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 [1]. 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.

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

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 [3], employing specialization [2], even rewriting a program by hand introducing some redundancy[1].

It has been shown in [2] 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 some static analysis before the specialization step. The most prominent of them is *binding-time analysis*. This analysis is used to determine which data is available statically, and which — dynamically. Having this annotations, specialization algorithm can ignore all dynamic data and only symbolically execute a program with respect to the static data. This usually leads to the 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.

[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 program in miniKanren is a goal accompanied with a set of relation definitions.

The goal of bta is to determine the order in which variables are getting bound within a given goal and all relations called within it.

We will consider a goal in MINIKANREN to be a disjunction of conjunctions of relations invocations or unifications. It is always possible to transform any MINIKANREN goal in this normal form.

We annotate each variable in a goal with either an $undef$ annotation or a natural number.

We start with a goal in which input parameters are annotated with 0, while all other variables are annotated with $undef$.

Then we run a the following procedure until a fix-point is reached.

Every disjunct is treated seperately from other disjuncts, since they do not influence each other.

Within a conjunction, there can be either a unification or an invocation. Whenever a new annotation is assigned to some variable use within a conjunction is propagated onto other uses of the same variable.

First we annotate unifications with some partial annotations. When there are no unifications which can be annotated, we annotate invocations of the current relation. Then — all other invocations. Any of the later annotations can create new partially annotated unifications, so they have to be treated again, in the order they have appeared.

Annotating unification works like this. There are several possible cases to consider. The annotations are written in superscripts, $t[x_0, \dots, x_k]$ denotes a term with free variables $x_0, \dots, x_k$.

- The unification of an unannotated variable with a term: $x^{undef} \equiv t[y_0^{i_0}, \dots, y_k^{i_k}]$, where all free variables in a term are annotated with numbers. Then we annotate x with a $1 + max\{i_0, \dots, i_k\}$.
- The unification of an annotated variable with a term, in which some variables are not annotated: $x^n \equiv t[y_0^{i_0}, \dots, y_k^{i_k}]$. Here all unannotated variables within the term are annotated with $1 + n$.
- The unification of two constructors with the same name and the same number of arguments: $\underline{cons}\ Name\ [t_0^{i_0}, \dots, t_k^{i_k}] \equiv \underline{cons}\ Name\ [s_0^{j_0}, \dots, s_k^{j_k}]$. 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.
- If a unification does not conform to any of these cases, do not annotate it, leaving it for later.

When annotating invocations, there are possible several cases.

- All variables used within an invocation are annotated with $undef$. Annotation in this case is impossible, proceed to the next case.
- All variables are annotated. No annotation is needed – proceed to the next case.
- We have already annotated the relation with the same 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] Jason Hemann & Daniel P. Friedman (2013): *uKanren: A Minimal Functional Core for Relational Programming*.

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

[3] 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.