Relational Synthesis for Pattern Matching*

DMITRY KOSAREV, Saint Petersburg State University, Russia and JetBrains Research, Russia
DMITRY BOULYTCHEV, Saint Petersburg State University, Russia and JetBrains Research, Russia

We apply relational programming techniques to the problem of synthesizing efficient implementation for a pattern matching construct. Although in principle pattern matching can be implemented in a trivial way, the result suffers from inefficiency in terms of both performance and code size. Thus, in implementing functional languages alternative, more elaborate approaches are widely used. However, as there are multiple kinds and flavors of pattern matching constructs, these approaches have to be specifically developed and justified for each concrete inhabitant of the pattern matching "zoo." We formulate the pattern matching synthesis problem in relational terms and develop optimizations which improve the efficiency of the synthesis and guarantee the optimality of the result. Our approach is based on relational representations of both the high-level semantics of pattern matching and the semantics of an intermediate-level implementation language. This choice make our approach, in principle, more scalable as we only need to modify the high-level semantics in order to synthesize the implementation of a new feature. Our evaluation on a set of small samples, partially taken from existing literature shows, that our framework is capable of synthesizing optimal implementations quickly. Our in-depth stress evaluation on a number of artificial benchmarks, however, has shown the need for future improvements.

CCS Concepts: • **Software and its engineering** → **Constraint and logic languages**; **Source code generation**;

Additional Key Words and Phrases: relational programming, relational interpreters, pattern matching

1 INTRODUCTION

Algebraic data types (ADT) are an important tool in functional programming which deliver a way to represent flexible and easy to manipulate data structures. To inspect the contents of an ADT's values a generic construct — *pattern matching* — is used. Pattern matching can be considered as a generalization of conventional conditional control-flow construct "`if` .. `then` .. `else`" and in principle can be decomposed into a nested hierarchy of those; from this standpoint the problem of pattern matching implementation can be considered trivial. However, some decompositions are obviously better than others. We repeat here an example from [Maranget 2008] to demonstrate this difference (see Fig. 1). Here we match a triple of boolean values x, y, and z against four patterns (Fig. 1a; we use OCaml [Leroy et al. 2020] as reference language). The naïve implementation of this example is shown on Fig. 1b; however if we decide to match y first the result becomes much better (Fig. 1c).

The quality of a pattern matching implementation can be measured in various ways. One can either optimise the run time cost by minimizing the amount of checks performed, or the static cost by minimizing the size of the generated code. *Decision trees* are considered suitable for the first criterion as they check every subexpression no more than once. However, minimizing the size of decision tree is known to be NP-hard [Baudinet and MacQueen 1985], and as a rule various heuristics, using, for example, the number of nodes, the length of the longest path

*This work was partially supported by the grant 18-01-00380 from The Russian Foundation for Basic Research

Authors' addresses: Dmitry Kosarev, Saint Petersburg State University, Russia , JetBrains Research, Russia, Dmitrii.Kosarev@pm.me; Dmitry Boulytchev, Saint Petersburg State University, Russia , JetBrains Research, Russia, dboulytchev@math.spbu.ru.

miniKanren.org/workshop/2021/8-ART8

```
match x, y, z with
| _, F, T → 1
| F, T, _ → 2
| _, _, F → 3
| _, _, T → 4
```

(a) Pattern matching

```
if x then
  if y then
    if z then 4 else 3
  else
    if z then 1 else 3
else
  if y then 2
  else
    if z then 1 else 3
```

(b) A correct but non-optimal implementation

```
if y then
  if x then
    if z then 4 else 3
  else 2
else
  if z then 1 else 3
```

(c) Optimal implementation

Fig. 1. Pattern matching implementation example

and the average length of all paths are applied during compilation. In [Scott and Ramsey 2000] the results of experimental evaluation of nine heuristics for Standard ML of New Jersey are reported.

For minimizing the static cost *backtracking automata* can be used since they admit a compact representation but in some cases can perform repeated checks.

There is a certain difference in dealing with pattern matching in strict and non-strict languages. For strict languages checking sub-expressions of the scrutinee in any order is allowed. The pattern matching implementation for strict languages can operate in *direct* or *indirect* styles. In the direct style the construction of an implementation is done explicitly. In indirect the construction of implementation requires some post-processing, which can vary from easy simplifications to complicated supercompilation techniques [Sestoft 1996]. The main drawback of indirect approach is that the size of intermediate data structures can be exponentially large.

