht University and Open Universiteit Nederland
Computer Science

CEFP, Budapest, Hungary

June 2011

Open Universiteit
www.ou.nl

OU - Functional Programming Tutor

http://ideas.cs.uu.nl/ProgTutor/tutor.html?exerciseid=programming.myreverse&description=Write%20a%20function%20that%20reverses%20a%20list:%20

Apple Google Maps Wikipedia

Functional Programming Tutor

Hint | Step | Solution | Ready? | Clear | Back | Back to exercise list

Exercise:

Write a function that reverses a list: myreverse :: [a] -> [a]

```
myreverse = reverse' []
  where reverse' acc [] = ...
```

Check

Feedback:

Introduce the variable pattern acc.

Calculating feedback

How do we calculate feedback?

- A strategy is specified as a context-free grammar over refinement (or rewrite) rules
- Most feedback is calculated from the grammar functions *empty* and *firsts*
- To verify that a submitted program follows a strategy we:
  - apply all allowed rules to the previous program
  - normalise the programs thus obtained
  - and compare these against the normalised program submitted by the student

[CEFP: **A strategy language**]

Strategy example

The third program for *reverse*:

$$reverse_3 = foldl\ (flip\ (:))\ []$$

is recognised by the strategy:

```
      patBind
<*> pVar "reverse"
<*> app <*> var "foldl"
        <*> (   (paren <*> app <*> var "flip"
                               <*> infixApp <*> con "(:)"
                )
          <%> con "[]"
            )
```

Introduction

- In this lecture I define the strategy combinators, together with the laws they satisfy
- We use a collection of standard combinators to combine strategies, resulting in more complex strategy descriptions.
- Our strategy language is very similar to the language for specifying context-free grammars (CFGs)
- The language is used for programming strategies, but also for strategies in other domains, such as mathematics, logic, ...

Outline of presentation

A strategy language

Running a strategy

Strategies in other domains

Limitations

A strategy language

A strategy language for what?

A strategy language for what?

A strategy language for what?

Lamp doesn't work

Lamp plugged in?

No → Plug in lamp

Yes ↓

Bulb burned out?

Yes → Replace bulb

No ↓

Buy new lamp

[CEFP: A strategy language]

A strategy language for what?

[CEFP: **A strategy language**]

Overview

Components of our strategy language:

1. Rewrite and refinement rules
2. Choice — $\sigma <|> \tau$
3. Sequence — $\sigma <\star> \tau$
4. Interleave — $\sigma <\%> \tau$
5. Unit elements — *succeed*, *fail*
6. Labels — *label* ℓ σ
7. Recursion — *fix* f

[CEFP: **A strategy language**]

The language of a strategy

- The semantics of the strategy combinators is given in terms of the language of a strategy.
- The language of a strategy is a set of sentences: a sequence of refinement or rewrite rules.
- $a, b, c, \ldots$ denote symbols, x, y, z sentences (sequences of symbols).
- ϵ is the empty sequence, xy (or ax) for concatenation.
- $\mathcal{L}$ generates the language of a strategy, by interpreting it as a context-free grammar.

[CEFP: **A strategy language**]

Rules

- $(a + b) \times c \;=\; a \times c + b \times c$
- $<?> \;\Rightarrow\; \lambda <?> \rightarrow <?>$
- LCD

The ‘alphabet’ of a strategy consists of rewrite and refinement rules.

$$\mathcal{L}\ (r) \;=\; \{r\}$$

Choice

A choice between two different model solutions

$$reverseS = reverseS_1 \mathbin{<\!|\!>} reverseS_2 \mathbin{<\!|\!>} reverseS_3$$

is modelled using the choice combinator $<\!|\!>$.

$$\mathcal{L}\ (\sigma <\!|\!> \tau) \;=\; \mathcal{L}\ (\sigma) \cup \mathcal{L}\ (\tau)$$

The *fail* combinator is a strategy that always fails.

$$\mathcal{L}\ (\mathit{fail}) \;=\; \emptyset$$

It is a unit element of $<\!|\!>$:

$$\begin{aligned} \mathit{fail} <\!|\!> \sigma &= \sigma \\ \sigma <\!|\!> \mathit{fail} &= \sigma \end{aligned}$$

Sequence

For 'first define *foldl*, and then its arguments' we use the sequence combinator $<\!\star\!>$.

$$\mathcal{L}\ (\sigma <\!\star\!> \tau) \;=\; \{xy \mid x \in \mathcal{L}\ (\sigma), y \in \mathcal{L}\ (\tau)\}$$

The *succeed* combinator is a strategy that always succeeds.

