Relational Programming in miniKanren: Techniques, Applications, and Implementations

William E. Byrd

Submitted to the faculty of the
University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University

August, 2009

Accepted by the faculty of the University Graduate School, Indiana University, in partial fulfillment of the requirements for the degree Doctor of Philosophy

Daniel P. Friedman, Ph.D.
(Principal Advisor)

Amr Sabry, Ph.D.

Christopher T. Haynes, Ph.D.

Lawrence S. Moss, Ph.D.

Bloomington, Indiana
May 7, 2009

This document is a re-formatted version of the original, archival dissertation:
`http://gradworks.umi.com/3380156.pdf`

This document, along with all files necessary to typeset it, can be found at:
`https://github.com/webyrd/dissertation-single-spaced`

Typeset by the author in XƎLaTeX.
November 5, 2014 version.

For my parents.

Hold, p'ease!
—Brian Byrd

Acknowledgments

I first want to acknowledge my mother—without her I wouldn't be here writing this dissertation. My father taught me a love of science and nature, which led to my interest in computers and programming languages—without him I wouldn't be here, either. I dedicate my dissertation to them, with love and admiration.

I also thank my brother Brian, my sister Mary, and their spouses, Claudia Díaz-Byrd and Donald Stevens. I can't imagine better siblings or in-laws.

Renzhong Chen has been part of the Byrd family for twenty years. I thank him for his friendship, and for the incredible tour of China that was a highlight of my time in grad school. I am also grateful for the hospitality of Renzhong's wife, Lea Li.

Dan Friedman has been my teacher, mentor, boss, hacking buddy, coauthor, and friend during my six years at Indiana University. It was Dan who introduced me to logic programming, and to his language that eventually evolved into miniKanren.

I cannot thank Dan without also thanking his wife Mary and the rest of the Friedmans, who have been a second family to me in Bloomington. I am especially grateful to Sa`{nd|mm}`i Friedman for hours of amusement.

My committee members, Dan, Amr, Chris, and Larry, have been unfailingly cheerful, patient, and supportive. This is fortunate, since their intellects might otherwise be intimidating. I am especially gratified that all of these scholars are deeply committed to the art of teaching. Thank you all!

Oleg Kiselyov taught me an unbelievable amount about logic programming, especially the benefits of purity, and the dangers of impure operators like **cond**^{a} and **cond**^{u}. It was Oleg who had the incredibly clever idea of implementing a relational arithmetic system inspired by hardware half-adders and full-adders (see Chapter 6). Oleg has also had a tremendous influence on the development and evolution of miniKanren and its predecessor, Kanren.

Chung-chieh (Ken) Shan has also greatly influenced the evolution, and especially the implementation, of miniKanren. Much of the brevity and elegance of the core miniKanren implementation in Chapter 3 is due to Ken, who designed the critical **case**$^\infty$ macro.

Some of the ideas, implementation code, and example programs in this dissertation were first presented, often in a slightly different form, in *The Reasoned Schemer* (Friedman et al. 2005). Chapter 6 is based on chapters seven and eight of *The Reasoned Schemer* and, to a lesser extent, Kiselyov et al. (2008). Chapters 9 and 11 are adapted from Byrd and Friedman (2007). Chapter 10 is adapted from Near et al. (2008). A paper containing the contents of Chapter 14 and the nestable engines code in Appendix D will be presented at Mitch Wand's Festschrift. Many thanks to all of my coauthors. Please see the acknowledgments sections of these papers for additional credits.

The first tabling implementation for miniKanren was designed by the author and Ramana Kumar, and inspired by the Dynamic Reordering of Alternatives (DRA) approach to tabling (Guo and Gupta 2009, 2001). Ramana implemented the design, with debugging assistance from the author. The tabling implementation presented in Chapter 12 is a slightly modified version of Ramana's second, and improved, tabling implementation. Most importantly, the new implementation is based on streams rather than success and failure continuations, which means answers are produced in the same order as in the core miniKanren implementation of Chapter 3.

The nominal unifier using triangular substitutions in section 11.4 is due to Joe Near. Ramana Kumar has implemented a faster but very different triangular unifier.

As described in section 4.1, Abdulaziz Ghuloum, David Bender, and Lindsey Kuper have explored which purely functional data structures are best for representing triangular substitutions.

The **pmatch** pattern-matching macro in Appendix B was written by Oleg Kiselyov. The **match**^{e} and λ^e pattern-matching macros in Appendix C were originally designed by the author and implemented by Ramana Kumar with the help of Dan Friedman. Andy Keep, Michael Adams, and Lindsey Kuper worked with us to implement an optimized version of **match**^{e} and λ^e, which will be presented at the 2009 Scheme Workshop.

