Literature Study on Algorithms for Pattern Completeness

Daniel Köves

Femke van Raamsdonk
Jörg Endrullis

Vrije Universiteit Amsterdam

November 2024

Outline

Functional Programming Example I/III

Functional programs are written using pattern matching.
Critical to ensure that these patterns are complete, otherwise:

- Runtime errors
- Untimely termination

Example:

```
first :: [Int] -> Maybe Int
first (x:_) = Just x
```

Notice the missing case for [].

Functional Programming Example II/III

Now when we run this:

```haskell
main :: IO ()
main = do
    print $ first [1,2,3]
    print $ first []
```

Our program crashes:

```
$ ./incomplete-pattern
Just 1
incomplete-pattern: incomplete-pattern.hs:2:1-21:
    Non-exhaustive patterns in function first
```

Functional Programming Example III/III

We can use GHC to identify these cases:

```
incomplete-pattern.hs:2:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘first’:
        Patterns of type ‘[Int]’ not matched: []
  |
2 | first (x:xs) = Just x
  | ^^^^^^^^^^^^^^^^^^^^^
```

How do we do this algorithmically?

Term Rewriting I/II

Idea:

- Analyse programs as term-rewrite systems
- Goal: decide when a function f (like our function `first`) is *pattern complete*

Ingredients of term rewrite systems:

- set of function symbols Σ with arity $\#$
- set of rewrite rules $\ell \to r$
- set variables $\mathcal{X}$
- terms $\mathcal{T}(\Sigma, \mathcal{X})$ from function symbols and variables

Term Rewriting II/II

Example term rewrite system $\mathcal{R}$:

- Function symbols
  - Constructors: true, false, 0 (constants), s (unary)
  - Defined symbol: even (unary)
- Rewrite rules:
  - $\text{even}(0) \rightarrow \text{true}$
  - $\text{even}(s(0)) \rightarrow \text{false}$
  - $\text{even}(s(s(x))) \rightarrow \text{even}(\text{x})$

Example – Reduction

$\text{even}(s(s(s(0)))) \rightarrow \text{even}(s(0)) \rightarrow \text{false}$
$\text{even}(s(s(s(s(0))))) \rightarrow \text{even}(s(s(0))) \rightarrow \text{even}(0) \rightarrow \text{true}$

Matching I/II

Definition

Matching problem: given terms s and t, find substitution σ from $\mathcal{X}$ to $\mathcal{T}(\Sigma, \mathcal{X})$ such that $s\sigma = t$.

Example

- Match z to 0. Take $\sigma = \{z \mapsto 0\}$
- Match even(z) to 0. No such σ exists
- Match $f(a, b)$ to $f(x, x)$. Take $\sigma = \{a \mapsto x, b \mapsto x\}$
- Match $f(a, a)$ to $f(x, s(x))$. No such σ exists

Matching II/II

Idea:

- Represent defined function f as TRS
- Match input term $f(z_1, ..., z_n)$ to LHS of TRS (domain of f) with z_i some constructor term
- If for all constructor substitution for z_i we find a match: f is pattern complete

Example

Previous TRS with defined symbol even. Matching problems:

- even(z) to even(0)
- even(z) to even(s(0))
- even(z) to even(s(s(x)))

Outline

Properties

- Algorithm to decide pattern completeness
- **Input**: matching problems $(f(z), \ell)$
  - For each LHS ℓ of TRS from f
  - z some arbitrary constructor term
- **Output**: success or failure
- Flow:
  - Iteratively decompose terms until match/clash
  - If we encounter z with ℓ not a variable, instantiate z via $\sigma = \{z \mapsto c(x_1, ..., x_n)\}$ for each constructor c

Example I

Constructors: 0, s, function f: $f(0) \to 0$ and $f(s(x)) \to x$

Algorithm

$(f(z), f(0))$		$(f(z), f(s(x)))$	
$(z, 0)$		$(z, s(x))$	
$(0, 0)$	$(0, s(x))$	$(s(z), 0)$	$(s(z), s(x))$
match	clash	clash	(z, x)
			match$^{\sigma=\{x \mapsto z\}}$