$$\mathcal{L}\ (\mathit{succeed}) \;=\; \{\epsilon\}$$

The *fail* combinator is a zero element of $<\!\star\!>$, and *succeed* is a unit element:

$$\begin{aligned} \mathit{fail} <\!\star\!> \sigma &= \mathit{fail} & \qquad \mathit{succeed} <\!\star\!> \sigma &= \sigma \\ \sigma <\!\star\!> \mathit{fail} &= \mathit{fail} & \qquad \sigma <\!\star\!> \mathit{succeed} &= \sigma \end{aligned}$$

Interleave

In a **case**-expression like

$$\begin{array}{l}\textbf{case}\ xs\ \textbf{of}\\ \quad [\,]\quad\ \ \to <?>\\ \quad x : xs \to <?>\end{array}$$

a student may refine any of the two right-hand sides, in any order.

The *interleave* combinator $<\%>$ interleaves steps of its argument strategies.

For example, $abc <\%> de$ gives:

$$\{abcde, abdce, abdec, adbce, adbec, adebc, dabce, dabec, daebc, deabc\}$$

[CEFP: **A strategy language**]

Interleaving sentences

interleave is defined in terms of an interleave operator on sentences.

As in ACP, *interleave* on sentences is defined in terms of left-interleave $x \mathbin{\%>} y$. For example, $abc \mathbin{\%>} de$ gives:

> $\{abcde, abdce, abdec, adbce, adbec, adebc\}$

$$
\begin{array}{lcll}
\epsilon \mathbin{<\%>} x & = & \{x\} & \\
x \mathbin{<\%>} \epsilon & = & \{x\} & \\
x \mathbin{<\%>} y & = & x \mathbin{\%>} y \cup y \mathbin{\%>} x & (x \not\equiv \epsilon \wedge y \not\equiv \epsilon)
\end{array}
$$

$$
\begin{array}{lcl}
\epsilon \mathbin{\%>} y & = & \emptyset \\
ax \mathbin{\%>} y & = & \{az \mid z \in x \mathbin{<\%>} y\}
\end{array}
$$

Number of interleavings

The number of interleavings for two sentences of lengths n and m equals $\frac{(n+m)!}{n!m!}$.

3	4	35
4	5	126
4	6	210
10	11	352716

Interleaving in CSP

Alternative definition of interleaving (Hoare):

$$
\begin{array}{rcl}
\epsilon \in (y <\%> z) & \Leftrightarrow & y = z = \epsilon \\
x \in (y <\%> z) & \Leftrightarrow & x \in (z <\%> y) \\
ax \in (y <\%> z) & \Leftrightarrow & (\exists\, y' : y = ay' \wedge x \in (y' <\%> z)) \\
 & \vee & (\exists\, z' : z = az' \wedge x \in (y <\%> z'))
\end{array}
$$

[CEFP: **A strategy language**]

Interleaving sets

Lifting the operations on sets:

$$\begin{array}{lcl} X <\%> Y & = & \bigcup\{x <\%> y \mid x \in X, y \in Y\} \\ X \%> Y & = & \bigcup\{x \%> y \mid x \in X, y \in Y\} \end{array}$$

For example, $\{a, ab\} <\%> \{c, cd\}$ gives:

> $\{abc, abcd, ac, acb, acbd, acd, acdb, ca, cab, cabd, cad, cadb, cda, cdab\}$

Lifted interleaving is commutative and associative, with $\{\epsilon\}$ as identity.

Left-interleave is not commutative nor associative, but does satisfy:

$$(X \%> Y) \%> Z \;=\; X \%> (Y <\%> Z)$$

[CEFP: **A strategy language**]

Atomicity

In an atomic block, no interleaving occurs with other sentences. For example, $a\langle bc\rangle <\%> \langle de\rangle f$ gives:

> $\{abcdef, adebcf, adefbc, deabcf, deafbc, defabc\}$

Atomicity satisfies:

$$\begin{aligned} \langle \epsilon \rangle &= \epsilon \\ \langle a \rangle &= a \\ \langle x\langle y\rangle z\rangle &= \langle xyz\rangle \end{aligned}$$

[CEFP: A strategy language]

The interleaving operators

The language of the interleaving operators is defined in terms of the lifted operators:

$$\begin{array}{lcl}
\mathcal{L}\ (\langle\sigma\rangle) & = & \{\langle x\rangle \mid x \in \mathcal{L}\ (\sigma)\} \\
\mathcal{L}\ (\sigma \mathbin{<\%>} \tau) & = & \mathcal{L}\ (\sigma) \mathbin{<\%>} \mathcal{L}\ (\tau) \\
\mathcal{L}\ (\sigma \mathbin{\%>} \tau) & = & \mathcal{L}\ (\sigma) \mathbin{\%>} \mathcal{L}\ (\tau)
\end{array}$$