Visits to Bloomington from Christian Urban, Matt Lakin, and Gopal Gupta greatly aided my research. I also benefited from the 3rd International Compulog/ALP Summer School on Logic Programming and Computational Logic at New Mexico State University, organized by Enrico Pontelli, Inna Pivkina, and Son Cao Tran. I enjoyed many stimulating conversations with visiting scholars Juliana Vizzotto, Katja Grace, Dave Herman, and Sourav Mukherjee.

Indiana University's PL Wonks lecture series, organized by Roshan James, has been most stimulating. I thank Roshan, Michael Adams, Andy Keep, Jeremiah Willcock, Ron Garcia, Jeremy Siek, Steve Ganz, Larisse Voufo, and all the other Wonks for many interesting conversations about programming languages. The PL Wonks also benefited from special visits by Jeffrey Siskind and Robby Findler. I look forward to a relaxing conversation with Olivier Danvy that does not require either of us to use our first-responder skills.

For the past eleven semesters I have had the great pleasure of being the associate instructor for Indiana University's undergraduate (C311) and graduate (B521) introductory programming languages courses, under the enthusiastic leadership of Dan Friedman. The material in this dissertation was greatly improved by the comments and corrections of our students. I am grateful to you all, especially those former students who have conducted summer research with us: Dave Bender, Jordan Brown, Adam Foltzer, Adam Hinz, Andy Keep, Jiho Kim, Ramana Kumar, Lindsey Kuper, Micah Linnemeier, and Joe Near.

Special thanks goes to former C311 student Jeremiah Penery, who discovered and corrected a subtle error in the definition of log^o from the first printing of *The Reasoned Schemer*.

Dan and I have had the great fortune to work with exceptional graduate and undergraduate associate instructors: David Mack, Alex Platte, Kyle Blocher, Joe Near, Ramana Kumar, and Lindsey Kuper. Their hard work has made C311 and B521 such a success. Joe and Ramana also read the final draft of this dissertation, and provided many insightful comments and corrections—thank you!

I learned a great deal from teaching the honors section of IU's introductory programming course (C211) under supreme Scheme hacker Kent Dybvig. Several years later I had the great pleasure of teaching C211 with another Scheme master, Aziz Ghuloum, as my associate instructor. Every teacher should be so lucky.

I thank Olin Shivers for writing an exemplary dissertation, the structure and organization of which I shamelessly ripped off. Once again I relied on Dorai Sitaram's Scheme typesetting program, SLaTeX.

Whenever I was floundering in grad school Mitch Wand seemed to magically appear, pulling me aside to see how I was doing, and offering much appreciated advice and encouragement. Although I did not always follow his advice (to my detriment, I am sure), I will always be grateful for his support.

In addition to sharing his teaching and programming expertise, Kent Dybvig also offered invaluable advice about navigating the pitfalls of grad school. I didn't heed all of Kent's advice, though I learned to pay special attention to anything following the catchphrase, "You'll be committing academic suicide."

Lucy Battersby, Rob Henderson, and the rest of the Indiana University Computer Science Department staff provided expert help and an outstanding work environment.

The friendly staff of the east-side Bloomington Quiznos, Sunny Bal, Kae Lunde, Caitlyn Muncy, Alisha Findley, Tylla Carlisle, and Meagan Perry, kept me rolling in veggie subs and "om-nom-nom-nom"-worthy cookies.

Caitlin Coar, Cortney Packett, and Alisha Stout, formerly of Cold Stone Creamery, supplied me with delicious and nourishing PB&C milkshakes.

I wrote most of this dissertation at the east-side Starbucks in Bloomington, where the generous, hilarious, and slightly-unhinged baristas provided a tasty setting for extended writing sessions. Many thanks to:

- Amanda Buck for all the fish stories;
- Andrea Jerabek for always giving me a hard time;
- Ben Canary for inventing the infamous and delectable "Christmas Cookie";

- Brian Ibison for adoring Olivia Munn;
- Christina Liwski for never giving me a hard time;
- Ciera Brannon for not believing in cells (*No cells, no mercy!*);
- Eric “Big Eric” Martin for pointing out that UFO over Starbucks;
- Erin Dobias for being a smart chick;
- Gabby Baehl for living up to her first name;
- Megan Traxinger-James for putting up with Eric and Brian;
- and Phil Wood for his Twitter-powered news reports.

I also thank former baristas Jessica Fugate and Shannon Pilrose for hanging out with me when I should have been writing.

I’ve been fortunate to have made many close friends in Bloomington. In particular, Aziz Ghuloum, Larisse Voufo, Ron Garcia, Suzanna Crage, Andy Keep, Lindsey Kuper, Lindsay and Ahmed Hamed, and Anne and Mike Faber helped keep me sane while I wrote this dissertation.

Marc Muher visited Bloomington for Dig Dug *battle royale.* Leslie Cuevas helped keep Jennifer Fitzgerald in line, while Ada Brunstein offered end game encouragement. As always, my childhood friends Mike, Daryl, and Bobby helped relieve the tension with the occasional game.

