Relational Parsing: Grammar-Driven Program Synthesis

MAX SNYDER, Harvard College, USA

This paper presents a relational parser, which supports queries for the input, the output, or the grammar itself. The implementation described is an OMeta-style parser embedded in a language with a relational interpreter. Two example synthesis applications of the implementation are provided: numeric algebra and Boolean algebra.

Additional Key Words and Phrases: relational programming, parsing, miniKanren, OMeta, program synthesis

1 INTRODUCTION

Relational interpretation is a powerful tool for program synthesis. Since relational programming does not distinguish between input and output, a relational interpreter for language $\mathcal{L}$ is able to synthesize possible input programs $I \in \mathcal{L}$ for a given output value O, in addition to the converse (i.e. traditional interpretation). Either or both of the input program and output can be partially or completely constrained to specific concrete values.

Consider a grammar, or sub-language, $\mathcal{G}_{BA} \subset \mathcal{L}_0$ for valid expressions in Boolean algebra:

$$\begin{array}{ll} \texttt{Bit} = 0 \mid 1 & \text{false} \mid \text{true} \\ \texttt{Exp} = \texttt{Bit} & \\ \quad \mid (- \texttt{Exp}) & \text{logical not} \\ \quad \mid (\texttt{Exp} + \texttt{Exp}) & \text{logical or} \\ \quad \mid (\texttt{Exp} \times \texttt{Exp}) & \text{logical and} \end{array}$$

Valid inputs $I \in \mathcal{G}_{BA}$ include $0, 1, (-1), (1 \times 1), ((-0) \times (0 + 0)) \ldots$ with semantic values $0, 1, 0, 1, 0$. These inputs are syntactically valid in language $\mathcal{L}_0$, but their semantics in $\mathcal{L}_0$ may not agree with the expected semantics of Boolean algebra. For instance, the input $(1 + 1)$ could be interpreted as a list of three symbols, or perhaps the number two under numeric algebra. Even though language $\mathcal{L}_0$ might be capable of expressing and computing the logic of Boolean algebra, its relational interpreter has already hard-coded its semantics for each valid syntactic expression. It would be cumbersome to create a new interpreter in order to support program synthesis under new semantics.

In this paper, I address this issue using "relational parsing," a model that offers flexibility in the syntax of synthesized expressions with semantics that can still be expressed in the language of $\mathcal{L}$, effectively widening the horizon of program synthesis using the same interpreter.

I offer the following contributions:

- (Section 3) I formalize the concept of a "relational parser," a function in $\mathcal{L}$ that can parse inputs under a grammar of syntax paired with semantics expressed using functions in $\mathcal{L}$. I describe an implementation, The Relational Parser (TRP), which is embedded in Racket/miniKanren.
- (Section 4) I provide implementations for two example use cases of TRP:
  - Grammar $\mathcal{G}_{NA}$: a grammar for base-ten numeric algebra
  - Grammar $\mathcal{G}_{BA}$: (as described above)
- (Section 5) I discuss possible future work in the domain of relational parsing.

2 BACKGROUND

The Relational Parser relies on three conceptual frameworks and three corresponding implementations: parsing (OMeta), relational programming (miniKanren), and relational interpretation (Racket/staged-miniKanren). I now provide a background on the relevant content of each.

2.1 Parsing and OMeta

Parsing is the discipline of recognizing the syntax of a programming language. The simplest parser is a "recognizer," which returns true if a program is valid under its grammar and false otherwise. If a parser associates a semantic value with each syntactic construction, it becomes an "interpreter." If the semantic value of a program is itself another program, the parser becomes a "compiler."

OMeta [Warth and Piumarta 2007] is a language based on Parsing Expression Grammars (PEGs) [Ford 2004] for representing a grammar where semantic actions can be associated with each syntactic construction. Each rule specifies a syntax using combinations of: the empty pattern, atoms (e.g. numbers), non-terminals, sequencing, alternation, iteration, negation, and lists. Each syntactic construction can have an associated semantic action, a function that returns its semantic value based on bound variables, which are the recursively computed semantic values of sub-expressions.

