A Relational Exploration of Computation

A Relational Exploration of Computation

William E. Byrd

For my H211 students:
Indiana University, Fall 2010 & 2011, and Team pw0ni3.

Learning with always trumps learning from.

—Woodie Flowers

Contents

Preface

When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.

—Alan J. Perlis

I want a programming language in which I need only say what I wish done.

I am not the only one. Researchers and industry programmers are increasingly interested in *declarative programming*, in which programmers specify *what* to do, rather than *how* to do it. Perhaps the most common examples of declarative programming languages are database query languages, such as SQL, XQuery, and Datalog. The equational reasoning provided by the lazy functional language Haskell can also be seen as a form of declarative programming.

For decades logic programming has been heralded as the ultimate in declarative programming, with the promise of writing programs as mathematical relations that do not distinguish between "input" and "output" arguments. This is illustrated by the traditional "hello world!" example from logic programming, list concatenation. In the best-known logic programming language—Prolog—the call `append([1,2,3],[4,5],X)` produces the answer `X = [1,2,3,4,5]`. More impressively, the call `append(X,Y,[1,2,3,4,5])` can produce all pairs of lists `X` and `Y` such that `X` ++ `Y = [1,2,3,4,5]`.

Alas, most interesting Prolog programs are *not* written in a relational style. For reasons of efficiency and expressivity, Prolog programmers tend to use impure "extra-logical" features of the language, such as the cut (`!`), used to prune the search tree, and `assert/retract`, used to modify the global database of facts. These features make reasoning about logic programs more difficult, and can even result in unsound behavior. Even when these features are used correctly, they inhibit the ability to treat programs as relations.

In many ways, the current practice of logic programming is reminiscent of how functional programming was practiced before Haskell. Lisp and ML programmers understood the benefits of coding in a pure functional style; in a pinch, however, they could resort to variable mutation or other side-effects. As with Prolog, these impure operators were used for efficiency and expressivity. Haskell's laziness required a more pure approach, however, which in turn led to the adoption of monads for encapsulating effects. By committing to a pure functional style, Haskell advanced the theory and practice of functional programming, even for strict languages. *Constraints spur creativity.*

For the past decade my colleagues and I have been developing *miniKanren* (`http://www.minikanren.org/`), a logic programming language specifically designed for programming in a relational style. During that time I have developed relational interpreters, term reducers, type inferencers, and theorem provers, all of which are capable of program synthesis, or of otherwise running "backwards."

Over the past three years miniKanren has gained a significant following in the Clojure community in the form of the popular `core.logic` library, which started as a Clojure port of the code in my dissertation. `core.logic` is now being used in both industry and academia. miniKanren is also growing in popularity in the Racket community, in the form of the `cKanren` library. miniKanren has been ported from Scheme to many other languages, including Python, Ruby, JavaScript, Haskell, and Scala.

Chapter 1

Introduction

Bibliography