It has been my honor to know the Miller family for many years, and am grateful for their friendship.

Cisco Nochera, former director of Camp Greentop, has been my friend and mentor for almost two decades.

During the spring of 2006 I visited Chile for a week, along with my brother Brian and my sister-in-law Claudia. Claudia’s parents, Hugo Díaz and Nancy Zuñiga de Díaz, warmly welcomed us to their home in the beautiful countryside on the outskirts of Casablanca, Chile. I thank them for their never-ending hospitality and goodwill.

I have had many incredible teachers in my life, but a few stand out. My absolutely amazing 11^{th} grade Spanish teacher, Tom Rahauser, taught me to never fear Señor Subjunctivo. Richard Saenz’s class on special relatively completely blew my mind. Tom Anastasio taught me LISP, cleverly disguised as C. Alan Sherman prepared me for graduate school. Dan Friedman taught me that you don’t understand your code until it fits on a 3x5 card.

Abstract

The promise of logic programming is that programs can be written *relationally*, without distinguishing between input and output arguments. Relational programs are remarkably flexible—for example, a relational type-inferencer also performs type checking and type inhabitation, while a relational theorem prover generates theorems as well as proofs and can even be used as a simple proof assistant.

Unfortunately, writing relational programs is difficult, and requires many interesting and unusual tools and techniques. For example, a relational interpreter for a subset of Scheme might use nominal unification to support variable binding and scope, Constraint Logic Programming over Finite Domains (CLP(FD)) to implement relational arithmetic, and tabling to improve termination behavior.

In this dissertation I present *miniKanren*, a family of languages specifically designed for relational programming, and which supports a variety of relational idioms and techniques. I show how miniKanren can be used to write interesting relational programs, including an extremely flexible lean tableau theorem prover and a novel constraint-free binary arithmetic system with strong termination guarantees. I also present interesting and practical techniques used to implement miniKanren, including a nominal unifier that uses triangular rather than idempotent substitutions and a novel "walk"-based algorithm for variable lookup in triangular substitutions.

The result of this research is a family of languages that supports a variety of relational idioms and techniques, making it feasible and useful to write interesting programs as relations.

Contents

Chapter 1

Introduction

1.1 My Thesis

A beginning is a very delicate time.
—Princess Irulan

miniKanren supports a variety of relational idioms and techniques, making it feasible and useful to write interesting programs as relations.

The promise of logic programming is that programs can be written *relationally*, without distinguishing between input and output arguments. Each relation produces meaningful answers, even when all of its arguments are unbound logic variables. Relational programs are remarkably flexible—for example, a relational type inferencer can also perform type checking and type inhabitation. Similarly, a relational theorem prover can also be used as a proof checker, proof generator, theorem generator, and even as a primitive proof assistant.

Unfortunately, writing remarkably flexible relational programs is remarkably difficult, and requires a variety of unusual and advanced tools and techniques. For example, a relational interpreter for a subset of Scheme might use nominal unification to support variable binding and scope, Constraint Logic Programming over Finite Domains (CLP(FD)) to implement relational arithmetic, and tabling to improve termination behavior.

This dissertation presents *miniKanren*, a family of languages specifically designed for relational programming, and which supports a variety of relational idioms and techniques. We show how miniKanren can be used to write interesting relational programs, including an extremely flexible lean tableau theorem prover and a novel constraint-free binary arithmetic system with strong termination guarantees. We also present interesting and practical techniques used to implement miniKanren, including a nominal unifier that uses triangular rather than idempotent substitutions and a novel "walk"-based algorithm for variable lookup in triangular substitutions.

Chapter 2 presents the core miniKanren language, which we then extend with disequality constraints (Chapter 7), nominal logic (Chapter 9), tabling (Chapter 12), and expression-level divergence avoidance using ferns (Chapter 14). We provide implementations of all of these language extensions in Chapters 3, 4, 8, 11, 13, and 15. Together, these chapters establish the first half of my thesis: miniKanren supports a variety of relational idioms and techniques.

To illustrate the use of these techniques, we present two non-trivial miniKanren applications. The constraint-free relational arithmetic system of Chapter 6 and the theorem prover of Chapter 10 establish the second half of my thesis: it is feasible and useful to write interesting programs as relations in miniKanren, using these idioms and techniques.

1.2 Structure of this Dissertation

With the exception of two early chapters (Chapters 2 and 5), each technical chapter in this dissertation is divided into one of three categories: techniques, applications, or implementations[1]. *Technique chapters* describe language features and idioms for writing relations, such as disequality constraints (Chapter 7) and nominal logic (Chapter 9). *Application chapters* demonstrate how to write interesting, non-trivial relations in miniKanren; these applications demonstrate the use of many of the language forms and idioms presented in the technique chapters. *Implementation chapters* show how to implement the language extensions presented in the technique chapters.