Result: **success**, since each constructor substitution for z results in match

Example I

Constructors: 0, s, function f: $f(0) \to 0$ and $f(s(x)) \to x$

Algorithm

$(f(z), f(0))$
$(z, 0)$

- $(0, 0)$ match
- $(0, s(x))$ clash

$(f(z), f(s(x)))$
$(z, s(x))$

- $(s(z), 0)$ clash
- $(s(z), s(x))$
  (z, x)
  match$^{\sigma=\{x \mapsto z\}}$

Result: **success**, since each constructor substitution for z results in match

Example I

Constructors: 0, s, function f: $f(0) \rightarrow 0$ and $f(s(x)) \rightarrow x$

Algorithm

$$\dfrac{\dfrac{(f(z), f(0))}{(z, 0)}}{\begin{array}{c|c} (0,0) & (0, s(x)) \\ \text{match} & \text{clash} \end{array}} \quad \Bigg| \quad \dfrac{\dfrac{(f(z), f(s(x)))}{(z, s(x))}}{\begin{array}{c|c} (s(z), 0) & (s(z), s(x)) \\ \text{clash} & (z, x) \\ & \text{match}^{\sigma=\{x \mapsto z\}} \end{array}}$$

Result: **success**, since each constructor substitution for z results in match

Example I

Constructors: 0, s, function f: $f(0) \to 0$ and $f(s(x)) \to x$

Algorithm

$$\dfrac{\begin{matrix}(f(z), f(0)) \\ (z, 0)\end{matrix}}{\begin{matrix}(0,0) \\ \text{match}\end{matrix} \;\Big|\; \begin{matrix}(0, s(x)) \\ \text{clash}\end{matrix}} \;\Bigg|\; \dfrac{\begin{matrix}(f(z), f(s(x))) \\ (z, s(x))\end{matrix}}{\begin{matrix}(s(z), 0) \\ \text{clash}\end{matrix} \;\Big|\; \begin{matrix}(s(z), s(x)) \\ (z, x) \\ \text{match}^{\sigma=\{x \mapsto z\}}\end{matrix}}$$

Result: **success**, since each constructor substitution for z results in match

Example II – Incomplete pattern

Constructors: 0, s, function f: $f(0) \to 0$ (missing case for s(x)).

Algorithm

$$(f(z), f(0))$$
$$(z, 0)$$

$(0, 0)$	$(s(z), 0)$
match	clash

Result: **failure**, since there's no match for substitution $\sigma = \{z \mapsto s(z)\}$

Example II – Incomplete pattern

Constructors: 0, s, function f: $f(0) \rightarrow 0$ (missing case for s(x)).

Algorithm

$$(f(z), f(0))$$
$$(z, 0)$$

$(0, 0)$	$(s(z), 0)$
match	clash

Result: **failure**, since there's no match for substitution $\sigma = \{z \mapsto s(z)\}$

Example II – Incomplete pattern

Constructors: 0, s, function f: $f(0) \to 0$ (missing case for s(x)).

Algorithm

$$\frac{\begin{array}{c}(f(z), f(0)) \\ (z, 0)\end{array}}{\begin{array}{c}(0, 0) \\ \text{match}\end{array} \quad \Bigg| \quad \begin{array}{c}(s(z), 0) \\ \text{clash}\end{array}}$$

Result: **failure**, since there's no match for substitution $\sigma = \{z \mapsto s(z)\}$

Example II – Incomplete pattern

Constructors: 0, s, function f: $f(0) \to 0$ (missing case for s(x)).

Algorithm

$$\frac{\begin{array}{c}(f(z), f(0)) \\ (z, 0)\end{array}}{\begin{array}{c}(0, 0) \\ \text{match}\end{array} \quad\Big|\quad \begin{array}{c}(s(z), 0) \\ \text{clash}\end{array}}$$

Result: **failure**, since there's no match for substitution $\sigma = \{z \mapsto s(z)\}$

Example III – General case

Function f: with LHS $f(x,x)$ and $f(x,y)$
Pattern $f(x,x)$ is called *non-linear*, due to the repeated variable x

Algorithm

$(f(a,b), f(x,x))$	$(f(a,b), f(x,y))$
$(a,x), (b,x)$	$(a,x), (b,y)$
clash	match $\sigma_1 = \{x \mapsto a\}$ $\sigma_2 = \{y \mapsto b\}$

- Result: **success**, since right-side matches both a and b (which are arbitrary constructor terms)
- Left side results in clash since we cannot match variable x to both a and b

Example III – General case

Function f: with LHS $f(x,x)$ and $f(x,y)$
Pattern $f(x,x)$ is called *non-linear*, due to the repeated variable x

