Declarative Programming with Sequence and Context Variables

Besik Dundua

Universidade de Brasília

Joint work with Mário Florido and Temur Kutsia

Outline

Different Kinds of Variables

Constraint Logic Programming

Rule-Based Programming

Functional Programming

Outline

Different Kinds of Variables

Constraint Logic Programming

Rule-Based Programming

Functional Programming

Different Kinds of Variables

- Sequence (aka hedge) variables stand for finite sequences of terms.

Different Kinds of Variables

- Sequence (aka hedge) variables stand for finite sequences of terms.
- Context variables denote contexts that can be seen as unary functions with a single occurrence of the bound variable.

Different Kinds of Variables

- Sequence (aka hedge) variables stand for finite sequences of terms.
- Context variables denote contexts that can be seen as unary functions with a single occurrence of the bound variable.
- Sequence and context variables give the user flexibility on selecting subsequences in sequences or subterms/contexts in terms.

Different Kinds of Variables

- Sequence (aka hedge) variables stand for finite sequences of terms.
- Context variables denote contexts that can be seen as unary functions with a single occurrence of the bound variable.
- Sequence and context variables give the user flexibility on selecting subsequences in sequences or subterms/contexts in terms.
- Sequence and context variables enhance expressive capabilities of a language, help to write short, neat, understandable code, and hide away many tedious data processing details from the programmer.

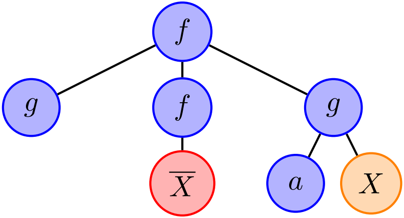
Different Kinds of Variables

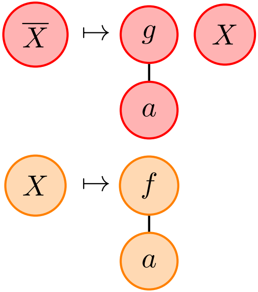
- Sequence (aka hedge) variables stand for finite sequences of terms.
- Context variables denote contexts that can be seen as unary functions with a single occurrence of the bound variable.
- Sequence and context variables give the user flexibility on selecting subsequences in sequences or subterms/contexts in terms.
- Sequence and context variables enhance expressive capabilities of a language, help to write short, neat, understandable code, and hide away many tedious data processing details from the programmer.
- We have also variables that stand for individual terms, and variables that stand for function symbols.

Intuition Behind Individual (X) and Sequence Variables ($\overline{X}$)

Example

$f(g, f(\overline{X}), g(a, X))$ $\{\overline{X} \mapsto (g(a), X),\ X \mapsto f(a)\}$

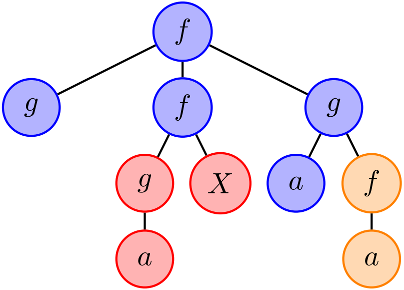


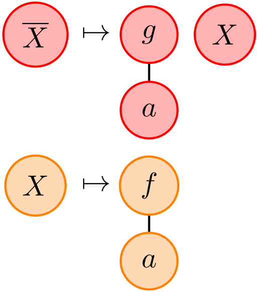


Intuition Behind Individual (X) and Sequence Variables ($\overline{X}$)

Example

$f(g, f(g(a), y), g(a, f(a)))$ $\{\overline{X} \mapsto (g(a), X),\ X \mapsto f(a)\}$

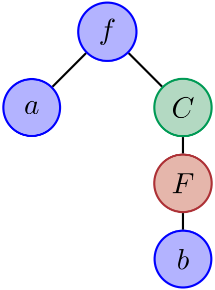


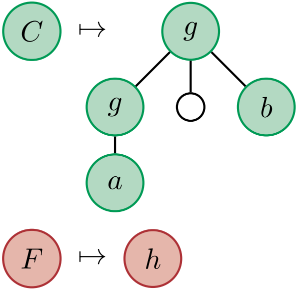


Intuition Behind Function (F) and Context Variables (C)