At a higher level, the dissertation is divided into six parts, which are organized by theme:

- Part I presents the core miniKanren language, which we will extend in the latter parts of the dissertation. Chapter 2 introduces the core language, along with a few simple examples, while Chapter 3 presents the implementation of the core language. These two chapters are especially important, since they form the foundation for the advanced techniques and implementations that follow. In Chapter 4 we optimize the *walk* algorithm presented in Chapter 3, which is the heart of miniKanren's unifier. Chapter 5 attempts to categorize the many ways miniKanren programs can diverge, and describes techniques that can be used to avoid each type of divergence. Avoiding divergence while maintaining declarativeness is what makes relational programming so fascinating, yet so challenging. Chapter 6 presents a non-trivial application of core miniKanren: a constraint-free arithmetic system with strong termination guarantees.

- Part II extends core miniKanren with disequality constraints, which allow us to express that two terms are different, and can never be unified. Disequality

[1] Hence the title of this dissertation: *Relational Programming in miniKanren: Techniques, Applications, and Implementations.*

constraints express a very limited form of negation, and can be seen as a very simple kind of constraint logic programming. Chapter 7 describes disequality constraints from the perspective of the user, while Chapter 8 shows how we can use unification in a clever way to simply and efficiently implement the constraints. We give special attention to *constraint reification*—the process of displaying constraints in a human-friendly manner.

- Part III extends core miniKanren with operators for expressing nominal logic; we call the resulting language αKanren. Nominal logic allows us to easily express notions of scope and binding, which is useful when writing declarative interpreters, type inferencers, and many other relations that deal with variables. Chapter 9 introduces nominal logic, explains αKanren's new language constructs, and provides a few simple example programs. Chapter 10 presents a non-trivial application of αKanren: a relational theorem prover. In Chapter 11 we present our implementation of αKanren, including two different implementations of nominal unification.

- Part IV adds tabling to our implementation of core miniKanren. Tabling is a form of memoization: the answers produced by a tabled relation are "remembered" (that is, stored in a table), so that subsequent calls to the relation can avoid recomputing the answers. Tabling allows our programs to run more efficiently in many cases; more importantly, many programs that would otherwise diverge terminate when using tabling. Chapter 12 introduces the notion of tabling, and explains which programs benefit from tabling. Chapter 13 presents our streams-based implementation of tabling, which demonstrates the advantage of embedding miniKanren in a language with higher-order functions.

- Part V presents a bottom-avoiding data structure called a *fern*, and shows how ferns can be used to avoid expression-level divergence. Chapter 14 introduces the fern data structure and implements a simple, miniKanren-like language using ferns. Chapter 15 presents our embedding of ferns in Scheme.

- Part VI provides context and conclusions for the work in this dissertation. Chapter 16 describes related work, while Chapter 17 proposes future research. We offer our final conclusions in Chapter 18.

The dissertation also includes four appendices. Appendix A contains several generic helper functions that could be part of any standard Scheme library. Appendix B describes and defines **pmatch**, a simple pattern matching macro for Scheme programs. Appendix C describes and defines $\textbf{match}^e$ and λ^e, pattern matching macros for writing concise miniKanren relations. Appendix D contains our implementation of nestable engines, which are used in our embedding of ferns.

1.3 Relational Programming

Relational programming is a discipline of logic programming in which every goal is written as a "pure" relation. Each relation produces meaningful answers, even when all of its arguments are unbound logic variables. For example, Chapter 6 presents $plus^o$, which performs addition over natural numbers. ($plus^o$ $\mathbf{1}$ $\mathbf{2}$ $\mathbf{3}$)[2] succeeds, since $1 + 2 = 3$—that is, the triple $(1, 2, 3)$ is in the ternary addition relation. We can use $plus^o$ to add two numbers: ($plus^o$ $\mathbf{1}$ $\mathbf{2}$ z) associates the logic variable z with 3. We can also subtract numbers using $plus^o$: ($plus^o$ $\mathbf{1}$ y $\mathbf{3}$) associates y with 2, since $3 - 1 = 2$. We can even call $plus^o$ with only logic variables: ($plus^o$ x y z) produces an infinite number of answers in which the natural numbers associated with x, y, and z satisfy $x{+}y{=}z$. For example, one such answer associates x with 3, y with 4, and z with 7.

To write relational goals, programmers must avoid a variety of powerful logic programming constructs, such as Prolog's cut (`!`), `var/1`, and `copy_term/2` operators. These operators inhibit relational programming, since their proper use is dependent upon the groundness or non-groundness of terms[3]. Programmers who wish to write relations must avoid these constructs, and instead use language features compatible with the relational paradigm.

A critical aspect of relational programming is the desire for relations to terminate whenever possible. Writing a goal without mode restrictions is not very interesting if the goal diverges when passed one or more fresh variables. In particular, we desire the *finite failure* property for our goals—if a goal is asked to produce an answer, yet no answer exists, that goal should fail in a finite amount of time. Although Gödel and Turing showed that it is impossible to guarantee termination for all goals we might wish to write, the use of clever data encoding, nominal unification, tabling, and the derivation of bounds on the maximum size of terms allows a careful miniKanren programmer to write surprisingly sophisticated programs that exhibit finite failure.