2.2 Relational Programming and miniKanren

Relational programming is a paradigm that has no notion of input or output. Variables are related to one another via "unification," which enforces that two quantities have equal value. One variable can be "queried" to yield a stream of possible values that it could take on given a set of unification constraints. The final program is therefore not a function, but a "goal" that relates its input variables.

miniKanren [Byrd 2009] is one implementation embedded in Scheme. Goals are run as follows:

```
(run* (q) (== q 1))
```

This queries for all possible values of the query variable `q` for the goal `(== q 1)`. The goal unifies `q` with a concrete value, and the program consequently returns `(1)`.

Goals can yield multiple results, too.

```
(run* (q) (conde ((== q 1)) ((== q 2))))
```

This program, with a disjunction of two goals, returns `(1 2)`. For programs with many solutions, `run n` can be used instead, where `n` is the integer number of desired solutions.

staged-miniKanren [Amin et al. 2021] is a version of miniKanren that stages compilation to reduce interpretive overhead. Instead of `run` and `run*`, smK uses `run-staged` and `run-staged*`.

2.3 Relational Interpretation and Racket/staged-miniKanren

A relational interpreter is a relational program whose goal relates an input program with its value. Programs can be synthesized within input language $\mathcal{L}$ by querying for the input. [Byrd et al. 2012]

Racket/staged-miniKanren, a subset of Racket, is one such language $\mathcal{L}$ with a relational interpreter written in staged-miniKanren. The interpreter is a goal `evalo-staged` that relates an input program (in Racket/smK) and its output value. The following is an example program that generates quines, or input programs whose outputs are precisely the input programs:

```
(run-staged 1 (q) (evalo-staged q q))
```

3 THE RELATIONAL PARSER

3.1 Formalism

A relational parser is a program $P \in \mathcal{L}$ with these inputs and outputs:

- Input 1: Grammar $\mathcal{G}$
- Input 2: List of Input Tokens in
- Input 3: Name of Rule to Instantiate n_0
- Output 1: Semantic Value out
- Output 2: List of Remaining Tokens in_{rem}

The grammar is a map from rule name n_i to a list of choices $\mathcal{G}(n_i)$ that can instantiate the rule. Choices are prioritized, which means they are attempted in sequential order starting with $\mathcal{G}(n_i)_0$. Each choice $\mathcal{G}(n_i)_j$ is an expression under the following grammar:

`Choice` = `Token`	atom
| (n_k)	non-terminal rule n_k
| $(*$ `Choice`$)$	iteration for 0+
| $(!$ `Choice`$)$	negation
| $(:\ x$ `Choice`$)$	binding to x
| $(= \lambda\mu \ldots)$	semantic action under store μ
| (`Choice`)	list
| `Choice Choice`	sequencing

The inference rules for parsing choices are based on Core-OMeta [Warth 2009]. In the absence of semantic actions, choices have a "default" semantic value:

- Atom: the atom itself (a number or a symbol)
- Non-Terminal: the recursively computed semantic value of the input under rule n_k
- Iteration: list of the semantic values from each iteration
- Negation: `NONE`
- Binding: the bound value
- List: the list itself, a token of *in*
- Sequencing: the semantic value of the final choice in the sequence

If a semantic action is provided, the default semantic value is overridden by the result of the given lambda expression. These are functions in the host language $\mathcal{L}$ that take as input a store μ, an association list comprised of bound variables within each rule. If a binding expression is parsed, the store is updated for the given name with the semantic value. Stores are locally scoped at the rule level, so are not modified by nested non-terminals. For iteration and negation, the store is updated after each nested parse, even if the parse attempt does not succeed.

The input is "parsed in" upon success: in_{rem} is a suffix of *in*. For each instantiation in a choice sequencing, a subsequent prefix of *in* is parsed in. Negation and semantic actions do not parse in.