Example

$f(a, C(F(b)))$ $\{C \mapsto g(g(a), \circ, b),\ F \mapsto h\}$

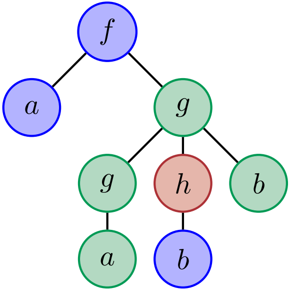


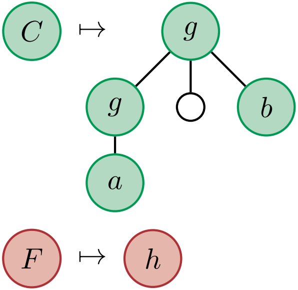


Intuition Behind Function (F) and Context Variables (C)

Example

$f(a, g(g(a), h(b), b))$ $\{C \mapsto g(g(a), \circ, b),\ F \mapsto h\}$





Programming with Sequence and Context Variables

We studied extensions with sequence and context variables of the formalisms for

- constraint logic programming,
- rule-based programming, and
- functional programming

Outline

CLP(SC)

- Constraint logic programming is one of the most successful areas of logic programming, combining logical deduction with constraint solving.

CLP(SC)

- Constraint logic programming is one of the most successful areas of logic programming, combining logical deduction with constraint solving.
- The main technique used in constraint logic programming research is introducing a new constraint domain, designing an efficient satisfiability and solving procedure for it, and putting it in the general constraint logic programming framework.

CLP(SC)

- Constraint logic programming is one of the most successful areas of logic programming, combining logical deduction with constraint solving.
- The main technique used in constraint logic programming research is introducing a new constraint domain, designing an efficient satisfiability and solving procedure for it, and putting it in the general constraint logic programming framework.
- The domain we studied is the domain of sequences and contexts. Constraint logic programming over this domain is denoted by CLP(SC).

CLP(SC): Rewriting Example

- A program that implements the rewriting mechanism, together with a rule to perform rewritings of the form $f \to f(b,b)$, $f(a) \to f(b,a,b)$, $f(a,a) \to f(b,a,a,b)$, etc.

$$rewrite(C(X), C(Y)) \leftarrow rule(X, Y).$$
$$rule(F(\overline{X}), F(b, \overline{X}, b)) \leftarrow \overline{X} \text{ in } a^*.$$

CLP(SC): Rewriting Example

- A program that implements the rewriting mechanism, together with a rule to perform rewritings of the form $f \to f(b,b)$, $f(a) \to f(b,a,b)$, $f(a,a) \to f(b,a,a,b)$, etc.

$$rewrite(C(X), C(Y)) \leftarrow rule(X, Y).$$
$$rule(F(\overline{X}), F(b, \overline{X}, b)) \leftarrow \overline{X} \text{ in } a^*.$$

- Goal: Find a term that rewrites to $f(a, f(b, f(b,a,a,b)))$:

$$\leftarrow rewrite(X, f(f(b,a,b), f(b, f(b,a,a,b)))).$$

CLP(SC): Rewriting Example

- A program that implements the rewriting mechanism, together with a rule to perform rewritings of the form $f \to f(b,b)$, $f(a) \to f(b,a,b)$, $f(a,a) \to f(b,a,a,b)$, etc.

$$rewrite(C(X), C(Y)) \leftarrow rule(X, Y).$$
$$rule(F(\overline{X}), F(b, \overline{X}, b)) \leftarrow \overline{X} \text{ in } a^*.$$

- Goal: Find a term that rewrites to $f(a, f(b, f(b,a,a,b)))$:

$$\leftarrow rewrite(X, f(f(b,a,b), f(b, f(b,a,a,b)))).$$

- Two answers:

$$X = f(f(a), f(b, f(b,a,a,b))),$$
$$X = f(f(b,a,b), f(b, f(a,a))).$$

Constraint Solving

- CLP(SC) relies on solving equational and membership constraints over the domain of sequences and contexts.
- We designed a constraint solving algorithm for this domain.
- We proved that the algorithm is sound, terminating, and incomplete.
- We identified fragments of constraints that can be completely solved by the algorithm.

CLP(SC)

- CLP(SC) is obtained from the CLP schema by instantiating the domain with sequences and contexts, and using the constraint solving algorithm that we developed.
- We studied declarative and operational semantics of CLP(SC).
- We investigated restrictions on programs leading to constraints in a special form for which the constraint solving algorithm is complete.

Outline

Rule-Based Programming in PρLog

- PρLog is a rule based system that supports programming with individual, sequence, function and context variables.
- It extends logic programming with strategic conditional transformation rules where sequence and context variables can be restricted by regular expressions.
- Rules perform nondeterministic transformations of sequences.
- Strategies provide a mechanism to control computation.
- PρLog is implemented in Prolog and uses its inference mechanism.
- Unification is replaced with matching for unranked terms and four kinds of variables.