Algorithm

$(f(a,b), f(x,x))$	$(f(a,b), f(x,y))$
$(a,x),(b,x)$	$(a,x),(b,y)$
clash	match $\sigma_1=\{x \mapsto a\}$ $\sigma_2=\{y \mapsto b\}$

- Result: **success**, since right-side matches both a and b (which are arbitrary constructor terms)
- Left side results in clash since we cannot match variable x to both a and b

Example III – General case

Function f: with LHS $f(x,x)$ and $f(x,y)$
Pattern $f(x,x)$ is called *non-linear*, due to the repeated variable x

Algorithm

$(f(a,b), f(x,x))$	$(f(a,b), f(x,y))$
$(a,x),(b,x)$	$(a,x),(b,y)$
clash	match $\sigma_1 = \{x \mapsto a\}$ $\sigma_2 = \{y \mapsto b\}$

- Result: **success**, since right-side matches both a and b (which are arbitrary constructor terms)
- Left side results in clash since we cannot match variable x to both a and b

Example III – General case

Function f: with LHS $f(x,x)$ and $f(x,y)$
Pattern $f(x,x)$ is called *non-linear*, due to the repeated variable x

Algorithm

$(f(a,b), f(x,x))$	$(f(a,b), f(x,y))$
$(a,x),(b,x)$	$(a,x),(b,y)$
clash	match $\sigma_1 = \{x \mapsto a\}$ $\sigma_2 = \{y \mapsto b\}$

- Result: **success**, since right-side matches both a and b (which are arbitrary constructor terms)
- Left side results in clash since we cannot match variable x to both a and b

Outline

Properties

- Due to Lazrek et al.
- Can be used to decide pattern completeness
- **Input**: TRS $\mathcal{R}$ and defined symbol f
- **Output**: Set of constructors where f is not defined

Idea

- Start with set M_0 with LHSs of function f, set N_0 $f(z_1, \ldots, z_n)$ where z_i is some constructor term
- Try to unify elements of $m \in M_i$ and $n \in N_i$ with substitution σ
- Compute *complement* of this substitution ρ
- Replace matched element in N_i with new elements $n\rho$
- Repeat until either M_{last} or N_{last} is empty, or no further unification is possible
- If M_{last} is empty but N_{last} is not empty, f is not defined on the terms in N_{last}

Example I - Linear case

Constructors: 0, s, function f: $f(0) \to 0$ and $f(s(x)) \to x$

Algorithm

	M	N	σ	ρ
0	$f(0), f(s(x))$	$f(z)$	$z \mapsto 0$	$z \mapsto s(z)$
1	$f(s(x))$	$f(s(z))$	$z \mapsto x$	
2	$\varnothing$	$\varnothing$		

Result: **success**, N_2 is empty

Example I - Linear case

Constructors: 0, s, function f: $f(0) \to 0$ and $f(s(x)) \to x$

Algorithm

	M	N	σ	ρ
0	$f(0), f(s(x))$	$f(z)$	$z \mapsto 0$	$z \mapsto s(z)$
1	$f(s(x))$	$f(s(z))$	$z \mapsto x$	
2	$\varnothing$	$\varnothing$		

Result: **success**, N_2 is empty

Example I - Linear case

Constructors: 0, s, function f: $f(0) \to 0$ and $f(s(x)) \to x$

Algorithm

	M	N	σ	ρ
0	$f(0), f(s(x))$	$f(z)$	$z \mapsto 0$	$z \mapsto s(z)$
1	$f(s(x))$	$f(s(z))$	$z \mapsto x$	
2	$\varnothing$	$\varnothing$		

Result: **success**, N_2 is empty

Example I - Linear case

Constructors: 0, s, function f: $f(0) \to 0$ and $f(s(x)) \to x$

Algorithm

	M	N	σ	ρ
0	$f(0), f(s(x))$	$f(z)$	$z \mapsto 0$	$z \mapsto s(z)$
1	$f(s(x))$	$f(s(z))$	$z \mapsto x$	
2	$\varnothing$	$\varnothing$		

Result: **success**, N_2 is empty

Example II - Incomplete pattern

Constructors: 0, s, function f: LHS $f(0)$ (missing case for $s(x)$).

Algorithm

	M	N	σ	ρ
0	$f(s(x))$	$f(z)$	$z \mapsto s(x)$	$z \mapsto 0$
1	$\emptyset$	$f(0)$		

Result: **failure**, N_1 is not empty

Example II - Incomplete pattern

Constructors: 0, s, function f: LHS $f(0)$ (missing case for $s(x)$).