Our emphasis on both pure relations and finite failure leads to different design choices than those of more established logic programming languages such as Prolog (Intl. Organization for Standardization 1995, 2000), Mercury (Somogyi et al. 1995), and Curry (Hanus et al. 1995; Hanus 2006). For example, unlike Prolog, miniKanren uses a complete (interleaving) search strategy by default. Unlike Mercury, miniKanren uses full unification, required to implement goals that take only fresh logic variables as their arguments[4]. And our desire for termination prevents us from

[2] **1**, **2**, and **3** are shorthand for the little-endian binary lists representing the numbers 1, 2, and 3—see Chapter 6 for details.

[3] A term is *ground* if it does not contain unassociated logic variables.

[4] Mercury is statically typed, and requires programmers to specify "mode annotations" (Apt and Marchiori 1994) indicating whether each argument to a goal is an "input" (that is, fully ground) or an "output" (that is, an unassociated logic variable). Programmers also specify whether each goal can produce one, finitely many, or infinitely many answers. Given all this information, the Mercury

adapting Curry's residuation[5].

1.4 miniKanren

This dissertation presents miniKanren, a language designed for relational programming, along with various language extensions that add expressive power without sacrificing the ability to write relations.

miniKanren is implemented as an embedding in Scheme, using only a handful of special forms and functions. The concise and purely functional implementation of the core operators makes the language easy to extend. miniKanren programmers have access to all of Scheme, including higher-order functions, first-class continuations, and Scheme's unique and powerful hygienic macro system. Having access to Scheme's features makes it easy for implementers to extend miniKanren; for example, from a single figure explaining XSB-style OLDT resolution we were able to design and implement a tabling system for miniKanren in under a week.

This thesis presents complete Scheme implementations of core miniKanren and its extensions, including two versions of nominal unification, a simple constraint system, a streams-based tabling system, and a minimal implementation of a miniKanren-like language using the bottom-avoiding fern data-structure. Our implementation of core miniKanren is purely functional, and is designed to be easily modifiable, encouraging readers to experiment with and extend miniKanren.

1.5 Typographical Conventions

The code in this dissertation uses the following typographic conventions. Lexical variables are in *italic*, forms are in **boldface**, and quoted symbols are in sans serif. Quotes, quasiquotes, and unquotes are suppressed, and quoted or quasiquoted lists appear with bold parentheses—for example **()** and **(**x **.** *x***)** are entered as `'()` and `` `(x . ,x) ``, respectively. By our convention, names of relations end with a superscript *o*—for example *subst*^{o}, which is entered as `substo`. Relational operators do not follow this convention: ≡ (entered as `==`), **cond**^{e} (entered as `conde`), and **exist**.

compiler can generate multiple specialized functions that perform the work of a single goal. For example, a ternary goal that expresses addition (similar to the *plus*^{o} function described above) might be compiled into separate functions that perform addition or subtraction; at runtime, the appropriate function will be called depending on which arguments are ground. In fact, compiled Mercury programs do not use logic variables or unification, and are therefore extremely efficient. Unfortunately, this lack of unification means it is not possible to write Mercury goals that take only "output" variables.

[5] *Residuation* (Hanus 1995) suspends certain operations on non-ground terms, until those terms become ground. For example, we could use residuation to express addition using Scheme's built-in + procedure. If we try to add x and 5, and x is an unassociated logic variable, we suspend the addition, and instead try running another goal. Hopefully this goal will associate x with a number; when that happens, we can perform the addition. However, if x never becomes ground, we will be unable to perform the addition, and we will never produce an answer.

Chapter 7 introduces the relational operator $\neq$ (entered as `=/=`), while Chapter 9 introduces **fresh**, $\#$ (entered as `hash`), and the term constructor $\bowtie$ (entered as `tie`). Similarly, (**run**$^{\mathbf{5}}$ (q) *body*) and (**run*** (q) *body*) are entered as `(run 5 (q) body)` and `(run* (q) body)`, respectively.

λ is entered as `lambda`. λ^{e} from Appendix C is entered as `lambdae`. The arithmetic relations $\leqslant l^{o}$ and $\leqslant^{o}$ from Chapter 6 are entered as `<=lo` and `<=o`, respectively. *occurs*$^{\surd}$ from Chapter 3 is entered as `occurs-check`.

Part I

Core miniKanren

Chapter 2

Introduction to Core miniKanren

This chapter introduces the core miniKanren language, provides several short example programs, and shows how to translate a simple Scheme function into a miniKanren relation.