For non-strict languages pattern matching should evaluate only those sub-expressions which are necessary for performing pattern matching. If not done carefully pattern matching can change the termination behavior of the program. In general non-strict languages put more constraints on pattern matching and thus admit a smaller set of heuristics. A few approaches for checking sub-expressions in lazy languages have been proposed. In [Augustsson 1985] a simple left-to-right order of subexpression checking was proposed with a proof that this particular order doesn't affect termination. The backtracking automaton being built takes a form of a DAG to reduce the code size. A few refinements have been added in [Wadler 1987] as a part of textbook [Peyton Jones 1987] on the implementation of lazy functional languages. The approach from this book is used in the current version of GHC [Marlow and Peyton Jones 2012]. [Laville 1991] models values in lazy languages using *partial terms*, although it doesn't scale to types with infinite sets of constructors (like integers). The approach doesn't test all subexpressions from left to right as does [Augustsson 1985] but aims to avoid performing unnecessary checks by constructing *lazy automaton*. Pattern matching for lazy languages has been compiled also to decision trees [Maranget 1992] and later into *decision DAGs* which in some cases allows the compiler to make the code smaller [Maranget 1994].

The inefficiency of backtracking automata have been improved in [Le Fessant and Maranget 2001]. The approach utilizes a matrix representation for pattern matching. It splits the current matrix according to constructors in the first column and reduces the task to compiling matrices with fewer rows. The technique is indirect; in the end a few optimizations are performed by introducing special *exit* nodes to the compiled representation. The approach from this paper is used in the current implementation of the OCaml compiler.

The previous approach uses the first column to split the matrix. In [Maranget 2008] the *necessity* heuristic has been introduced which recommends which column should be used to perform the split. Good decision trees which are constructed in this work can perform better in corner cases than [Le Fessant and Maranget 2001], but for practical use the difference is insignificant.

While existing approaches deliver appropriate solutions for certain forms of pattern matching constructs, they have to be extended in an *ad hoc* manner each time the syntax and semantics of pattern matching construct changes. For example, besides a simple conventional form of pattern matching there are a number of extensions: guards (first appeared in KRC language [Turner 2013]), disjunctive patterns [Leroy et al. 2020], non-linear patterns [McBride et al. 1969], active patterns [Syme et al. 2007] and pattern matching for polymorphic variants [Garrigue 1998] which require a separate customized algorithms to be developed.

We present an approach to pattern matching implementation based on application of relational programming [Byrd 2009; Friedman et al. 2005] and, in particular, relational interpreters [Byrd et al. 2017] and relational conversion [Lozov et al. 2017]. Our approach is based on relational representation of the source language pattern matching semantics on the one hand, and the semantics of the intermediate-level implementation language on the other. We formulate the condition necessary for a correct and complete implementation of pattern matching and use it to construct a top-level goal which represents a search procedure for all correct and complete implementations. We also present a number of techniques which make it possible to come up with an *optimal* solution as well as optimizations to improve the performance of the search. Similarly to many other prior works we use the size of the synthesized code, which can be measured statically, to distinguish better programs. Our implementation[1] makes use of OCanren[2] — a typed implementation of miniKanren for OCaml [Kosarev and Boulytchev 2016], and noCanren[3] — a converter from the subset of plain OCaml into OCanren [Lozov et al. 2017]. On an initial evaluation, performed for a set of benchmarks taken from other papers, showed our synthesizer performing well. However, being aware of some pitfalls of our approach, we came up with a set of counterexamples on which it did not provide any results in observable time, so we do not consider the problem completely solved. We also started to work on mechanized formalization[4], written in Coq [Bertot and Castéran 2004], to make the justification of our approach more solid and easier to verify, but this formalization is not yet complete.

2 THE PATTERN MATCHING SYNTHESIS PROBLEM

We start from a simplified view on pattern matching which does not incorporate some practically important aspects of the construct such as name bindings in patterns, guards or even semantic actions in branches. In a purified form, however, it represents the essence of pattern matching as an "inspect-and-branch" procedure. Once we come up with the solution for the essential part of the problem we embellish it with other features until it reaches a complete form.