Algorithm

	M	N	σ	ρ
0	$f(s(x))$	$f(z)$	$z \mapsto s(x)$	$z \mapsto 0$
1	$\varnothing$	$f(0)$		

Result: **failure**, N_1 is not empty

Example II - Incomplete pattern

Constructors: 0, s, function f: LHS $f(0)$ (missing case for $s(x)$).

Algorithm

	M	N	σ	ρ
0	$f(s(x))$	$f(z)$	$z \mapsto s(x)$	$z \mapsto 0$
1	$\emptyset$	$f(0)$		

Result: **failure**, N_1 is not empty

Outline

Comparison

- Both Thiemann and Yamada's and the complement algorithm can be used to decide pattern completeness
- Complement algorithm's N_{last} set contains patterns where f still needs to be defined
- Thiemann and Yamada's algorithm is proven to work for non-linear patterns, whereas complement algorithm might fail
- Complement algorithm has built-in counterexample-generation

Conclusion

- Literature study to compare algorithms for pattern completeness
- Detailed comparison between Thiemann and Yamada's algorithm and the complement algorithm of Lazrek et al.
- Further research could:
  - Perform more thorough performance comparison between algorithms
  - More analysis as to why Thiemann and Yamada's version outperforms the other variants
  - Suggestion as per Thiemann and Yamada: construct a similar syntax-based algorithm to decide quasi-reducibility

Further notable work

- Calculus of components by Thiel on which the complement algorithm by Lazrek et al is based on
- Decidability of quasi-reducibility by Kapur et al.
- Aota and Toyama introduce *strong quasi-reducibility*
- Kop derives quasi-reducibility of logically constrained TRSs
- Bouhoula et al. use tree-automata based algorithm to decide sufficient completeness

References I

[1] René Thiemann and Akihisa Yamada. "A Verified Algorithm for Deciding Pattern Completeness". In: *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*. Ed. by Jakob Rehof. Vol. 299. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 27:1–27:17. ISBN: 978-3-95977-323-2. DOI: 10.4230/LIPIcs.FSCD.2024.27. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2024.27.

[2] Azeddine Lazrek, Pierre Lescanne, and Jean-Jacques Thiel. "Tools for proving inductive equalities, relative completeness, and ω-completeness". *Information and Computation* 84.1 (1990), pp. 47–70. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(90)90033-E. URL: https://www.sciencedirect.com/science/article/pii/089054019090033E.

[3] Jean Jacques Thiel. "Stop losing sleep over incomplete data type specifications". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: Association for Computing Machinery, 1984, pp. 76–82. ISBN: 0897911253. DOI: 10.1145/800017.800518. URL: https://doi.org/10.1145/800017.800518.

References II

[4] Deepak Kapur, Paliath Narendran, and Hantao Zhang. "On sufficient-completeness and related properties of term rewriting systems". *Acta Inf.* 24.4 (Aug. 1987), pp. 395–415. ISSN: 0001-5903. DOI: 10.1007/BF00292110. URL: https://doi.org/10.1007/BF00292110.

[5] Cynthia Kop. *Quasi-reductivity of Logically Constrained Term Rewriting Systems*. 2017. arXiv: 1702.02397 [cs.LO]. URL: https://arxiv.org/abs/1702.02397.

[6] Aart Middeldorp, Alexander Lochmann, and Fabian Mitterwallner. "First-Order Theory of Rewriting for Linear Variable-Separated Rewrite Systems: Automation, Formalization, Certification". 67.2 (Apr. 2023). ISSN: 0168-7433. DOI: 10.1007/s10817-023-09661-7. URL: https://doi.org/10.1007/s10817-023-09661-7.

References III

[7] Takahito Aoto and Yoshihito Toyama. "Ground Confluence Prover based on Rewriting Induction". In: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 33:1–33:12. ISBN: 978-3-95977-010-1. DOI: 10.4230/LIPIcs.FSCD.2016.33. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2016.33.

[8] Adel Bouhoula and Florent Jacquemard. "Sufficient completeness verification for conditional and constrained TRS". *Journal of Applied Logic* 10.1 (2012). Special issue on Automated Specification and Verification of Web Systems, pp. 127–143. ISSN: 1570-8683. DOI: https://doi.org/10.1016/j.jal.2011.09.001. URL: https://www.sciencedirect.com/science/article/pii/S1570868311000413.