Example: Remove Duplicates

- Remove a repeated element from a sequence:

  $remove\_duplicates :: (\overline{X}, X, \overline{Y}, X, \overline{Z}) \Longrightarrow (\overline{X}, X, \overline{Y}, \overline{Z}).$

Example: Remove Duplicates

- Remove a repeated element from a sequence:

  $remove\_duplicates :: (\overline{X}, X, \overline{Y}, X, \overline{Z}) \Longrightarrow (\overline{X}, X, \overline{Y}, \overline{Z}).$

- Query:

  $remove\_duplicates :: (a, f(a), f(a), a) \Longrightarrow \overline{Result}.$

Example: Remove Duplicates

- Remove a repeated element from a sequence:

$$remove\_duplicates :: (\overline{X}, X, \overline{Y}, X, \overline{Z}) \Longrightarrow (\overline{X}, X, \overline{Y}, \overline{Z}).$$

- Query:

$$remove\_duplicates :: (a, f(a), f(a), a) \Longrightarrow \overline{Result}.$$

- Two answers, computed via backtracking:

$$Result = (a, f(a), f(a)),$$
$$Result = (a, f(a), a).$$

Example: Remove Duplicates

- Goal: Remove all repeated elements from a sequence.
- Idea: Compute a normal form with respect to $remove\_duplicates$.

Example: Remove Duplicates

- Goal: Remove all repeated elements from a sequence.
- Idea: Compute a normal form with respect to $remove\_duplicates$.
- Query:

$$nf(remove\_duplicates) :: (a, f(a), f(a), a) \Longrightarrow \overline{Result}.$$

nf: PρLog's strategy for computing normal forms.

Example: Remove Duplicates

- Goal: Remove all repeated elements from a sequence.
- Idea: Compute a normal form with respect to $remove\_duplicates$.
- Query:

  $$nf(remove\_duplicates) :: (a, f(a), f(a), a) \Longrightarrow \overline{Result}.$$

  nf: PρLog's strategy for computing normal forms.
- Result: $\overline{Result} = (a, f(a))$.

Example: Flattening

- A program to remove from a term a nested occurrence of the function symbol F:

$$\mathit{flatten}(F) :: C(F(\overline{X}, F(\overline{Y}), \overline{Z})) \Longrightarrow C(F(\overline{X}, \overline{Y}, \overline{Z})).$$

Example: Flattening

- A program to remove from a term a nested occurrence of the function symbol F:

$$\mathit{flatten}(F) :: C(F(\overline{X}, F(\overline{Y}), \overline{Z})) \Longrightarrow C(F(\overline{X}, \overline{Y}, \overline{Z})).$$

- Remove a nested occurrence of f. Query:

$$\mathit{flatten}(f) :: g(f(a, f(b, f(c, d))), g(e)) \Longrightarrow \mathit{Result}.$$

Example: Flattening

- A program to remove from a term a nested occurrence of the function symbol F:

$$\mathit{flatten}(F) :: C(F(\overline{X}, F(\overline{Y}), \overline{Z})) \Longrightarrow C(F(\overline{X}, \overline{Y}, \overline{Z})).$$

- Remove a nested occurrence of f. Query:

$$\mathit{flatten}(f) :: g(f(a, f(b, f(c, d))), g(e)) \Longrightarrow \mathit{Result}.$$

- Two answers, computed via backtracking:

$$\mathit{Result} = g(f(a, b, f(c, d)), g(e)),$$
$$\mathit{Result} = g(f(a, f(b, c, d)), g(c)).$$

Example: Flattening

- A program to remove form a term a nested occurrence of the function symbol F:

$$\mathit{flatten}(F) :: C(F(\overline{X}, F(\overline{Y}), \overline{Z})) \Longrightarrow C(F(\overline{X}, \overline{Y}, \overline{Z})).$$

- Remove a nested occurrence of g. Query:

$$\mathit{flatten}(g) :: g(f(a, f(b, f(c, d))), g(e)) \Longrightarrow \mathit{Result}.$$

Example: Flattening

- A program to remove form a term a nested occurrence of the function symbol F:

$$\mathit{flatten}(F) :: C(F(\overline{X}, F(\overline{Y}), \overline{Z})) \Longrightarrow C(F(\overline{X}, \overline{Y}, \overline{Z})).$$

- Remove a nested occurrence of g. Query:

$$\mathit{flatten}(g) :: g(f(a, f(b, f(c, d))), g(e)) \Longrightarrow \mathit{Result}.$$

- One answer:

$$\mathit{Result} = g(f(a, b, f(c, d)), e).$$

Constructing Complex Strategies

Complex strategies can be constructed from simpler ones by strategy combinators.