If a complete instantiation exists for choice $\mathcal{G}(n_i)_j$, the semantic value and the input yet to be parsed in are outputted. Otherwise, the next choice $\mathcal{G}(n_i)_{j+1}$ is tried. Once all choices are exhausted, with none successful, the semantic value `FALSE` and $in_{rem} = in$ are outputted.

3.2 Implementation

The Relational Parser extends Racket/smK with the `parse` function, which is accessible throughout code nested within a `the-relational-parser` call. Each rule in the grammar is provided a name, as well as the syntax and semantics for each constituent choice (1A, 1B, 2A, 2B . . .).

```
(run-staged 1 (q)
  (evalo-staged
    (the-relational-parser
      `(letrec
        (...) ; HELPER FUNCTIONS
          (parse
            (list ; GRAMMAR
              (list 'RULE-NAME-1 ; rule 1
                (list ... ; syntax 1A
                  (list '= (lambda (u) ...))) ; semantics 1A
                (list ... ; syntax 1B
                  (list '= (lambda (u) ...))) ; semantics 1B
                ...)
              (list 'RULE-NAME-2 ; rule 2
                (list ... ; syntax 2A
                  (list '= (lambda (u) ...))) ; semantics 2A
                (list ... ; syntax 2B
                  (list '= (lambda (u) ...))) ; semantics 2B
                ...)
              ...)
            '(...) 'RULE-NAME)) ; INPUT
    '(... ()))) ; OUTPUT
```

The implementation can be found at https://github.com/maxsnyder2000/TheRelationalParser [Snyder 2021].

4 EXAMPLES

The behavior of The Relational Parser is best understood through example grammars.

4.1 Numeric Algebra

This is an example recognizer grammar $\mathcal{G}_{NA}$ for numeric algebra (`test-algebra-1.scm`).

```
(run-staged 100 (q)
  (fresh (any-out)
    (evalo-staged
      (the-relational-parser
        `(parse
          '((Dig (0) (1) (2) (3) (4) (5) (6) (7) (8) (9))
            (Num (0) ((! 0) (Dig) (* (Dig))))
            (Exp
              ((Num))
              (((: x (Exp)) + (: y (Exp))))
              (((: x (Exp)) - (: y (Exp))))
              (((: x (Exp)) x (: y (Exp))))
              (((: x (Exp)) / (: y (Exp))))))
          ',q 'Exp))
      `(,any-out ()))))
```

Grammar $\mathcal{G}_{NA}$ has three rules:

- `Dig`. This rule has ten choices, each one a single token symbolizing a base ten digit.
- `Num`. This rule has two choices, either the digit zero, or a digit that is not zero followed by any number of digits. This ensures that zero is the only number that starts with zero (not 00123).
- `Exp`. This rule has five choices. The first choice accepts every input that the `Num` rule accepts. The remaining choices each parse a single list containing three elements: operand `x`, token (`+ - x /`), and operand `y`. `x` and `y` are both recursive instantiations of the `Exp` rule.

The query above yields 100 input programs under grammar $\mathcal{G}_{NA}$ and the rule `Exp` that parse with any semantic output and no remaining input `()`. (Note: input `()` with output `(#f ())` is a trivial solution, where `#f` is the semantic value of invalid inputs. The remaining 99 inputs are valid.)