First, we introduce a finite set of *constructors* $\mathcal{C}$, equipped with arities, a set of values $\mathcal{V}$ and a set of patterns $\mathcal{P}$:

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

We define a matching of a value v (*scrutinee*) against an ordered non-empty sequence of patterns $p_1, \ldots, p_k$ by means of the following relation

[1] https://github.com/kakadu/pat-match
[2] https://github.com/JetBrains-Research/OCanren
[3] https://github.com/Lozov-Petr/noCanren
[4] https://github.com/dboulytchev/Coq-matching-workout

$$\langle v;\ \_\rangle \qquad \text{[Wildcard]}$$

$$\frac{\forall i\ \ \langle v_i;\ p_i\rangle}{\left\langle C^k\, v_1 \ldots v_k;\ C^k\, p_1 \ldots p_k\right\rangle},\ k \geq 0 \qquad \text{[Constructor]}$$

Fig. 2. Matching against a single pattern

$$\frac{\langle v;\ p_1\rangle}{i \vdash \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow_* i} \qquad \text{[MatchHead]}$$

$$\frac{\neg\, \langle v;\ p_1\rangle \qquad i+1 \vdash \langle v;\ p_2, \ldots, p_k\rangle \longrightarrow_* j}{i \vdash \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow_* j} \qquad \text{[MatchTail]}$$

$$i \vdash \langle v;\ \varepsilon\rangle \longrightarrow_* i \qquad \text{[MatchOtherwise]}$$

$$\frac{1 \vdash \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow_* i}{\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i} \qquad \text{[Match]}$$

Fig. 3. Matching against an ordered sequence of patterns

$$\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i,\ 1 \leq i \leq k+1$$

which gives us the index of the leftmost matched pattern or $k+1$ if no such pattern exists. We use an auxiliary relation $\langle ; \rangle \subseteq \mathcal{V} \times \mathcal{P}$ to specify the notion of a value matched by an individual pattern (see Fig. 2). The rule [Wildcard] says that a wildcard pattern "_" matches any value, and [Constructor] specifies that a constructor pattern matches exactly those values which have the same constructor at the top level and all subvalues matched by corresponding subpatterns. The definition of "$\rightarrow$" is shown on Fig. 3. An auxiliary relation "$\rightarrow_*$" is introduced to specify the left-to-right matching strategy, and we use current index as an environment. An important rule, [MatchOtherwise] specifies that if we exhausted all the patterns with no matching we stop with the current index (which in this case is equal to the number of patterns plus one).

The relation "$\rightarrow$" gives us a *declarative* semantics of pattern matching. Since we are interested in synthesizing implementations, we need a *programmatical* view on the same problem. Thus, we introduce a language $\mathcal{S}$ (the "switch" language) of test-and branch constructs:

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

Here $\mathcal{M}$ stands for a *matching expression*, which is either a reference to a scrutinee "$\bullet$" or an indexed subexpression of scrutinee. Programs in the switch language can discriminate based on the structure of matching

$$v \vdash \bullet \longrightarrow_{\mathcal{M}} v \qquad \text{[Scrutinee]}$$

$$\frac{v \vdash m \longrightarrow_{\mathcal{M}} C^k v_1 \dots v_k}{v \vdash m[i] \longrightarrow_{\mathcal{M}} v_i} \qquad \text{[SubMatch]}$$

Fig. 4. Semantics of matching expression

$$v \vdash \textbf{return } i \longrightarrow_{\mathcal{S}} i \qquad \text{[Return]}$$

$$\frac{v \vdash m \longrightarrow_{\mathcal{M}} C^k\, v_1 \dots v_k \qquad v \vdash s \longrightarrow_{\mathcal{S}} i}{v \vdash \textbf{switch } m \textbf{ with } [C^k \to s]s^* \textbf{ otherwise } s' \longrightarrow_{\mathcal{S}} i} \qquad \text{[SwitchMatched]}$$

$$\frac{v \vdash m \longrightarrow_{\mathcal{M}} D^n\, v_1 \dots v_n \qquad C^k \neq D^n \qquad v \vdash \textbf{switch } m \textbf{ with } s^* \textbf{ otherwise } s' \longrightarrow_{\mathcal{S}} i}{v \vdash \textbf{switch } m \textbf{ with } [C^k \to s]s^* \textbf{ otherwise } s' \longrightarrow_{\mathcal{S}} i} \qquad \text{[SwitchNotMatched]}$$

$$\frac{v \vdash s \longrightarrow_{\mathcal{S}} i}{v \vdash \textbf{switch } m \textbf{ with } \varepsilon \textbf{ otherwise } s \longrightarrow_{\mathcal{S}} i} \qquad \text{[SwitchOtherwise]}$$

Fig. 5. Semantics of switch programs

expressions, testing their top-level constructors and eventually returning natural numbers as results. The switch language is similar to the intermediate representations for pattern matching code used in previous works on pattern matching implementation [Le Fessant and Maranget 2001; Maranget 2008].