Constructing Complex Strategies

Complex strategies can be constructed from simpler ones by strategy combinators.

Example

- The strategy definition

  $$\mathit{flatten\_all\_and\_remove\_all\_duplicates}(F) := \\ \quad \mathit{compose}(\mathit{map}_1(\mathit{nf}(\mathit{flatten}(F))), \mathit{nf}(\mathit{remove\_duplicates})).$$

  defines a strategy that composes two strategies: $\mathit{map}_1(\mathit{nf}(\mathit{flatten}(F)))$ and $\mathit{nf}(\mathit{remove\_duplicates})$.

Constructing Complex Strategies

Complex strategies can be constructed from simpler ones by strategy combinators.

Example

- The strategy definition

  $$\mathit{flatten\_all\_and\_remove\_all\_duplicates}(F) := \mathit{compose}(\mathit{map}_1(\mathit{nf}(\mathit{flatten}(F))), \mathit{nf}(\mathit{remove\_duplicates})).$$

  defines a strategy that composes two strategies: $\mathit{map}_1(\mathit{nf}(\mathit{flatten}(F)))$ and $\mathit{nf}(\mathit{remove\_duplicates})$.
- $\mathit{map}_1(\mathit{nf}(\mathit{flatten}(F)))$ applies the strategy $\mathit{nf}(\mathit{flatten}(F))$ to each element of the input sequence.

Constructing Complex Strategies

Complex strategies can be constructed from simpler ones by strategy combinators.

Example

- The strategy definition

  $$\begin{aligned}&\mathit{flatten\_all\_and\_remove\_all\_duplicates}(F) := \\ &\quad \mathit{compose}(\mathit{map}_1(\mathit{nf}(\mathit{flatten}(F))), \mathit{nf}(\mathit{remove\_duplicates})).\end{aligned}$$

  defines a strategy that composes two strategies: $\mathit{map}_1(\mathit{nf}(\mathit{flatten}(F)))$ and $\mathit{nf}(\mathit{remove\_duplicates})$.
- $\mathit{map}_1(\mathit{nf}(\mathit{flatten}(F)))$ applies the strategy $\mathit{nf}(\mathit{flatten}(F))$ to each element of the input sequence.
- The result sequence is then processed by the strategy $\mathit{nf}(\mathit{remove\_duplicates})$ to remove all duplicates.

Example: Flatten All and Remove All Duplicates

- Flatten all occurrences of f from the input sequence $(g(a), f(a, f(b)), g(g(a)), f(f(a, b)))$ and remove all duplicates from the obtained sequence.

Example: Flatten All and Remove All Duplicates

- Flatten all occurrences of f from the input sequence $(g(a), f(a, f(b)), g(g(a)), f(f(a, b)))$ and remove all duplicates from the obtained sequence.
- Query:

$$\mathit{flatten\_all\_and\_remove\_all\_duplicates}(f) :: (g(a), f(a, f(b)), g(g(a)), f(f(a, b))) \Longrightarrow \overline{Result}.$$

Example: Flatten All and Remove All Duplicates

- Flatten all occurrences of f from the input sequence $(g(a), f(a, f(b)), g(g(a)), f(f(a, b)))$ and remove all duplicates from the obtained sequence.
- Query:

$$\mathit{flatten\_all\_and\_remove\_all\_duplicates}(f) ::$$
$$(g(a), f(a, f(b)), g(g(a)), f(f(a, b))) \Longrightarrow \overline{Result}.$$

- Answer: $\overline{Result} = (g(a), f(a, b), g(g(a)))$.

Applications of PρLog

We have applications of PρLog in

- XML processing,
- Web reasoning, and
- implementing rewriting strategies.

PρLog can be downloaded from

http://www.risc.jku.at/people/tkutsia/software.html

Outline

Pattern-Based Calculi

- Functional programming has its roots in the lambda calculus.
- Pattern calculi generalize the lambda calculus.
- The main idea behind the generalization:
  - Integrate pattern matching into the lambda calculus.
  - Abstraction on arbitrary terms (patterns), not only on variables.
- "A small typed pattern calculus supports all the main programming styles."

  B. Jay. *The Pattern Calculus.* Springer, 2009.

Pattern-Based Calculi

- Functional programming has its roots in the lambda calculus.
- Pattern calculi generalize the lambda calculus.
- The main idea behind the generalization:
  - Integrate pattern matching into the lambda calculus.
  - Abstraction on arbitrary terms (patterns), not only on variables.
- "A small typed pattern calculus supports all the main programming styles."