This is the output of inputs, computed on a MacBook Pro in 103 seconds of elapsed real time:

```
((0) () ((0 + 0)) ((0 - 0)) ((0 x 0)) ((0 / 0)) (1) (2) ((0 - (0 + 0))) (1 0)
↪  ((0 - (0 - 0))) ((0 + (0 + 0))) (((0 + 0) + 0)) (((0 + 0) - 0)) ((0 - (0 x
↪  0))) (((0 + 0) x 0)) ((0 + (0 - 0))) (((0 + 0) / 0)) ((0 - (0 / 0))) (((0 -
↪  0) - 0)) ((0 / (0 + 0))) (((0 - 0) + 0)) ((0 x (0 + 0))) ((0 / (0 - 0)))
↪  (((0 - 0) x 0)) ((0 x (0 - 0))) (3) (2 0) (1 1) ((0 / (0 x 0))) (1 0 0) ((0
↪  x (0 x 0))) (((0 - 0) / 0)) ((0 / (0 / 0))) ((0 + (0 x 0))) ((0 x (0 / 0)))
↪  (((0 + 0) - (0 + 0))) (((0 + 0) - (0 - 0))) ((0 + (0 / 0))) ((0 - 1)) (((0 +
↪  0) - (0 x 0))) (4) (3 0) (2 1) (1 2) (1 0 1) (2 0 0) (1 1 0) (((0 + 0) - (0
↪  / 0))) (((0 + 0) / (0 + 0))) (1 0 0 0) (((0 + 0) / (0 - 0))) (((0 + 0) x (0
↪  + 0))) (((0 + 0) x (0 - 0))) (((0 + 0) / (0 x 0))) (((0 x 0) - 0)) (((0 - 0)
↪  - (0 + 0))) (((0 x 0) + 0)) (((0 + 0) x (0 x 0))) (((0 + 0) / (0 / 0))) (((0
↪  x 0) x 0)) (((0 - 0) - (0 - 0))) (((0 x 0) / 0)) (((0 + 0) x (0 / 0))) ((0 /
↪  1)) ((0 x 1)) (((0 - 0) - (0 x 0))) ((0 - 2)) (5) ((0 - (0 - (0 + 0)))) ((0
↪  - 1 0)) (4 0) (3 1) (2 2) (1 3) ((0 - (0 - (0 - 0)))) (((0 - 0) / (0 + 0)))
↪  (1 0 2) (2 0 1) (1 1 1) (3 0 0) (2 1 0) (1 2 0) (((0 - 0) - (0 / 0))) (((0 -
↪  0) / (0 - 0))) ((0 - (0 + (0 + 0)))) (((0 - 0) x (0 + 0))) (1 0 0 1) (1 0 1
↪  0) (2 0 0 0) (1 1 0 0) ((0 - ((0 + 0) + 0))) (1 0 0 0 0) ((0 - ((0 + 0) -
↪  0))) (((0 - 0) x (0 - 0))) ((0 - (0 - (0 x 0)))) (((0 - 0) / (0 x 0))) (((0
↪  + 0) - 1)) ((0 - ((0 + 0) x 0))) ((0 - (0 + (0 - 0)))))
```

An example interpreter grammar (i.e. with semantics) is available on the GitHub repo as well (`test-algebra-2.scm`). It is currently much slower than the first example due to semantic overhead.

4.2 Boolean Algebra

This is an example interpreter grammar $\mathcal{G}_{BA}$ for Boolean algebra (`test-boolean-algebra.scm`). Since computation for Boolean algebra is simpler than numeric algebra, program synthesis is faster.

```
(run-staged 100 (q)
  (evalo-staged
    (the-relational-parser
      `(letrec
        ([neg (lambda (x) (if (equal? x 1) 0 1))]
         [add (lambda (x y) (if (or  (equal? x 1) (equal? y 1)) 1 0))]
         [mul (lambda (x y) (if (and (equal? x 1) (equal? y 1)) 1 0))])
        (parse
          (list
            (list 'Bit '(0) '(1))
            (list 'Exp
              (list '(Bit))
              (list '- '(: x (Exp))
                (list '= (lambda (u) (neg (get u 'x)))))
              (list '((: x (Exp)) + (: y (Exp)))
                (list '= (lambda (u) (add (get u 'x) (get u 'y)))))
              (list '((: x (Exp)) x (: y (Exp)))
                (list '= (lambda (u) (mul (get u 'x) (get u 'y)))))))
          ',q 'Exp)))
  '(1 ()))
```

Grammar $\mathcal{G}_{BA}$ has two rules:

- `Bit`. This rule has two choices, token `0` or token `1`.
- `Exp`. This rule has four choices.
  - The first choice accepts every input that the `Bit` rule accepts.
  - The second choice parses the token `-` then expression `x`. The semantic value of this choice is the negation of the bound value for `x` in the store, computed using the `neg` helper function.
  - The last two choices each parse a single list containing three elements: operand `x`, token `(+ x)`, and operand `y`. `x` and `y` are both recursive instantiations of the `Exp` rule. The two semantic actions compute Boolean "or" and "and" using the `add` and `mul` helper functions.

