Binding-Time Analysis for MINIKANREN

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

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

Dmitri Boulytchev
SPbSU
Saint Petersburg, Russia
dboulytchev@gmail.com

We present a binding-time analysis algorithm for MINIKANREN. It is capable to determine the order in which names within a program are bound and can be used to facilitate specialization and as a step of translation into a functional language.

1 Introduction

MINIKANREN is a family of domain-specific languages for pure *relational* programming. The core of MINIKANREN is very small and can be implemented in few lines of the host language [2]. A program in MINIKANREN consists of a *goal* and a set of *relation definitions* with fixed arity. A goal is either a *disjunction* of two goals, a *conjunction* of two goals, *invocation of a relation* or a *unification of two terms*. A term is either a *variable* or a *constructor* of arity n applied to n terms. All free variables within a goal are brought up into scope with a $\underline{fresh}$ operator. The abstract syntax of the language is presented below. We denote a goal with G, a term with T, a variable with V, a name of the relation with arity n with R^n and a name of the constructor with arity n with C^n.

$$G := G \vee G \mid G \wedge G \mid \underline{call}\ R^n\ [T_1, \ldots, T_n] \mid T \equiv T \mid \underline{fresh}\ V\ G$$
$$T := V \mid \underline{cons}\ C^n\ [T_1, \ldots, T_n]$$

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*. The search employed in MINIKANREN is complete, so all possible answers will be computed eventually, albeit it may take a long time. In reality, running time is often unpredictable and depends on the direction. There are different approaches to tackling this problem: using a divergence test to stop execution of definitely diverging computations [4], employing specialization [3], even rewriting a program by hand introducing some redundancy[1].

It has been shown in [3] that online conjunctive partial deduction is capable of improving running time of a program, but we believe that offline specialization may provide better results. Offline specializers employ a static analysis before the specialization step. The most prominent of them is *binding-time analysis*. It is used to determine which data is available statically, and which — dynamically. Having this annotations, specialization algorithm can ignore dynamic data and only symbolically execute a program with respect to the static data. This usually leads to more precise and powerful specialization.

The authors are also working on translation of the MINIKANREN code into a functional programming language. In translation, it is crucial to know the order in which variables of a program are bound. We see binding-time analysis as a suitable technique for determining this order.

In the next section we present a binding-time analysis algorithm for MINIKANREN.

[1] Example of a program which can be rewritten by hand to improve running time in both direction: `https://github.com/JetBrains-Research/OCanren/blob/master/regression/test002sort.ml`

Submitted to:
TEASE-LP 2020

2 Binding-Time Analysis for MINIKANREN

A binding-time analysis starts by annotating a given goal. Some free variables of the goal are marked with 0 which denotes they are the *input* or *static* variables, others are annotated with $undef$. The aim is to determine the order in which the $undef$ variables are bound within the goal as well as to annotate all relation definitions called. Existing binding-time analysis for pure PROLOG [1] and MERCURY [6] only distinguish between static and dynamic variables, which does not help to work out the order. Natural numbers can serve as annotations [5] and are better suited to our problem. Thus we use natural numbers, 0 and $undef$ for annotations.

We assume that all goals are in *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:

$$G = \underline{fresh}\ V_1 \ldots V_n \bigvee \bigwedge (\underline{call}\ R^k\ T_1 \ldots T_k \mid T_1 \equiv T_2)$$

The algorithm is to execute the following steps until a fix-point 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 several 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 an unannotated variable 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 an annotated variable with a term, in which some variables are not annotated: $x^n \equiv t[y_0^{i_0}, \ldots, y_k^{i_k}]$. Here all unannotated variables of the term are annotated with $1+n$.
- The unification of two constructors with the same name and the same number of arguments: $\underline{cons}\ C^k\ [t_0^{i_0}, \ldots, t_k^{i_k}] \equiv \underline{cons}\ C^k\ [s_0^{j_0}, \ldots, s_k^{j_k}]$. This is equivalent to the conjunction of the unifications $t_l^{i_l} \equiv s_l^{j_l}$ and is treated accordingly.
- Two cases symmetrical to the first two, in which a term is unified with a variable.

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 have not been annotated (are marked with $undef$) as well as those already annotated with some 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 relation calls have a *consistent* direction if they call a relation with the same name and

- We have already annotated the relation with the same name in the consistent direction. Then all $undef$ variables get the annotation $1+n$, where n is the maximum annotation of the annotated variables.
- We have not encountered a consistent invocation. Then we try to annotate the body of the corresponding relation.

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

[4] 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.

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

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