Pattern Matching with Sequence Variables

L. Thomas van Binsbergen

Royal Holloway, University of London

11 May, 2016

In your favourite programming language

Can a function, method or procedure have multiple arguments and multiple return values?

In your favourite programming language

Can a function, method or procedure have multiple arguments and multiple return values?

- Procedural programmer: Off course, we use in/out parameters.

In your favourite programming language

Can a function, method or procedure have multiple arguments and multiple return values?

- Procedural programmer: Off course, we use in/out parameters.
- OOer: Off course, we objects with multiple fields.

In your favourite programming language

Can a function, method or procedure have multiple arguments and multiple return values?

- Procedural programmer: Off course, we use in/out parameters.
- OOer: Off course, we objects with multiple fields.
- FPer: No way, a function has a single return value.

In your favourite programming language

Can a function, method or procedure have multiple arguments and multiple return values?

- Procedural programmer: Off course, we use in/out parameters.
- OOer: Off course, we objects with multiple fields.
- FPer: No way, a function has a single return value.
- Haskell Curry: No way, every function has 1 argument and 1 result.

In your favourite programming language

Can a function, method or procedure have **an arbitrary, unfixed** *number of arguments?*

In your favourite programming language

Can a function, method or procedure have **an arbitrary, unfixed** *number of arguments?*

- Procedural programmer: Off course, by passing an array.

In your favourite programming language

Can a function, method or procedure have **an arbitrary, unfixed** *number of arguments?*

- Procedural programmer: Off course, by passing an array.
- Java'er: Off course, by using `varargs`.

In your favourite programming language

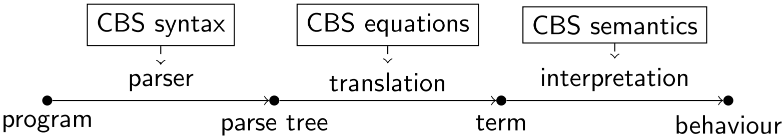
Can a function, method or procedure have **an arbitrary, unfixed** *number of arguments?*

- Procedural programmer: Off course, by passing an array.
- Java'er: Off course, by using `varargs`.
- FPer: No way! How to infer types? How to write patterns?

Java Example

```
public static int sum (int... numbers) {
    int sum = 0;
    for (int i = 0; i < numbers.length; i++) {
        sum = sum + numbers [i];
    }
    return sum;
}
```

The CBS Language - Executable Formal Specification



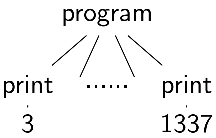
CBS Example

$$\begin{array}{lll} \textit{program} & ::= & (\textit{stmt}\ \text{';'}\ )^* \\ \textit{stmt} & ::= & \textit{print}\ \textbf{int} \mid \ldots \end{array}$$

print 3;

...

print 1337;



$\boxed{\textbf{left-to-right}(\textbf{print}(3),\ldots,\textbf{print}(1337))}$

Pattern Matching

- Split a function by a *case-analysis* on its arguments.

$$
\begin{aligned}
&\mathit{sum} : [\mathbf{int}] \to \mathbf{int} \\
&\mathit{sum}\,([\,]) \qquad\;\; = 0 \\
&\mathit{sum}\,(x :: xs) = \mathit{plus}\,(x, \mathit{sum}\,(xs))
\end{aligned}
$$

Patterns

- A pattern is either:
  - A wildcard _
  - A variable, e.g. X
  - Or an applications of a *term constructor* to patterns
- Examples: X, **true**, **list**$(X, \_)$, **tuple**$(\_, 1, X)$
- A pattern can *match* a term, producing *bindings*

Basic Pattern Matching Algorithm

- Any term is matched by _
- Any term T is matched by X, resulting in $\{X \mapsto T\}$
- Term $f(T_1, \ldots, T_n)$ is matched by pattern $g(P_1, \ldots, P_m)$, iff:
  - $f \equiv g$
  - $n \equiv m$
  - $\forall 1 \leqslant i \leqslant n$, T_i is matched by P_i (bindings are united)
- Failure otherwise

Sequences

Sequences

- A *sequence* denotes zero, one or more terms:
  - $1, 2, \mathit{true}$
  - **sum**$(1, 2, 3)$

Sequence Variables

- A sequence variable X^*, X^+ or $X^?$ is bound to a sequence.
- Sequences implictly merge (no nesting).
- Given $X^* \mapsto 2, \mathit{true}$: $[1, X^*] \equiv [1, 2, \mathit{true}]$

Pattern Matching with Sequence Variables

The proposal

A pattern is either:

- A wildcard _
- A variable, e.g. X
- A sequence variable, e.g. X^*, X^+ or $X^?$
- A sequence variable with a predicate, e.g. $X^* : p$
- Or formed by applications of term constructors to patterns

Example - sum

$$\begin{aligned}
&\mathbf{sum} : \mathbf{int}^* \to \mathbf{int} \\
&\mathbf{sum}\,() &&= 0 \\
&\mathbf{sum}\,(X) &&= X \\
&\mathbf{sum}\,(X, Y) &&= \mathbf{plus}\,(X, Y) \\
&\mathbf{sum}\,(X, Y, Z^+) &&= \mathbf{plus}\,(\mathbf{plus}\,(X, Y), \mathbf{sum}\,(Z^+))
\end{aligned}$$

Example - list-map

$$
\begin{aligned}
&\textbf{list-map} : (V \rightarrow R) \ \times \ [V] \rightarrow [R] \\
&\textbf{list-map}\ (F, [\,]) \qquad\quad = [\,] \\
&\textbf{list-map}\ (F, [V, V^*]) = [F\ (V), \textbf{list-map}\ (F, [V^*])]
\end{aligned}
$$