  B. Jay. *The Pattern Calculus.* Springer, 2009.

Example

$\lambda f(x).\, g(x)$ is a well-formed expression in the lambda calculus with patterns.

Pattern-Based Calculi

β-reduction idea:

$$(\lambda P.M)Q \to M\sigma, \text{ where } \sigma \text{ is a matcher of } P \text{ to } Q.$$

Various Pattern Calculi

- Lambda calculus with patterns (van Oostrom, 1990, Klop et al, 2008).
- ρ-calculus (Cirstea and Kirchner, 2000).
- Lambda (eta) calculus with a case construct (Arbiser et al, 2009).
- Pure pattern calculus (Jay and Kesner, 2006, 2009).
- ...

Properties of Pattern Calculi

- Patterns themselves can be reduced and instantiated.
- It makes pattern calculi expressive, but there is a price to pay for it.

Properties of Pattern Calculi

- Patterns themselves can be reduced and instantiated.
- It makes pattern calculi expressive, but there is a price to pay for it.
- Good properties of the lambda calculus (confluence, termination of reduction in the presence of types) are lost.
- Restrictions are needed to recover them.

Example of Non-Confluence

- Assume matching is done syntactically (not modulo β-reduction).
- The term $(\lambda(x\,a).\,x)\,((\lambda y.y)\,a)$ can be reduced in two different ways to non-joinable terms:
  - $(\lambda(x\,a).\,x)\,((\lambda y.y)\,a) \to \lambda y.y.$
  - $(\lambda(x\,a).\,x)\,((\lambda y.y)\,a) \to (\lambda(x\,a).\,x)\,a.$

Confluence

- Confluence is a desirable property.
- It allows to reason about programs with respect to any convenient sequence of reductions, since the other reductions lead to the same result.

Sufficient Conditions for Confluence for Unitary Matching

Various works on establishing conditions for confluence when matching is unitary:

- van Oostrom, 1990,
- Cirstea and Faure, 2007,
- Klop et al, 2008,
- Jay and Kesner, 2009.

Sufficient Conditions for Confluence for Unitary Matching

Various works on establishing conditions for confluence when matching is unitary:

- van Oostrom, 1990,
- Cirstea and Faure, 2007,
- Klop et al, 2008,
- Jay and Kesner, 2009.

But we have finitary matching...

Confluence for Finitary Matching

How to deal with multiple reductions caused by multiple matchers?

- Commutative f.
- $(\lambda f(x, y).x)f(a, b) \to a$.
- $(\lambda f(x, y).x)f(a, b) \to b$.

Confluence for Finitary Matching

How to deal with multiple reductions caused by multiple matchers?

- Commutative f.
- $(\lambda f(x, y).x) f(a, b) \to a$.
- $(\lambda f(x, y).x) f(a, b) \to b$.

Idea: Permit term sums as terms:

$$(\lambda f(x, y).x) f(a, b) \to a + b.$$

$+$ should be associative, commutative, idempotent, and application should distribute over it (the ACID property).

Confluence for Finitary Matching

The rule for β-reduction:

$$(\lambda_V P.N)\, Q \rightarrow N\varphi_1 + \cdots + N\varphi_n,$$
$$\text{where } solve(P \ll_V Q) = \{\varphi_1, \ldots, \varphi_n\},\ n \geq 1.$$

$solve$ is a parameter: a matching function.

Confluence for Finitary Matching

- Properties of $solve$ affect confluence.
- We proved confluence when $solve$ satisfies three conditions:
  - matchers introduce no new free variables,
  - matching is stable under substitution application,
  - matching is stable under reduction.

Instances of the Matching Function

- Our proof is generic, for any finitary matching function that satisfies the confluence conditions.
- From it one can obtain confluence proofs for concrete instantiations of the underline matching.
- We presented three concrete instances of the matching function:
  - free sequence matching (and its special case, commutative matching),
  - unordered sequence matching,
  - sequence matching with linear algebraic patterns.

Summary

- We defined CLP(SC) with a sound and terminating constraint solver over the domain of sequences and contexts.
- We implemented the PρLog language and applied to several domains (rewriting, XML processing, Web reasoning).
- We defined a finitary pattern calculus with sequence variables and proved its confluence under certain conditions on the matching function.

Future Work

- Define higher-order typed term language with sequence variables.
- Study computationally well-behaved fragments of higher-order matching with sequence variables.
- Construction of rewriting rules over the proposed term language.
- Investigate syntactic restrictions for rewrite systems under which confluence and termination hold.