The query above yields 100 input programs under grammar $\mathcal{G}_{BA}$ and the rule `Exp` that parse with semantic value `1` and no remaining input `()`. This is an instance of semantic program synthesis.

This is the output of inputs, computed on a MacBook Pro in 34 seconds of elapsed real time:

```
((1) (- 0) (- - 1) ((1 x 1)) (- (0 x 0)) ((0 + 1)) (- - - 0) (- (0 x 1)) ((1 x -
↪  0)) ((1 + 0)) (- (1 x 0)) (- (0 + 0)) ((1 + 1)) (- (0 x - 0)) (- (0 x - 1))
↪  ((0 + - 0)) (- - - - 1) (- (0 x (0 x 0))) (- (0 x - - 0)) ((1 x - - 1)) (-
↪  (1 x - 1)) (- (0 x (0 x 1))) (- - (1 x 1)) (- - - (0 x 0)) ((- 0 x 1)) (- -
↪  (0 + 1)) (- (0 x - - 1)) ((1 + - 0)) (- - - - - 0) (- - - (0 x 1)) ((- 0 +
↪  0)) ((1 x (1 x 1))) ((- 0 x - 0)) (- (0 x (1 x 0))) (- (1 x (0 x 0))) (- (0
↪  x (0 + 0))) ((1 x - (0 x 0))) ((1 x (0 + 1))) ((- 0 + 1)) (- (1 x - - 0)) (-
↪  (0 x (0 x - 0))) (- (0 x (1 x 1))) (- - (1 x - 0)) (- - (1 + 0)) ((1 x - - -
↪  0)) (- (0 + - 1)) (- (1 x (0 x 1))) (- (0 x - (0 x 0))) (- - - (1 x 0)) ((1
↪  x - (0 x 1))) (- - - (0 + 0)) (- (0 x (0 + 1))) (- (0 x (0 x - 1))) (- - (1
↪  + 1)) ((1 + - 1)) (- (0 x - - - 0)) (- - - (0 x - 0)) (- (0 x - (0 x 1)))
↪  ((1 x (1 x - 0))) ((1 x (1 + 0))) ((- 0 x - - 1)) (- (- 0 x 0)) (- (1 x (1 x
↪  0))) ((1 x - (1 x 0))) (- (1 x (0 + 0))) (- (0 x - - - 1)) (- - - (0 x - 1))
↪  ((1 x - (0 + 0))) (- (0 x (1 x - 0))) (- (0 x (1 + 0))) ((1 x (1 + 1))) (-
↪  (1 x (0 x - 0))) (- - (0 + - 0)) (- (0 x (0 x (0 x 0)))) ((1 x - (0 x - 0)))
↪  (- (0 x - (1 x 0))) ((- 0 x (1 x 1))) (- (0 x - (0 + 0))) (- (- 0 x - 1)) ((-
↪  0 x - (0 x 0))) (- - - - - - 1) (- (0 x (0 x - - 0))) (- (0 x (1 + 1))) (-
↪  (0 x (1 x - 1))) ((- 0 x (0 + 1))) (- (1 x (0 x - 1))) (- (0 x - (0 x - 0)))
↪  (- (0 x (0 x (0 x 1)))) ((1 x - (0 x - 1))) ((- 0 x - - - 0)) (- (0 x - (1 x
↪  1))) (- - - (0 x (0 x 0))) ((- 0 x - (0 x 1))) (- (0 x - - (0 x 0))) (- (- 0
↪  x (0 x 0))) ((- 1 + 1)) ((1 x (0 + - 0))) (- (0 x - (0 + 1))) (- (0 x (0 x -
↪  - 1))) (- - - (0 x - - 0)))
```