The semantics of the switch language is given by mean of relations "$\rightarrow_{\mathcal{M}}$" and "$\rightarrow_{\mathcal{S}}$" (see Fig. 4 and 5). The first one describes the semantics of matching expression, while the second describes the semantics of the switch language itself. In both cases the scrutinee v is used as an environment ($v \vdash$).

The following observations can be easily proven by structural induction.

Observation 1. *For arbitrary pattern the set of matching values is non-empty:*

$$\forall p \in \mathcal{P} : \{v \in \mathcal{V} \mid \langle v;\, p\rangle\} \neq \varnothing$$

Observation 2. *Relations "$\rightarrow$" and "$\rightarrow_{\mathcal{S}}$" are functional and deterministic respectively:*

$$\forall p_1, \dots, p_k \in \mathcal{P},\ \forall v \in \mathcal{V},\ \forall \pi \in \mathcal{S} \quad : \quad |\{i \in \mathbb{N} \mid \langle v;\, p_1, \dots, p_k\rangle \longrightarrow i\}| = 1$$
$$|\{i \in \mathbb{N} \mid v \vdash \pi \longrightarrow_{\mathcal{S}} i\}| \leq 1$$

With these definitions, we can formulate the *pattern matching synthesis problem* as follows: for a given ordered sequence of patterns $p_1, \dots, p_k$ find a switch program π, such that

$$\forall v \in \mathcal{V},\ \forall 1 \leq i \leq n+1 : \langle v;\, p_1, \dots, p_n\rangle \longrightarrow i \iff v \vdash \pi \longrightarrow_{\mathcal{S}} i \qquad (\star)$$

In other words, program π delivers a correct and complete implementation for pattern matching.

3 PATTERN MATCHING SYNTHESIS, RELATIONALLY

In this section we describe a relational formulation for the pattern matching synthesis problem. Practically, this amounts to constructing a goal with a free variable corresponding to the switch program to synthesize for (arbitrary) list of patterns. In order to come up with a tractable goal certain steps have to be performed. We first describe the general idea, and then consider these steps is details.

Our idea of using relational programming for pattern matching synthesis is based on the following observations:

- For the switch language we can implement a relational interpreter $eval^o_{\mathcal{S}}$ with the following property: for arbitrary $v \in \mathcal{V}$, $\pi \in \mathcal{S}$ and $i \in \mathbb{N}$

$$eval^o_{\mathcal{S}}\ v\ \pi\ i \iff v \vdash \pi \longrightarrow_{\mathcal{S}} i$$

  In other words, $eval^o_{\mathcal{S}}$ interprets a program π for a scrutinee v and returns exactly the same branch (if any) which is prescribed by the semantics of the switch language.
- On the other hand, we can directly encode the declarative semantics of pattern matching as a relational program $match^o$ such that for arbitrary $v \in \mathcal{V}$, $p_i \in \mathcal{P}$ and $i \in \mathbb{N}$

$$match^o\ v\ p_1, \ldots, p_k\ i \iff \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i$$

  Again, $match^o$ succeeds with $1 \le i \le k$ iff p_i is the leftmost pattern, matching v; otherwise it succeeds with $i = k + 1$.

We address the construction of relational interpreters for both semantics in Section 3.1.

Being relational, both $eval^o_{\mathcal{S}}$ and $match^o$ do not just succeed or fail for ground arguments, but also can be *queried* for arguments with free logical variables, thus performing a search for all substitutions for these variables which make the relation hold. This observation leads us to the idea of utilizing the definition of the pattern matching synthesis problem, replacing "$\rightarrow$" with $match^o$, "$\rightarrow_{\mathcal{S}}$" with $eval^o$, and π with a free logical variable $\textcircled{?}$, which gives us the goal

$$\forall v \in \mathcal{V},\ \forall 1 \le i \le n+1 : match^o\ v\ p_1, \ldots, p_n\ i \iff eval^o\ v\ \textcircled{?}\ i$$

This goal, however, is problematic from relational point of view due to a number of reasons.

First, miniKanren provides rather a limited support for universal quantification. Apart from being inefficient from a performance standpoint, existing implementations either do not coexist with disequality constraints [Byrd [n. d.]] or do not support quantified goals with infinite number of answers [Moiseenko 2019]. As we will see below, both restrictions are violated in our case. Second, there is no direct support for the equivalence of goals ("$\Leftrightarrow$"). Thus, reducing the original synthesis problem to a viable relational goal involves some "massaging".