The interleave combinator is commutative and associative, and has *succeed* as identity element:

$$\begin{array}{lcl}
\sigma \mathbin{<\%>} \tau & = & \tau \mathbin{<\%>} \sigma \\
\sigma \mathbin{<\%>} (\tau \mathbin{<\%>} u) & = & (\sigma \mathbin{<\%>} \tau) \mathbin{<\%>} u \\
\sigma \mathbin{<\%>} \mathit{succeed} & = & \sigma
\end{array}$$

Labels

When developing a program, a student may ask for a hint at any time.

We mark positions in the strategy with a label, which allows us to attach feedback to a strategy.

$$\mathcal{L}\ (\mathit{label}\ \ell\ \sigma) \ = \ \{\mathsf{Enter}_\ell\ x\ \mathsf{Exit}_\ell \mid x \in \mathcal{L}\ (\sigma)\}$$

[CEFP: **A strategy language**]

Recursion

Recursion is used for example to specify that a user replaces all occurrences of a particular expression in a program by another expression.

$$\mathcal{L}\ (\mathit{fix}\ f) \ = \ \mathcal{L}\ (f\ (\mathit{fix}\ f))$$

Overview: a grammar for strategies

A strategy is an element of the language of the following grammar:

$$
\begin{array}{rll}
\sigma ::= & r & \\
\mid & \sigma <|> \sigma & \mid \quad \mathit{fail} \\
\mid & \sigma <\star> \sigma & \mid \quad \mathit{succeed} \\
\mid & \mathit{label}\ \ell\ \sigma & \\
\mid & \mathit{fix}\ f & \\
\mid & \langle \sigma \rangle \quad \mid \quad \sigma <\%> \sigma & \mid \quad \sigma\ \%>\ \sigma
\end{array}
$$

where r is a rewrite rule or a refinement rule, ℓ is a label, and f is a function that takes a strategy as argument, and returns a strategy.

[CEFP: **A strategy language**]

Overview: the language of a strategy

The language of a strategy is defined by:

$$
\begin{array}{lcl}
\mathcal{L}\ (r) & = & \{r\} \\
\mathcal{L}\ (\sigma <\!|\!> \tau) & = & \mathcal{L}\ (\sigma) \cup \mathcal{L}\ (\tau) \\
\mathcal{L}\ (\mathit{fail}) & = & \emptyset \\
\mathcal{L}\ (\sigma <\!\star\!> \tau) & = & \{xy \mid x \in \mathcal{L}\ (\sigma), y \in \mathcal{L}\ (\tau)\} \\
\mathcal{L}\ (\mathit{succeed}) & = & \{\epsilon\} \\
\mathcal{L}\ (\mathit{label}\ \ell\ \sigma) & = & \{\mathsf{Enter}_\ell\ x\ \mathsf{Exit}_\ell \mid x \in \mathcal{L}\ (\sigma)\} \\
\mathcal{L}\ (\mathit{fix}\ f) & = & \mathcal{L}\ (f\ (\mathit{fix}\ f)) \\
\mathcal{L}\ (\langle\sigma\rangle) & = & \{\langle x\rangle \mid x \in \mathcal{L}\ (\sigma)\} \\
\mathcal{L}\ (\sigma_1 <\!\%\!> \sigma_2) & = & \mathcal{L}\ (\sigma_1) <\!\%\!> \mathcal{L}\ (\sigma_2) \\
\mathcal{L}\ (\sigma_1 \%\!> \sigma_2) & = & \mathcal{L}\ (\sigma_1) \%\!> \mathcal{L}\ (\sigma_2)
\end{array}
$$

[CEFP: **A strategy language**]

Running a strategy

Running a strategy

A sequence of rules follows a strategy iff the sequence of rules is (a prefix of) a sentence in the language generated by the strategy.

An exercise gives us an initial term (say t_0), and we are only interested in sequences of rules that can be applied successively to this term.

A possible derivation that starts with t_0 can be depicted in the following way:

$$t_0 \xrightarrow{r_0} t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} t_3 \xrightarrow{r_3} \ldots$$

Running a strategy is defined by:

$$\mathit{run}\ \sigma\ t_0 = \{ t_{n+1} \mid r_0 \ldots r_n \in \mathcal{L}\ (\sigma), \forall_{i \in 0 \ldots n} : t_{i+1} \in \mathit{apply}\ r_i\ t_i \}$$

[CEFP: **A strategy language**]

A strategy recogniser

Determining whether or not a sequence of rules is an element of the language of a strategy:

$$run\ \sigma\ t_0 = \{ \ldots \mid r_0 \ldots r_n \in \mathcal{L}\ (\sigma), \ldots \}$$

might be very expensive.