This chapter is organized as follows. Section 2.1 introduces the core miniKanren language. In section 2.2 we show how to translate the standard Scheme *append* function into a miniKanren relation. Section 2.3 describes several "impure" operators that, while not part of the pure miniKanren core language, are useful when trying to model Prolog programs.

2.1 Core miniKanren

miniKanren extends Scheme with three operators: $\equiv$, **cond**^{e}, and **exist**. There is also **run**, which serves as an interface between Scheme and miniKanren, and whose value is a list.

exist, which syntactically looks like λ, introduces new variables into its scope; $\equiv$ unifies two values. Thus

(**exist** (x y z) ($\equiv$ x z) ($\equiv$ 3 y))

would associate x with z and y with 3. This, however, is not a legal miniKanren program—we must wrap a **run** around the entire expression.

(**run**$^{\mathbf{1}}$ (q) (**exist** (x y z) ($\equiv$ x z) ($\equiv$ 3 y))) $\Rightarrow$ **(**$\__0$**)**

The value returned is a list containing the single value $\__0$; we say that $\__0$ is the *reified value* of the fresh variable q. q also remains fresh in

(**run**$^{\mathbf{1}}$ (q) (**exist** (x y) ($\equiv$ x q) ($\equiv$ 3 y))) $\Rightarrow$ **(**$\__0$**)**

We can get back other values, of course.

```
(run^1 (y)            (run^1 (q)          (run^1 (y)
  (exist (x z)          (exist (x z)        (exist (x y)
    (≡ x z)               (≡ x z)             (≡ 4 x)
    (≡ 3 y)))             (≡ 3 z)             (≡ x y))
                          (≡ q x)))         (≡ 3 y))
```

Each of these examples returns **(3)**; in the rightmost example, the *y* introduced by **exist** is different from the *y* introduced by **run** because the variables are lexically scoped. **run** can also return the empty list, indicating that there are no values.

```
(run^1 (x) (≡ 4 3)) ⇒ ()
```

We use **cond**^{e} to get several values—syntactically, **cond**^{e} looks like **cond** but without ⇒ or **else**. For example,

```
(run^2 (q)
  (exist (x y z)
    (cond^e
      ((≡ (x y z x) q))
      ((≡ (z y x z) q))))) ⇒
((_0 _1 _2 _0) (_0 _1 _2 _0))
```

Although the two **cond**^{e}-clauses are different, the values returned are identical. This is because distinct reified fresh variables are assigned distinct numbers, increasing from left to right—the numbering starts over again from zero within each value, which is why the reified value of *x* is $\_{0}$ in the first value but $\_{2}$ in the second value.

Here is a simpler example using **cond**^{e}.

```
(run^5 (q)
  (exist (x y z)
    (cond^e
      ((≡ a x) (≡ 1 y) (≡ d z))
      ((≡ 2 y) (≡ b x) (≡ e z))
      ((≡ f z) (≡ c x) (≡ 3 y)))
    (≡ (x y z) q))) ⇒
((a 1 d) (b 2 e) (c 3 f))
```

The superscript 5 denotes the maximum length of the resultant list. If the superscript $*$ is used, then there is no maximum imposed. This can easily lead to infinite loops:

```
(run^* (q)
  (let loop ()
    (cond^e
      ((≡ #f q))
      ((≡ #t q))
      ((loop)))))
```

Had the $*$ been replaced by a non-negative integer n, then a list of n alternating #f's and #t's would be returned. The **cond**^{e} succeeds while associating q with #f, which accounts for the first value. When getting the second value, the second **cond**^{e}-clause is tried, and the association made between q and #f is forgotten—we say that q has been *refreshed*. In the third **cond**^{e}-clause, q is refreshed once again.

We now look at several interesting examples that rely on *any*^{o}.

```
(define any^o
  (λ (g)
    (cond^e
      (g)
      ((any^o g)))))
```

any^o tries g an unbounded number of times. Here is our first example using *any*^{o}.

```
(run^* (q)
  (cond^e
    ((any^o (≡ #f q)))
    ((≡ #t q))))
```

This example does not terminate, because the call to *any*^{o} succeeds an unbounded number of times. If $*$ is replaced by 5, then instead we get (#t #f #f #f #f). (The user should not be concerned with the order in which values are returned.)

Now consider

```
(run^10 (q)
  (any^o
    (cond^e
      ((≡ 1 q))
      ((≡ 2 q))
      ((≡ 3 q))))) ⇒
```

(1 2 3 1 2 3 1 2 3 1)

Here the values 1, 2, and 3 are interleaved; our use of *any*^{o} ensures that this sequence will be repeated indefinitely.

Here is *always*^{o},

```
(define always^o (any^o (≡ #f #f)))
```

along with two **run** expressions that use it.

```
                                (run^5 (x)
  (run^1 (x)                      (cond^e
    (≡ #t x)                        ((≡ #t x))
    always^o                        ((≡ #f x)))
    (≡ #f x))                     always^o
                                  (≡ #f x))
```

The left-hand expression diverges—this is because *always*^{o} succeeds an unbounded number of times, and because (≡ #f x) fails each of those times.