We eliminate the universal quantification over the infinite set of scrutinees, replacing it by a *finite* conjunction over a *complete set of samples*. For a sequence of patterns $p_1, \ldots, p_k$ a complete set of samples is a finite set of values $\mathcal{E}(p_1, \ldots, p_k) \subseteq \mathcal{V}$ with the following property:

$$\begin{aligned}\forall \pi \in \mathcal{S}\ \ :\ \ & [\forall v \in \mathcal{E}(p_1, \ldots, p_k),\ \forall i \in \mathbb{N} : \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i \iff v \vdash \pi \longrightarrow_{\mathcal{S}} i] \Longrightarrow \\ & [\forall v \in \mathcal{V},\ \forall i \in \mathbb{N} : \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i \iff v \vdash \pi \longrightarrow_{\mathcal{S}} i]\end{aligned}$$

In other words, if a program implements a correct and complete pattern matching for all values in a complete set of samples, then this program implements a correct and complete pattern matching for all values. The idea of using a complete set of samples originates from the following observation: each pattern describes a (potentially infinite) set of values, and pattern matching splits the set of all values into equivalence classes, each corresponding to a certain matching pattern. Moreover, the values of different classes can be distinguished only by looking

down to a *finite* depth (as different patterns can be distinguished in this way). The generation of complete sample set will be addressed below (see Section 3.2).

To eliminate the universal quantification over the set of answers we rely on the functionality of declarative pattern matching semantics. Indeed, given a fixed sequence $p_1, \ldots, p_k$ of patterns, for every value v there is exactly one answer value i, such that $\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i$. We can reformulate this property as

$$\exists i :\ \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i \Longrightarrow \left(\forall j : \langle v;\ p_1, \ldots, p_k\rangle \longrightarrow j \Longrightarrow j = i\right)$$

Thus, we can replace universal quantification over the sets of answers by existential one, for which we have an efficient relational counterpart — the "**fresh**" construct.

Following the same argument, we may replace the equivalence with conjunction: indeed, if

$$\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow i$$

for some i, then (by functionality), for any other $j \neq i$

$$\neg\ (\langle v;\ p_1, \ldots, p_k\rangle \longrightarrow j)$$

A correct pattern matching implementation π should satisfy the condition

$$v \vdash \pi \longrightarrow_{\mathcal{S}} i$$

But, by the determinism of the switch language semantics, it immediately follows, that for arbitrary $j \neq i$

$$\neg\ (v \vdash \pi \longrightarrow_{\mathcal{S}} j)$$

Thus, the goal we eventually came up with is

$$\bigwedge_{v \in \mathcal{E}\ (p_1,\ldots,p_k)} \textbf{fresh}\ (i)\ \{match^o\ v\ p_1, \ldots, p_k\ i \wedge eval^o_{\mathcal{S}}\ v\ \textcircled{?}\ i\} \qquad (\star\star)$$

From relational point of view this is a pretty conventional goal which can be solved by virtually any decent miniKanren implementation in which the relations $eval^o_{\mathcal{S}}$ and $match^o$ can be encoded.

Finally, we can make the following important observation. Obviously, any pattern matching synthesis problem has at least one trivial solution. This, due to the completeness of relational interleaving search [Kiselyov et al. 2005; Rozplokhas and Boulytchev 2019], means that the goal above *can not diverge* with no results. Actually it is rather easy to see that any pattern matching synthesis problem has *infinitely many* solutions: indeed, having just one it is always possible to "pump" it with superfluous "**otherwise**" clauses; thus, the goal above is *refutationally complete* [Byrd 2009; Rozplokhas and Boulytchev 2018]. These observations justify the totality of our synthesis approach. In Section 4 we show how we can make it provide optimal solution.

3.1 Constructing Relational Interpreters

In this section we address the implementation of relations $eval^o_{\mathcal{S}}$ and $match^o$. In principle, it amounts to accurate encoding of relations "$\Rightarrow$" and "$\Rightarrow_{\mathcal{S}}$" in miniKanren (in our case, OCanren). We, however, make use of a relational conversion [Lozov et al. 2017] tool, called noCanren, which automatically converts a subset of OCaml into OCanren. Thus, both interpreters are in fact implemented in OCaml and repeat corresponding inference rules almost literally in a familiar functional style. For example, functional implementation of a declarative semantics looks like follows:

```
let rec ⟨v; p⟩ =
  match (v, p) with
  | (_, Wildcard)  → true
  | (C^k v*, C^k p*) → list_all ⟨;⟩ (list_combine v* p*)
  | _              → false

let match^o v p* =
  let rec inner i p* =
    match p* with
    | []       → i
    | p :: p*  → if ⟨v; p⟩ then i else inner S(i) p*
  in inner 0 p*
```

We mixed here the concrete syntax of OCaml and mathematical notation, used in the definition of the relation in question; the actual implementation only a few lines of code longer. Note, we use here natural numbers in Peano form and custom list processing functions in order to apply relational conversion later.

