Relational Synthesis of Pattern Matching

Dmitry Kosarev
Saint Petersburg State University and
JetBrains Research, Russia
Dmitrii.Kosarev@protonmail.ch

Dmitry Boulytchev
Saint Petersburg State University and
JetBrains Research, Russia
dboulytchev@math.spbu.ru

We present an approach to pattern matching code generation based on application of relational programming [4, 2] and, in particular, relational interpreters [3].

In a simplified case, we consider a finite set of constructors with arities $\mathscr{C}$, a set of values $\mathscr{V}$, and a set of patterns $\mathscr{P}$

$$\begin{array}{rcl} \mathscr{C} & = & \{C_1^{k_1}, \ldots, C_n^{k_n}\} \\ \mathscr{V} & = & \mathscr{C}\,\mathscr{V}^* \\ \mathscr{P} & = & \_ \mid \mathscr{C}\,\mathscr{P}^* \end{array}$$

and address a problem of matching a value (*scrutinee*) against an ordered list of patterns.

Our approach is based on two representations of pattern matching semantics. First, we use *declarative semantics*, representing it as a relation

$$match\,(s, ps, i)$$

where $s \in \mathscr{V}$ is a scrutinee, $ps \in \mathscr{P}^*$ — an ordered list of patterns, and $i \in \mathbb{N}$ — a natural number. This relation holds iff s matches the i-th pattern of ps. For a fixed language of patterns $match$ can be implemented directly in MINIKANREN once and for all.

On the other hand, we introduce a simple language $\mathscr{S}$ of test-and-branch constructs:

$$\begin{array}{rcl} \mathscr{M} & = & \bullet \\ & & \mathscr{M}\,[\mathbb{N}] \\ \mathscr{S} & = & \textbf{return}\,\mathbb{N} \\ & & \textbf{switch}\ \mathscr{M}\ \textbf{with}\ [\mathscr{C} \to \mathscr{S}]^*\ \textbf{otherwise}\ \mathscr{S} \end{array}$$

Here $\mathscr{M}$ stands for a *matchable expression*, which is either a reference to a scrutinee ("$\bullet$") or a denotation of some indexed subvalue of a matchable expression. The programs in $\mathscr{S}$ can discriminate on the structure of matchable expressions, testing their top constructors and eventually returning natural numbers as results. The language $\mathscr{S}$ is similar to the intermediate representations for pattern matching code, used in previous works on pattern matching implementation [6, 7].

We use a *relational interpreter* for $\mathscr{S}$

$$eval^o_{\mathscr{S}}\,(s, p, i)$$

Here s and i have the same meaning as in declarative semantics description, $p \in \mathscr{S}$ — a syntactic representation of a program in $\mathscr{S}$. The relation $eval^o_{\mathscr{S}}$ encodes the operational semantics of $\mathscr{S}$; it holds, if evaluating p for s returns i. Being relational interpreter, however, $eval^o_{\mathscr{S}}$ is capable of solving a synthesis problem: by a scrutinee s and a number i calculate a program p which makes the relation to hold. Within this setting, we can formulate the pattern-matching synthesis problem as follows: *for a given ordered list of patterns ps find a program p, such that*

Submitted to:
TEASE-LP 2020

$$\forall s \in \mathscr{V}, \forall i \in \mathbb{N}, eval^o_{\mathscr{S}}(s,p,i) \wedge match(s,ps,i)$$

It is rather problematic to directly solve this synthesis problem with existing MINIKANREN implementations as they provide a rather limited support for universal quantification [1, 8]. However, in our concrete case there is a straightforward way to alleviate this problem. Indeed, we may replace universal quantification over i by a finite conjunction, as the length of ps is known at the synthesis time. As for the quantification over s, for any concrete ps and type of scrutinee we may compute a *complete set of examples* $\mathscr{E}(ps) \subseteq \mathscr{V}$ with the following property:

$$\begin{aligned} \forall p \in \mathscr{S} \quad : \quad & [\forall s \in \mathscr{E}(ps), \forall i \in \mathbb{N} : match(s,ps,i) \Longleftrightarrow eval^o_{\mathscr{S}}(s,p,i)] \Longrightarrow \\ & [\forall s \in \mathscr{V}, \forall i \in \mathbb{N} : match(s,ps,i) \Longleftrightarrow eval^o_{\mathscr{S}}(s,p,i)] \end{aligned}$$