The right-hand expression returns a list of five #f's. This is because both **cond**^{e}-clauses are tried, and both succeed. However, only the second **cond**^{e}-clause contributes to the values returned. Nothing changes if we swap the two **cond**^{e}-clauses. If we change the last expression to (≡ #t *x*), we instead get a list of five #t's.

Even if some **cond**^{e}-clauses loop indefinitely, other **cond**^{e}-clauses can contribute to the values returned by a **run** expression. For example,

```
(run^3 (q)
  (let ((never^o (any^o (≡ #f #t))))
    (cond^e
      ((≡ 1 q))
      (never^o)
      ((cond^e
         ((≡ 2 q))
         (never^o)
         ((≡ 3 q)))))))
```

returns (1 2 3); replacing **run**^{3} with **run**^{4} causes divergence, however, since there are only three values, and since *never*^{o} loops indefinitely.

2.2 Translating Scheme Code to miniKanren

In this section we translate the standard Scheme function *append* to the equivalent miniKanren relation, *append*^{o}. *append* takes two lists as arguments, and returns the appended list.

(*append* (a b c) (d e)) ⇒ (a b c d e)

Here is the definition of *append*.

```
(define append
  (λ (l s)
    (cond
      ((null? l) s)
      (else (cons (car l) (append (cdr l) s))))))
```

Rather than translate the Scheme definition directly to miniKanren, we will massage the Scheme code to make it closer in spirit to a miniKanren relation. Only after we have performed several Scheme-to-Scheme transformations will we translate to miniKanren[1].

First we replace the always-true **else** test with an explicit *pair?* test, making the **cond** clauses *non-overlapping*[2].

[1] This approach differs from that of (Friedman et al. 2005), which translates Scheme functions directly to miniKanren.

[2] The concept of non-overlapping clauses is revisited in section 7.3.

```
(define append
  (λ (l s)
    (cond
      ((null? l) s)
      ((pair? l) (cons (car l) (append (cdr l) s))))))
```

Next we replace **cond** with the **pmatch** pattern-matching macro from Appendix B. The use of pattern matching is close in spirit to unification, and lets us easily translate the code to use **match**^{e} or λ^e from Appendix C.

```
(define append
  (λ (l s)
    (pmatch (l s)
      ((() s) s)
      (((a . d) s)
       (cons a (append d s))))))
```

We then perform an *unnesting* step reminiscent of the Continuation-Passing Style (CPS) transformation[3] (see, for example, Friedman and Wand (2008)): we unnest any nested calls, introducing **let**-bound variables where necessary[4].

```
(define append
  (λ (l s)
    (pmatch (l s)
      ((() s) s)
      (((a . d) s)
       (let ((res (append d s)))
         (cons a res))))))
```

After unnesting, we are ready to translate the Scheme function into a miniKanren relation. We add a superscript *o* to the name, to indicate the new function is a relation. We add an "output" argument[5] and change **pmatch** to **match**^{e}. We add the output argument to the list of values being matched against by **match**^{e}, and the individual patterns. Any value that would have previously been returned must now be unified with the *out* argument, either explicitly using ≡ or implicitly using pattern matching. We also change the **let** to **exist** introducing a "temporary" logic variable.

[3] More correctly, the unnested program is similar to one in A-Normal Form (ANF) (Flanagan et al. 1993).

[4] Unlike in the CPS transformation we must unnest *every* call, even those guaranteed to terminate. For example, unnesting (*cons* (*cons* 1 2) 3) results in (**let** ((*tmp* (*cons* 1 2))) (*cons* *tmp* 3)).

[5] When translating a Scheme predicate to a miniKanren relation we do not add an "output" argument. This is because success or failure of a call to the relation is equivalent to the Scheme predicate returning `#t` or `#f`, respectively.

```
(define append^o
  (λ (l s out)
    (match^e (l s out)
      ((() s s))
      (((a . d) s out)
       (exist (res)
         (append^o d s res)
         (≡ (cons a res) out))))))
```

Since we are matching against all the arguments of *append*^{o}, we can use λ^e rather than **match**^{e}. Also, we may wish to replace (*cons a res*) with (*a* . *res*) to reflect our use of unification as pattern matching.

```
(define append^o
  (λ^e (l s out)
    ((() s s))
    (((a . d) s out)
     (exist (res)
       (append^o d s res)
       (≡ (a . res) out)))))
```

If we do not wish to use the **match**^{e} or λ^e pattern matching macros, we can rewrite *append*^{o} in core miniKanren.

```
(define append^o
  (λ (l s out)
    (cond^e
      ((≡ () l) (≡ s out))
      ((exist (a d)
         (≡ (a . d) l)
         (exist (res)
           (append^o d s res)
           (≡ (a . res) out)))))))
```