Using relational conversion saves a lot of efforts as OCanren specifications tend to be much more verbose; in addition relational conversion implements some "best practices" in relational programming (for example, moves unifications forward in conjunctions and puts recursive calls last). Finally, it has to be taken into account that relational conversion of pattern matching introduces disequality constraints.

3.2 Dealing with a Complete Set of Samples

As we mentioned above, a complete set of samples plays an important role in our approach: it allows us to eliminate universal quantification over the set of all values. As we replace the universal quantifier with a finite conjunction with one conjunct per sample value reducing the size of the set is an important task. At the present time, however, we build an excessively large (worst case exponential of depth) number of samples. We discuss the issues with this choice in Section 5 and consider developing a more advanced approach as the main direction for improvement.

Our construction of a complete set of samples is based upon the following simple observations. We simultaneously define the *depth* measure for patterns and sequences of patterns as follows:

$$\begin{aligned} d\,(p_1, \ldots, p_k) &= max\,\{d\,(p_i)\} \\ d\,(\_) &= 0 \\ d\,(C^k\,p_1, \ldots p_k) &= 1 + d\,(p_1, \ldots, p_k) \end{aligned}$$

As a sequence of patterns is the single input in our synthesis approach we will call its depth *synthesis depth.*

Similarly, we define the depth of matching expressions

$$\begin{aligned} d_{\mathcal{M}}\,(\bullet) &= 1 \\ d_{\mathcal{M}}\,(m\,[i]) &= 1 + d_{\mathcal{M}}\,(m) \end{aligned}$$

and switch programs:

$$\begin{aligned} d_{\mathcal{S}}\,(\textbf{return}\ i) &= 0 \\ d_{\mathcal{S}}\,(\textbf{switch}\ m\ \textbf{of}\ C_1 \to s_1, \ldots, C_k \to s_k\ \textbf{otherwise}\ s) &= max\,\{d_{\mathcal{M}}\,(m),\, d_{\mathcal{S}}\,(s_i),\, d_{\mathcal{S}}\,(s)\} \end{aligned}$$

Informally, the depth of a switch program tells us how deep the program can look into a value.

From the definition of $\langle ; \rangle$ it immediately follows that a pattern p can only discriminate values up to its depth $d\,(p)$: changing a value at the depth greater or equal than $d\,(p)$ cannot affect the fact of matching/non matching. This means that we need only consider switch programs of depth no greater than the synthesis depth. But for these programs the set of all values with height no greater than the synthesis depth forms a complete set of samples. Indeed, if the height of a value less or equal to the synthesis depth, then this value is a member of complete set of samples and by definition the behavior of the synthesized program on this value is correct. Otherwise there exists some value s from the complete set of samples, such that given value can be obtained as an "extension" of s at the depth greater than the synthesis depth. Since neither declarative semantics nor switch programs can discriminate values at this depth, they behavior for a given value will coincide with the correct-by-definition behavior for s.

The implementation of complete set generation, again, is done using relational conversion. The enumeration of all terms up to a certain depth can be acquired from a function which calculates the depth of a term: indeed, converting it into a relation and then running with *fixed* depth and *free* term arguments delivers what we need. Thus, we add an extra conjunct which performs the enumeration of all values to the relational goal ($\star\star$), arriving at

$$depth^o\ v\ n \wedge \textbf{fresh}\ (i)\ \{match^o\ v\ p_1, \ldots, p_k\ i \wedge eval^o_{\mathcal{S}}\ v\ \textcircled{?}\ i\} \qquad (\star\star\star)$$

Here n is a precomputed synthesis depth in Peano form.

4 IMPLEMENTATION AND OPTIMIZATIONS

In this section we address two aspects of our solution: a number of optimizations which make the search more efficient, and the way it ends up with the optimal solution.

The relational goal in its final form, presented in the previous section, does not demonstrate good performance. Thus, we apply a number of techniques, some of which require extending the implementation of the search. Namely, we apply the following optimizations:

- We make use of type information to restrict the subset of constructors which may appear in a certain branch of program being synthesized.
- We implement *structural constraints* which allow us to restrict the shape of terms during the search, and utilize them to implement pruning.