In the next lecture we describe an efficient strategy recogniser.

Strategies in other domains

Examples of strategies in other domains

Strategies can be used for all domains in which procedural skills are expressed in terms of rewriting and refinement rules.

- programming
- logic
- mathematics
- physics
- ...

Example: *many*

Repetition, zero or more occurrences of something, is a well-known recursion pattern.

$$many\ \sigma = fix\ (\lambda x \rightarrow succeed \mathbin{<\!|\!>} (\sigma \mathbin{<\!\star\!>} x))$$

The strategy that applies transformation rule r zero or more times would thus be:

$$
\begin{aligned}
& many\ r \\
& \quad = succeed \mathbin{<\!|\!>} (r \mathbin{<\!\star\!>} many\ r) \\
& \quad = succeed \mathbin{<\!|\!>} (r \mathbin{<\!\star\!>} (succeed \mathbin{<\!|\!>} (r \mathbin{<\!\star\!>} many\ r))) \\
& \quad = \ldots
\end{aligned}
$$

[CEFP: **A strategy language**]

Example: simplifying fractions

Add two fractions, for example, $\frac{2}{5}$ and $\frac{2}{3}$.

Improper fractions (numerator is larger than or equal to denominator) are converted to mixed numbers.

$$
\begin{array}{ll}
 & \dfrac{2}{5} + \dfrac{2}{3} \\
\Rightarrow & \{ \quad \text{Rename a denominator } \} \\
 & \dfrac{6}{15} + \dfrac{2}{3} \\
\Rightarrow & \{ \quad \text{Rename a denominator } \} \\
 & \dfrac{6}{15} + \dfrac{10}{15} \\
\Rightarrow & \{ \quad \text{Add } \} \\
 & \dfrac{16}{15} \\
\Rightarrow & \{ \quad \text{Simplify } \} \\
 & 1\dfrac{1}{15}
\end{array}
$$

[CEFP: **A strategy language**]

Rules for fractions

$$\text{Add:} \quad \frac{a}{c} + \frac{b}{c} = \frac{a+b}{c}$$

$$\text{Mul:} \quad \frac{a}{b} \times \frac{c}{d} = \frac{a \times c}{b \times d}$$

$$\text{Rename:} \quad \frac{b}{c} = \frac{a \times b}{a \times c}$$

$$\text{Simpl:} \quad \frac{a+b}{b} = 1 + \frac{a}{b}$$

Buggy rules for fractions

B1: $\dfrac{a}{b} + \dfrac{c}{d} \quad \neq \quad \dfrac{a+c}{b+d}$

B2: $a \times \dfrac{b}{c} \quad \neq \quad \dfrac{a \times b}{a \times c}$

B3: $a + \dfrac{b}{c} \quad \neq \quad \dfrac{a+b}{c}$

A strategy for simplifying fractions

- *Step 1.* Find the least common denominator (LCD) of the fractions: let this be n
- *Step 2.* Rename the fractions such that n is the denominator
- *Step 3.* Add the fractions by adding the numerators
- *Step 4.* Simplify the fraction if it is improper

Strategy:

$$
\begin{array}{l}
\mathit{addFractions} = \textit{label } \ell_0 \\
\quad (\quad \textit{label } \ell_1 \text{ LCD} \\
\quad \mathbin{<\star>} \textit{label } \ell_2 \ (\mathit{repeat}\ (\mathit{somewhere}\ \text{Rename})) \\
\quad \mathbin{<\star>} \textit{label } \ell_3 \text{ Add} \\
\quad \mathbin{<\star>} \textit{label } \ell_4 \ (\mathit{try}\ \text{Simpl}) \\
\quad)
\end{array}
$$

Minor rules

- Some rules don't change the term, but change the focus, perform administrative tasks, or store information in the environment
- Such rules are called minor rules
- Minor rules are applied silently

Term location

- *somewhere* changes the focus in an abstract syntax tree by means of minor navigation rules, and then applies its argument rule
- Navigation by means of Down, Left, Right, and Up
- Inspired by the operations on the zipper data structure
- The programming domain uses holes for locations in terms
- Using navigation rules and the zipper in the functional programming domain is less convenient at the moment:

```
...      foldlS
    <⋆> Down
    <⋆> (flipS consS)
    <⋆> Right
    <⋆> nilS
    <⋆> Up
```

