μKanren: A Minimal Functional Core for Relational Programming

Jason Hemann Daniel P. Friedman
Indiana University
{jhemann,dfried}@cs.indiana.edu

Abstract

This paper presents μKanren, a minimalist language in the miniKanren family of relational (logic) programming languages. Its implementation comprises fewer than 40 lines of Scheme. We motivate the need for a minimalist miniKanren language, and iteratively develop a complete search strategy. Finally, we demonstrate that through sufficient user-level features one regains much of the expressiveness of other miniKanren languages. In our opinion its brevity and simple semantics make μKanren uniquely elegant.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages, Constraint and logic languages

Keywords miniKanren, relational programming, logic programming, Scheme

1. Introduction

miniKanren is the principal member of an eponymous family of relational (logic) programming languages. Many of its critical design decisions are a reaction to those of Prolog and other well-known 5th-generation languages. One of the differences is that, while a typical Prolog implementation might be thousands of lines of C code, a miniKanren language is usually implemented in somewhere under 1000 lines. Though there are miniKanren languages of varied sizes and feature sets (see `http://miniKanren.org`), the original published implementation was 265 lines of Scheme code. In those few lines, it provides an expressiveness comparable to that of an implementation of a pure subset of Prolog.

We argue, though, that deeply buried within that 265-line miniKanren implementation is a small, beautiful, relational programming language seeking to get out. We believe μKanren is that language. By minimizing the operators to those strictly necessary to do relational programming and placing much of the interface directly under the user's control, we have further simplified the implementation and illuminated the role and interrelationships of the remaining components. By making the implementation entirely functional and devoid of macros, heretofore opaque sections of the system's internals are made manifest. What is more, by re-adjudicating what functions are properly the purview of the end user, we develop an implementation that weighs in at 39 lines of Scheme. The presentation of the language and its features follows. The complete implementation of μKanren is found in the appendix.

2. The μKanren Language

Herein, we briefly describe the syntax of μKanren programs, with a focus on those areas in which μKanren differs from earlier miniKanren languages. Readers intimately familiar with miniKanren programming may lightly peruse this section; readers seeking a thorough introduction to miniKanren programming are directed to Byrd [3] and Friedman et. al [4], from which the present discussion is adapted.

A μKanren program proceeds through the application of a *goal* to a *state*. Goals are often understood by analogy to predicates. Whereas the application of a predicate to an element of its domain can be either true or false, a goal pursued in a given state can either *succeed* or *fail*. A goal's success may result in a sequence of (enlarged) states, which we term a *stream* [12]. We use functions to simulate relations. An arbitrary n-ary relation is viewed as an $(n-1)$-ary partial function, mapping tuples of domain elements into a linearized submultiset of elements of the codomain over which the initial relation holds. A given collection of goals may be satisfied by zero or more states. The result of a μKanren program is a stream of satisfying states. The stream may be finite or infinite, as there may be finite or infinitely many satisfying states.

A program's resulting stream thus depends on the goals that comprise that program. In μKanren there are four primitive goal constructors: $\equiv$, `call/fresh`, `disj`, and `conj`. The $\equiv$ goal constructor is the primary workhorse of the system; goals constructed from $\equiv$ succeed when the two arguments *unify* [1]. The success of goals built from $\equiv$ may cause the state to grow. Unlike the implementation of $\equiv$ detailed in Friedman et. al [4], that presented here does not prohibit circularities in the substitution.

The `call/fresh` goal constructor creates a fresh (new) logic variable. `call/fresh`'s sole argument is a unary func-