In our formalization we do not make any use of types since as a rule type information does not affect matching. In addition, utilizing the properties of a concrete type system would make our approach too coupled with this particular type system, hampering its reusability for other languages. Nevertheless we may use a certain abstraction of type system which would deliver only that part of information which is essential for our approach to function. Currently, we calculate the type of any matching expression in the program being synthesized and from this type extract the subset of constructors which can appear when branching on this expression is performed. The number of these constructors restricts the number of branches which a corresponding **switch** expression can have. In our implementation we assume the constructor set ordered, and we consider only ordered branches, which restricts branching even more.

Our approach to finding an optimal solution in fact implements branch-and-bound strategy. The birds-eye view of our plan is as follows:

Patterns	Size constraint	Answers requested	Number of samples	1st answer time (ms)	Answers found	Total search time (ms)

`A`
`B`
`C`
	100	all	3	1	1	1

`true` **`false`**	100	all	2	<1	1	<1
(**`true`**, _) (_, **`true`**) (**`false`**, **`false`**)	100	all	4	6	2	10
(_, **`false`**, **`true`**) (**`false`**, **`true`**, _) (_, _, **`false`**) (_, _, **`true`**)	100	all	8	323	3	729
([], _) (_, []) (_ :: _, _ :: _)	100	10	4	5	1	6
(Succ _, Succ _) (Zero, _) (_, Zero)	1	all	4	53	2	108
(Nil, _) (_, Nil) (Nil2, _) (_, Nil2) (_ :: _, _ :: _)	1	10	9	643	2	3776
	10	10	9	95	2	540
	100	10	9	45	2	239

Fig. 6. The results of synthesis evaluation

- We construct a trivial solution, which gives us the first estimation.
- During the search we prune all partial solutions whose size exceeds current estimation. We can do this due to the top-down nature of partial solution construction.
- When we come up with a better solution we remember it and update current a estimate.

This strategy inevitably delivers us the optimal solution since there are only finitely many switch programs, shorter than trivial solution.

In order to implement this strategy we extended OCanren with a new primitive called *structural constraint*, which may fail on some terms depending on some criterion specified by an end-user. Structural constraints can be seen as a generalization of some known constraints like `absent`^{o} or `symbol`^{o} [Byrd et al. 2012] in existing miniKanren implementations, so they can be widely used in solving other problems as well. Note, we could

```
let rec run a s c =
  match a,s,c with
  | (_,_,Ldi i::_)  → 1
  | (_,_,Push::_)   → 2
  | (Int _,Val (Int _)::_,IOp _::_)  → 3
  | (Int _,_,Test (_,_)::c)  → 4
  | (Int _,_,Test (_,_)::c)  → 5
  | (_,_,Extend::_)  → 6
  | (_,_,Search _::_)  → 7
  | (_,_,Pushenv::_)  → 8
  | (_,Env e::s,Popenv::_)  → 9
  | (_,_,Mkclos cc::_)  → 10
  | (_,_,Mkclosrec _::_)  → 11
  | (Clo (_,_), Val _::_, Apply::_)  → 12
  | (_,(Code _::Env _::_),[])  → 13
  | (_,[],[])  → 14
```

Fig. 7. An example from a bytecode machine for PCF

implement other constraints we considered (on the depth of switch programs, on the type of scrutinee) as structural. However, our experience has shown that this leads to a less efficient implementation. Since these constraints are inherent to the problem, we kept them "hard coded".

5 EVALUATION

We performed an evaluation of the pattern matching synthesizer on a number of benchmarks. The majority of benchmarks were prepared manually; we didn't use any specific benchmark sets mentioned in literature [Scott and Ramsey 2000] yet. The 4th benchmark was taken from [Maranget 2008], we used it in Section 1 as the first example. The evaluation was performed on a desktop computer with Intel Core i7-4790K CPU @ 4.00GHz processor and 8GB of memory, OCanren was compiled with ocaml-4.07.1+fp+flambda. All benchmarks were executed in the native mode ten times, then average monotonic clock time was taken. The results of the evaluation are shown on Figure 6.

In the table the double bar separates input data from output. Inputs are: the patterns used for synthesis, the requested number of answers and the pruning factor. For example, in the first benchmark pruning factor equals 100 which means that structural constraint is checked every 100 unifications. Outputs are: the size of generated complete samples set, the running time before receiving the first answer, the total number of programs synthesized, the total search time.

