Binding-Time Analysis for MINIKANREN

Ekaterina Verbitskaia
JetBrains Research
Saint Petersburg, Russia
kajigor@gmail.com

Irina Artemeva
ITMO University
Saint Petersburg, Russia
irina-pluralia@rambler.ru

Daniil Berezun
JetBrains Research
Saint Petersburg, Russia
daniil.berezun@jetbrains.com

MINIKANREN[1] is a family of domain-specific languages for pure *relational* programming the core of which can be implemented in a few lines of the host language [2]. A program in MINIKANREN can be run in different directions: given some of its arguments, it computes all the possible values of the rest of the arguments. When the relation is run with only the last argument unknown, we say it is run in the *forward direction*, otherwise we call it running in the *backward direction*. A single relational specification creates a multitude of directions, each solving a distinctive problem.

The search employed in MINIKANREN is complete, so all possible answers will be computed eventually, albeit it may take a long time. In reality, the running time depends on the direction and is highly unpredictable. This is where the promise of MINIKANREN falls short: one has to write the relational specification with the specific direction in mind for it to be efficient. There are several attempts to tackle this problem in an automatic fashion: one way is to use a *divergence test* to stop execution of definitely diverging computations [5], the other is to employ *specialization* [4], while the one we have been working on recently is to converse a relational specification into a function which runs fast in the given direction.

Given a relational specification equipped with a certain direction, the functional conversion constructs a function in which unifications are mapped either into a pattern matching or are used in let-bindings to compute the intermediate results. Disjunctions give rise to branches in pattern matchings within a function definition, while conjunctions mean sequential computations. The nondeterministic computations are modeled with lists. The most nontrivial part of the functional conversion is to determine the order in which to bind each variable within the function body.

In compilation and specialization to determine at what time the values of the variables can be computed, *binding-time analysis*, or bta, is used. Typically, bta divides all computations into two stages: *compile-time*, or *static*, and *execution time*, or *dynamic*. One approach to bta is to explicitly annotate program variables and computations by equipping them with their binding times. In logic programming there is no predefined execution direction — no order in which conjunctions should be considered — which complicates the problem of identifying variables binding times. It has been shown [4] that online conjunctive partial deduction may improve program running time in a given direction. We are investigating the applicability of offline specialization methods in the context of functional conversion. Offline approaches are known [3] to be safe, helpful in the design and implementation of partial evaluation projects, and are more efficient than the online ones, which is crucial in the context of logic programming. The existing binding-time analyses for pure PROLOG [1] and MERCURY [7] only distinguish between static and dynamic variables, which does not help to work out the order. Natural numbers serve as annotations in [6] and are better suited to our problem.

The binding-time analysis for MINIKANREN should annotate a given goal as well as determine suitable directions for all relation calls involved in the result computation. We start by marking some free variables within the goal with 0 which means they are the *input* or *static* variables, while all others are annotated with $undef$. This is the way we specify a direction for the goal.

[1] MINIKANREN language web site: http://minikanren.org

Submitted to:
TEASE-LP 2020

We assume that all goals are in *canonical normal form*: a goal is a disjunction of conjunctions of either relation calls or unifications; all free variables are brought up into scope by the $\underline{fresh}$ operator on the top level. Relation calls and constructors can only have variables as their arguments. Unification should have a single variable on the left-hand side with the exception of unifying two constructors of arity 0. We leave unifications of the zero-arity constructors out, since they can either be evaluated to a failure or to a success without extending the substitution which can be determined statically. Any core MINIKANREN program can be converted to this form.

$$Goal = \underline{fresh}\ V_1 \ldots V_n \bigvee \bigwedge (\underline{call}\ R^k\ V_1 \ldots V_k \mid V \equiv T)$$

The algorithm is to execute the following steps until a fixpoint is reached. Since disjuncts do not influence each other, we treat every disjunct in isolation. A conjunct is either a unification or a relation call. At each step, a unification suitable for annotation is selected. If there are no unifications to annotate, then we select a relation call. Whenever a new annotation is assigned to a variable, all other uses of the same variable within the conjunction get the same annotation.