5 FUTURE WORK

Due to the open ended nature of relational querying, there are many avenues for future exploration of relational parsing as a tool for program synthesis. This paper discusses four areas of future study:

(1) **Improve the interpreter grammar for numeric algebra.**
The current interpreter grammar makes use of Peano numerals for arithmetic calculation; for instance, the number 100 is represented by a sequence of one hundred 1s `(1 1 1 . . . 1)`. Switching to a digital representation, where numbers are represented by a sequence of digits (e.g. binary), would improve runtime and enable further exploration of algebraic synthesis. The performance can be compared against relational algebra implementations in miniKanren.

(2) **Write a grammar for compilation to a lowered language.**
This paper provides example recognizer and interpreter grammars, but no compiler grammars. By constructing a grammar that parses higher-level language $\mathcal{L}_H$ (e.g. Racket, Python, OMeta)

into lower-level language $\mathcal{L}_L$ (e.g. x86, LLVM, C), one could synthesize programs in $\mathcal{L}_H$ that compile to a program in $\mathcal{L}_L$. To support this more complex use case, it may be useful to first add support to the parser for other features from OMeta beyond the Core-OMeta semantics, including the higher-order rules and the object-oriented behavior like state and inheritance.

(3) **Query for a grammar given a set of input and output pairs.**
The examples presented so far have all been oriented towards synthesizing valid programs. However, grammar synthesis is also possible given a relational parser by providing a set of valid input and output pairs and querying for the grammar. Based on preliminary testing, this appears to take a long time when the structure of the grammar is completely unrestricted. However, under additional constraints, perhaps the grammar may be more easily learned. These constraints could be syntactic by providing some subset of well formed expressions. These constraints could be semantic by implementing helper functions that are known to be useful in advance, enabling the relational search to more quickly discover a correct solution. Also, careful choice of input/output pairs may be key to prevent under-/over-generalization.

(4) **Create *parseKanren,* a language optimized for relational parsing.**
As reflected by the example outputs, the ordering of the miniKanren search has a strong effect on the quality and speed of the results. Consequently, a new implementation of miniKanren designed specifically for relational parsing, omitting extraneous features, could improve both optimality and efficiency. The parser might be better embedded within this language as a miniKanren relation instead of a Racket/smK function. Racket/smK itself could be expanded with more features, perhaps with support for the quasiquote ‘ to simplify the grammar syntax.

6 CONCLUSION

In this paper, I introduced the notion of "relational parsing," provided two examples of the TRP implementation in use, and discussed four areas covering a range of future work in this new domain.

ACKNOWLEDGMENTS

Much thanks to Nada Amin, Will Byrd, and Liam McInroy for inspiration and advice along the way.

REFERENCES

Nada Amin, Michael Ballantyne, and William E. Byrd. 2021. Staged Relational Interpreters: Running with Holes, Faster. (2021). https://namin.seas.harvard.edu/files/namin/files/staged-mk.pdf

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University. https://github.com/webyrd/dissertation-single-spaced/blob/master/thesis.pdf

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2012. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2012), 1–26. https://dl.acm.org/doi/pdf/10.1145/3110252

Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *POPL'04.* ACM, New York City, 1–12. https://bford.info/pub/lang/peg.pdf

Max Snyder. 2021. *The Relational Parser.* GitHub. https://github.com/maxsnyder2000/TheRelationalParser

Alessandro Warth. 2009. *Experimenting with Programming Languages.* Ph.D. Dissertation. University of California, Los Angeles. https://web.cs.ucla.edu/~todd/theses/warth_dissertation.pdf

Alessandro Warth and Ian Piumarta. 2007. OMeta: an Object-Oriented Language for Pattern Matching. In *DLS'07.* ACM, New York City, 1–9. http://tinlizzie.org/~awarth/papers/dls07.pdf