Our approach currently does not work fast on a large benchmark. On Fig. 7 we cite an example extracted from a bytecode machine for PCF [Maranget 2008; Plotkin 1977]. For such complex examples (in terms of type definition complexity and number and size of patterns) both the size of the search space and the number of samples is too large for our approach to work so far.

6 CONCLUSION AND FUTURE WORK

We presented an approach for pattern matching implementation synthesis using relational programming. Currently, it demonstrates a good performance only on a very small problems. The performance can be improved by searching for new ways to prune the search space and by speeding up the implementation of relations and structural constraints. Also it could be interesting to integrate structural constraints more closely into OCanren's core. Discovering an optimal order of samples and reducing the complete set of samples is another direction for research.

The language of intermediate representation can be altered, too. It is interesting to add to an intermediate language so-called *exit nodes* described in [Le Fessant and Maranget 2001]. The straightforward implementation of them might require nominal unification, but we are not aware of any miniKanren implementation in which both disequality constraints and nominal unification [Byrd and Friedman 2007] coexist nicely.

At the moment we support only simple pattern matching without any extensions. It looks technically easy to extend our approach with non-linear and disjunctive patterns. It will, however, increase the search space and might require more optimizations.

REFERENCES

Lennart Augustsson. 1985. Compiling Pattern Matching. In *FPCA*.

Marianne Baudinet and David MacQueen. 1985. Tree pattern matching for ML. (1985).

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

William Byrd. [n. d.]. Relational Synthesis of Programs. ([n. d.]). http://webyrd.net/cl/cl.pdf

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph.D. Dissertation. Indiana University.

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

William E. Byrd and Daniel P. Friedman. 2007. αkanren: A Fresh Name in Nominal Logic Programming. In *Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming*. 79–90.

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, live and untagged: quine generation via relational interpreters (programming pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*. 8–29. https://doi.org/10.1145/2661103.2661105

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The reasoned schemer*. MIT Press.

Jacques Garrigue. 1998. Programming with polymorphic variants. In *In ACM Workshop on ML*.

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). (2005), 192–203. https://doi.org/10.1145/1086365.1086390

Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. (2016), 1–22. https://doi.org/10.4204/EPTCS.285.1

Alain Laville. 1991. Comparison of Priority Rules in Pattern Matching and Term Rewriting. *J. Symb. Comput.* 11 (1991), 321–347.

Fabrice Le Fessant and Luc Maranget. 2001. Optimizing Pattern Matching. *SIGPLAN Not.* 36, 10 (Oct. 2001), 26–37. https://doi.org/10.1145/507669.507641

Xavier Leroy, Damien Doligez, Jacques Garrigue Alain Frisch, Didier Rémy, and Jérôme Vouillon. 2020.

Petr Lozov, Andrei Vyatkin, and Dmitri Boulytchev. 2017. Typed Relational Conversion. In *TFP*.

Luc Maranget. 1992. Compiling lazy pattern matching. In *LFP '92*.

Luc Maranget. 1994. Two Techniques for Compiling Lazy Pattern Matching.

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, 35–46. https://doi.org/10.1145/1411304.1411311

Simon Marlow and Simon Peyton Jones. 2012. *The Glasgow Haskell Compiler* (the architecture of open source applications, volume 2 ed.). Lulu. https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/

FV McBride, DJT Morrison, and RM Pengelly. 1969. A symbol manipulation system. *Machine Intelligence* 5 (1969), 337–347.

Eugene Moiseenko. 2019. Constructive Negation for miniKanren. In *miniKanren and Relational Programming Workshop*.

Simon Peyton Jones. 1987. *The Implementation of Functional Programming Languages*. Prentice Hall. https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/

G. D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5 (1977), 223–255.

Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. Association for Computing Machinery, New York, NY, USA, Article 18, 13 pages. https://doi.org/10.1145/3236950.3236958

Dmitry Rozplokhas and Dmitry Boulytchev. 2019. Certified Semantics for miniKanren. In *miniKanren and Relational Programming Workshop*.

Kevin D Scott and Norman Ramsey. 2000. When Do Match-compilation Heuristics Matter?

Peter Sestoft. 1996. ML pattern match compilation and partial evaluation. In *Partial Evaluation*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 446–464.

Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible Pattern Matching via a Lightweight Language Extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/1291151.1291159

D. A. Turner. 2013. Some History of Functional Programming Languages. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

Philip Wadler. 1987. Compilation of pattern matching.