Unifications are annotated if there is enough information. There are two possible cases described below. We write annotations in superscripts; $t[x_0, \ldots, x_k]$ denotes a term with free variables $x_0, \ldots, x_k$.

- The unification of a variable annotated with $undef$ with a term in which all free variables are annotated with numbers: $x^{undef} \equiv t[y_0^{i_0}, \ldots, y_k^{i_k}]$. In this case we annotate x with $1 + max\{i_0, \ldots, i_k\}$.
- The unification of a variable annotated with a number with a term, some variables of which are annotated with $undef$: $x^n \equiv t[y_0^{i_0}, \ldots, y_k^{i_k}]$. Here all $undef$ variables are annotated with $1+n$.

If a unification does not conform to any of these cases, it should not be annotated at the current step. Relation calls are only annotated when all unifications have been considered. A relation call in which all variables are annotated with $undef$ as well as those annotated with only positive numbers do not need to be considered. Thus relation calls in which only some of the variables are annotated with positive numbers are to be annotated.

We say that two calls to the same relation have a *consistent* direction if their free variables which are annotated with numbers are ordered in the same way. More formally: a relation call $R^n\ v_1^{i_1} \ldots\ v_k^{i_k}$ has a direction consistent with $R^n\ u_1^{j_1} \ldots\ u_k^{j_k}$, if all variables with numerical annotations are ordered in the same way: $v_{l_1}^{i_{l_1}} \leq \cdots \leq v_{l_m}^{i_{l_m}}$ and $u_{l_1}^{j_{l_1}} \leq \cdots \leq u_{l_m}^{j_{l_m}}$. For example, relation calls $R^3\ x^{undef}\ y^0\ z^0$, $R^3\ x^{undef}\ y^1\ z^2$ and $R^3 x^{undef} y^{undef} z^3$ all have consistent directions, while the direction of $R^3\ x^1\ y^2\ z^{undef}$ is not consistent with the direction of $R^3\ x^{undef}\ y^1\ z^2$.

If the current conjunct has a consistent direction with some relation call annotated before, then there is no need to explore this direction again. We use the previous call to annotate the current one. Only a relation call with the direction which has not been encountered before should be annotated. It is done by considering its body: variables annotated within the call keep their annotations in the body, which is annotated by the described algorithm. All annotated relation definitions are stored while algorithm runs, so they can be reused if necessary.

We presented a binding-time analysis algorithm for MINIKANREN that determines the order in which variables are bound. The algorithm terminates since there are finitely many different relation calls to consider, and each relation call has finitely many possible annotations. Unfortunately, some variables can remain $undef$ after the algorithm is finished. This happens when there are relation calls which influence the direction of each other. Currently we choose the leftmost relation call to annotate first; while another, less efficient but more fair, solution may be to consider all possible combinations of directions. In our experience, this problem rarely arises in the context of relational interpreters, but we should further investigate the class of such programs and develop a better strategy of dealing with them.

References

[1] Stephen-John Craig, John P Gallagher, Michael Leuschel & Kim S Henriksen (2004): *Fully automatic binding-time analysis for Prolog*. In: *International Symposium on Logic-Based Program Synthesis and Transformation*, Springer, pp. 53–68.

[2] Jason Hemann & Daniel P. Friedman (2013): *uKanren: A Minimal Functional Core for Relational Programming*.

[3] Neil D Jones, Carsten K Gomard & Peter Sestoft (1993): *Partial evaluation and automatic program generation*. Peter Sestoft.

[4] Petr Lozov, Ekaterina Verbitskaia & Dmitry Boulytchev (2019): *Relational Interpreters for Search Problems*. In: *Relational Programming Workshop*, p. 43.

[5] Dmitri Rozplokhas & 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*, pp. 1–13.

[6] Peter Thiemann (1997): *A Unified Framework for Binding-Time Analysis*. In: *TAPSOFT*.

[7] Wim Vanhoof, Maurice Bruynooghe & Michael Leuschel (2004): *Binding-time analysis for Mercury*. In: *Program Development in Computational Logic*, Springer, pp. 189–232.