It easy to see, that for arbitrary ps there exists a finite complete set of examples (indeed, any patterns describes the "shape" of a scrutinee up to some finite depth, beyond which all scrutinees become indistinguishable). Thus, for a given ps and a type τ of scrutinee we may completely eliminate the quantification by enumerating all inhabitants of type τ up to finite depth, reformulating the synthesis problem as

$$\bigwedge_{i \in [1 \ldots |ps|]} \bigwedge_{s \in \mathscr{E}(ps)} (eval^o_{\mathscr{S}}(s,p,i) \wedge match(s,ps,i))$$

During synthesis we aim to generate programs that have less checks in its' bodies and expect them to show better performance.

We implemented the synthesis framework using OCANREN — an embedding of MINIKANREN into OCAML [5], — and evaluated it on the set of benchmarks, reported in the previous works on *ad-hoc* algorithms for pattern matching code generation [6, 7]. In comparison with a simplified setting, presented above, our implementation deals with a more elaborate pattern matching problem — in particular, we can support *guard expressions*, name bindings in patterns and incorporate a deterministic top-down matching strategy, which is common in functional languages. Of course, conventional techniques to deal with these are still applicable, although, for example, for guards it will require postprocessing of generated programs.

Initially, our synthesis did not demonstrate adequate results. However, we applied the following techniques to improve both the performance and the quality of synthesized programs:

- we implemented a pruning technique, which makes the search stop exploring a certain branch if the program, synthesized so far, contains too much nesting constructs (this factor can be precomputed by patterns analysis) or is strictly worse then already synthesized one;
- we restricted the number of `switch` branches using type information about subexpressions of scrutinee.

With these adjustments, our synthesis framework in a negligible time provides the same results as those reported in the previous works. However for most types of scrutinee an amount of required examples and size of search space is exponential. Our future steps (besides performance optimizations) include extending the pattern matching language to completely match that of OCAML (for now we do not support GADTs), integrate the synthesis into the existing OCAML compiler and evaluate it on a set of real-world programs. Another direction is extending the pattern matching language to incorporate features which are known to be hard, tedious or error-prone to implement (for example, non-linear patterns).

An alternative for our approach can be using SyGuS where algebraic data types support was recently added [9] to the language. Although we don't know any tools that already support new standard.

References

[1] William Byrd: *Relational Synthesis of Programs*. In: *Unpublished manuscript*.

[2] William E. Byrd (2009): *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph.D. thesis, Indiana University.

[3] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt & Matthew Might (2017): *A unified approach to solving seven programming problems (functional pearl)*. *PACMPL* 1(ICFP), pp. 8:1–8:26, doi:10.1145/3110252. Available at `https://doi.org/10.1145/3110252`.

[4] Daniel P. Friedman, William E. Byrd & Oleg Kiselyov (2005): *The reasoned schemer*. MIT Press.

[5] Dmitry Kosarev & Dmitry Boulytchev (2016): *Typed Embedding of a Relational Language in OCaml*, pp. 1–22. doi:10.4204/EPTCS.285.1. Available at `https://doi.org/10.4204/EPTCS.285.1`.

[6] Fabrice Le Fessant & Luc Maranget (2001): *Optimizing Pattern Matching*. *SIGPLAN Not.* 36(10), p. 26–37, doi:10.1145/507669.507641. Available at `https://doi.org/10.1145/507669.507641`.

[7] Luc Maranget (2008): *Compiling Pattern Matching to Good Decision Trees*. In: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, Association for Computing Machinery, New York, NY, USA, p. 35–46, doi:10.1145/1411304.1411311. Available at `https://doi.org/10.1145/1411304.1411311`.

[8] Eugene Moiseenko (2019): *Constructive Negation for miniKanren*. In: *miniKanren and Relational Programming Workshop*.

[9] Mukund Raghothaman, Andrew Reynolds & Abhishek Udupa: *The SyGuS Language Standard Version 2.0*.