Exercise: from λ to SKI

Develop a strategy for rewriting a λ-term into *SKI*-combinators. For example,

$$\lambda x \to \lambda y \to x\ y$$

can be transformed into

$$S\ (S\ (K\ S)\ (S\ (K\ K)\ I))\ (K\ I)$$

where

$$\begin{aligned} S &= \lambda x\ y\ z \to x\ z\ (y\ z) \\ K &= \lambda x\ y \to x \\ I &= \lambda x \to x \end{aligned}$$

See `Feedback/trunk/src/Domain/Lambda.hs`.

Exercise: pointfree – λ's without λ

Develop both rules and a strategy, and probably adapt the λ-datatype to rewrite a λ-term into pointfree form.

$\lambda x \rightarrow \lambda y \rightarrow x \mathbin{:@:} y$

$\Rightarrow$ { Write infix operation as prefix function }

$\lambda x \rightarrow \lambda y \rightarrow (:@:) \mathbin{:@:} x \mathbin{:@:} y$

$\Rightarrow$ { Remove an abstraction: $\lambda y \rightarrow f \mathbin{:@:} y \Rightarrow (f \mathbin{:@:})$ }

$\lambda x \rightarrow ((:@:) \mathbin{:@:} x \mathbin{:@:})$

$\Rightarrow$ { Write sectioned operation (last occurrence of :@:) as a prefix function }

$\lambda x \rightarrow (:@:) \mathbin{:@:} ((:@:) \mathbin{:@:} x)$

$\Rightarrow$ { Introduce composition }

$\lambda x \rightarrow ((:@:) \circ ((:@:) \mathbin{:@:})) \mathbin{:@:} x$

$\Rightarrow$ { Remove an abstraction }

$(((:@:) \circ ((:@:) \mathbin{:@:})) \mathbin{:@:})$

[CEFP: **A strategy language**]

Limitations

Left-recursion

The following strategy is left-recursive:

$$leftRecursive = fix\ (\lambda x \rightarrow x <\star> \mathsf{Add})$$

Strategies with leading minor rules may or may not be left-recursive. For example, if we use a minor rule that increases a counter in the environment, which is an action that always succeeds, the strategy is left-recursive. But

$$leftRecursive' = fix\ (\lambda x \rightarrow \mathsf{Down} <\star> x <\star> \mathsf{Add})$$

is not left-recursive.

A strategy may not be left-recursive

- We use top-down recursive parsing to track student behaviour and give feedback, because we want to support the top-down, incremental construction of derivations
- Top-down recursive parsing using a left-recursive context-free grammar is difficult
- For a strategy to be used in our framework, it should not be left-recursive
- Using bottom-up parsers would introduce other problems
- Our strategy parser has a 'time-out' to avoid non-termination caused by recursion

A strategy needs to be LL(1)

We have to choose which (choice-)branch in a strategy to take based on the next input symbol.

Consider the following, somewhat contrived, strategy:

$$\begin{aligned} \mathit{leftFactor} \;&=\; \textsf{\textit{label}}\ \ell_1\ (\textsf{Add} <\!\star\!> \textsf{Simpl}) \\ &<\!|\!>\ \textsf{\textit{label}}\ \ell_2\ (\textsf{Add} <\!\star\!> \textsf{Rename}) \end{aligned}$$

We do not know which branch to choose when we encounter an Add.

A strategy needs to be LL(1).

Left-factoring

Left-factoring a strategy can help in making a strategy LL(1), and hence in not committing early to a particular sub-strategy.

$$\mathit{leftFactor}' = \text{Add} \mathbin{<\!\star\!>} (\text{Simpl} \mathbin{<\!|\!>} \text{Rename})$$

How do we left-factor labels, or minor rules in general? Pushing labels inwards:

$$\mathit{leftFactor}'' = \text{Add} \mathbin{<\!\star\!>} (\mathit{label}\ \ell_1\ \text{Simpl} \mathbin{<\!|\!>} \mathit{label}\ \ell_2\ \text{Rename})$$

breaks the relation between the label and the strategy.

Automatic left-factoring seems hard.

A strategy must be left-factored

- At the moment we require strategies to be left-factored
- This is very undesirable, since it makes it hard to generate functional programming strategies from model solutions
- We intend to use some form of parallel parsing to solve this problem

Conclusions

- Our strategy language is very similar to the language of context-free grammars
- We check student actions by running/parsing against a strategy
- A strategy may not be left-recursive and should be left-factored