Example - myswap

$$\text{Goal}: \textbf{myswap}\,(\texttt{'a'}, \texttt{'b'}, \ldots, 1, 2, \ldots) \equiv (1, 2, \ldots, \texttt{'a'}, \texttt{'b'}, \ldots)$$

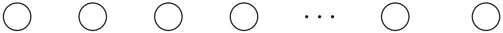
$$\textbf{myswap}\,(X^{+} : \textbf{char}, Y^{+} : \textbf{int}) = (Y^{+}, X^{+})$$

Ambiguity

- Example of an ambiguous pattern: $X^* : \textbf{int}, Y^+$
- Can match $1,$ *true* in two ways.
- Greedy longest-match is no solution, e.g. $1, 2, 3$

Overview of the Algorithm

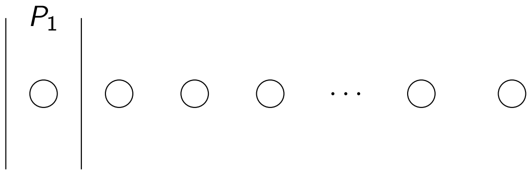
$T_1 \ldots T_n$



$P_1 \ldots P_m$

Overview of the Algorithm

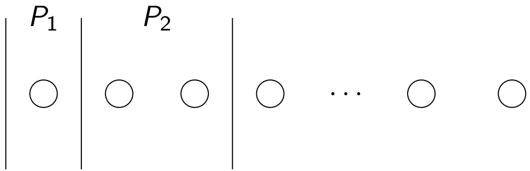
$T_1 \ldots T_n$



$P_1 \ldots P_m$

Overview of the Algorithm

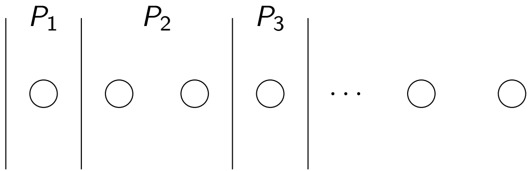
$T_1 \ldots T_n$



$P_1 \ldots P_m$

Overview of the Algorithm

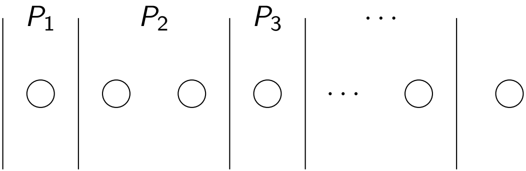
$T_1 \dots T_n$



$P_1 \dots P_m$

Overview of the Algorithm

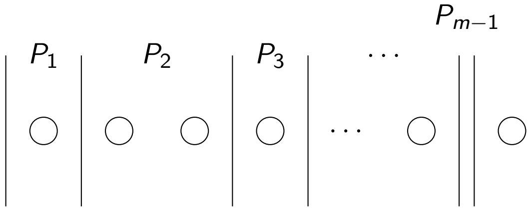
$T_1 \ldots T_n$



$P_1 \ldots P_m$

Overview of the Algorithm

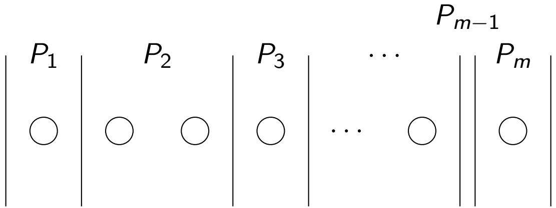
$T_1 \ldots T_n$



$P_1 \ldots P_m$

Overview of the Algorithm

$T_1 \ldots T_n$



$P_1 \ldots P_m$

Conclusions

Summary

- Motivated the usage of sequence variables.
- An algorithm for a more powerful matching scheme.
- CBS suggests sequences variables are convenient.

Open Questions

- Is the scheme useful in other languages and domains?
- Can it be integrated into a language with type inferencing?
- What is the worst-case complexity of the algorithm?

Pattern Matching Algorithm M^*

- Assume matcher M for single terms and patterns
- We match $T_1, \ldots, T_n$ with $P_1, \ldots, P_m$
- Initially $\{(0, 0, \emptyset)\} \equiv \mathcal{R}$
- Pop $(i, j, env) \in \mathcal{R}$, until $\mathcal{R} \equiv \emptyset$:
  1. If $i \equiv n + 1$ and $j \equiv m + 1$, **return** *env*
  2. If $i < n + 1$ and $j \equiv m + 1$, **continue**
  3. If P_j is a simple pattern:
     - Add $(i + 1, j + 1, env') \in \mathcal{R}$ iff $env' = M(T_i, P_j, env)$
  4. If P_j is a sequence variable: ... (next slide)
- Failure if no *env* was returned

Pattern Matching Algorithm (2)

- If P_j a sequence variable, add $(k, j+1, env'(k)) \in \mathcal{R}$, with:
  - If P_j is X^* then $\forall i \leqslant k \leqslant n+1$
  - If P_j is X^+ then $\forall i+1 \leqslant k \leqslant n+1$
  - If P_j is $X^?$ then $\forall i \leqslant k \leqslant i+2$
  - $env'(k) = [P_j \mapsto T_i, \ldots, T_{k-1}]env$
- If P_j has predicate p, then k is the maximum number s.t. $T_i, \ldots, T_{k-1}$ satisfy p

Simple Pattern Matching Extended

- ...
- Term $f(T_1, \ldots, T_n)$ is matched by pattern $g(P_1, \ldots, P_m)$, iff:
  - $f \equiv g$.
    - $T_1, \ldots, T_n$ is matched by $P_1, \ldots, P_m$, according to M^*
    - Otherwise as before...
- ...