Of course we can use the *append*^{o} relation to append two lists.

```
(run* (q) (append^o (a b c) (d e) q)) ⇒ ((a b c d e))
```

But we can also find all pairs of lists that, when appended, produce (a b c d e).

```
(run^6 (q)
  (exist (l s)
    (append^o l s (a b c d e))
    (≡ (l s) q))) ⇒

((() (a b c d e))
 ((a) (b c d e))
 ((a b) (c d e))
 ((a b c) (d e))
 ((a b c d) (e))
 ((a b c d e) ()))
```

Unfortunately, replacing **run**$^{\mathbf{6}}$ with **run**$^{\mathbf{7}}$ results in divergence, for reasons explained in Chapter 5. We can avoid this problem if we swap the last two lines of *append*^{o}.

```
(define appendo
  (λ (l s out)
    (conde
      ((≡ () l) (≡ s out))
      ((exist (a d)
         (≡ (a . d) l)
         (exist (res)
           (≡ (a . res) out)
           (appendo d s res)))))))
```

This final version of *append*^{o} illustrates an important principle: unifications should always come before recursive calls, or calls to other "serious" relations.

2.3 Impure Operators

In this section we include several *impure* operators that appear in earlier work on miniKanren, notably Friedman et al. (2005) and Near et al. (2008): **project**, **cond**^{a}, **cond**^{u}, *once*^{o}, and *copy-term*^{o}. These operators are not considered part of core miniKanren, and are inherently non-relational since they may not work correctly for every goal ordering of a program; also, it is not legal to pass only fresh variables to some of these operators, namely *once*^{o} and *copy-term*^{o}. As a result we only use these operators to demonstrate impure Prolog-like features, for example in Chapter 10 during translation of the lean*TAP* theorem prover from Prolog to miniKanren. Importantly, the final version of the translated prover does not use any impure operators.

project can be used to access the values associated with logic variables. For example, the expression

```
(run* (q)
  (exist (x)
    (≡ 5 x)
    (≡ (* x x) q)))
```

has no value, since Scheme's multiplication function operates only on numbers, not logic variables associated with numbers. We can solve this problem by projecting x: within the body of the **project** form, x is a lexical variable bound to 5.

```
(run* (q)
  (exist (x)
    (≡ 5 x)
    (project (x)
      (≡ (* x x) q)))) ⇒

(25)
```

Unfortunately, the expression

```
(run* (q)
  (exist (x)
    (project (x)
      (≡ (* x x) q))
    (≡ 5 x)))
```

has no value, since x is unassociated when $(* \ x \ x)$ is evaluated. This example demonstrates that **project** is not a relational operator[6].

cond^a and **cond**^{u} are used to prune a program's search tree, and can be used in place of Prolog's *cut* (`!`)[7]. The examples from chapter 10 of *The Reasoned Schemer* (Friedman et al. 2005) demonstrate uses of **cond**^{a} and **cond**^{u}, and the pitfalls that await the unsuspecting programmer.

cond^a and **cond**^{u} differ from **cond**^{e} in that at most one clause can succeed. Furthermore, the clauses are tried in order, from top to bottom. Also, the first goal in each clause is treated specially, as a "test" goal that determines whether to commit to that clause; in this way, **cond**^{a} and **cond**^{u} are reminiscent of **cond**.

For example,

```
(run* (x)
  (cond^a
    ((≡ olive x))
    ((≡ oil x))))
```

returns (olive) since **cond**^{a} commits to the first clause when (≡ olive x) succeeds. However,

```
(run* (x)
  (cond^a
    ((≡ virgin x) (≡ #t #f))
    ((≡ olive x))
    ((≡ oil x))))
```

returns () since (≡ #t #f) fails, and since **cond**^{a} committed to the first clause once (≡ virgin x) succeeded. The expression

```
(run* (q)
  (cond^a
    ((≡ #t #f))
    (always^o))
  (≡ #t q))
```

[6] We explore a relational approach to arithmetic in Chapter 6.

[7] More specifically, **cond**^{a} corresponds to a *soft-cut* (Clocksin 1997), while **cond**^{u} corresponds to Mercury's *committed-choice* (Henderson et al. 1996; Naish 1995).

diverges. The "test" goal for the first clause, (#t #f), fails. The test goal for the second clause, *always*^{o}, succeeds; therefore **cond**^{a} commits to this clause. Since *always*^{o} can succeed an unbounded number of times, the **run*** expression diverges.

However, if we replace **cond**^{a} with **cond**^{u}, the resulting expression

```
(run* (q)
  (condu
    ((≡ #t #f))
    (alwayso))
  (≡ #t q))
```

returns (#t). This is because the test goal of a **cond**^{u} clause can succeed at most once, which is the only difference between **cond**^{a} and **cond**^{u}.

The next impure operator, *once*^{o}, can be trivially defined using **cond**^{u}. *once*^{o} takes a single argument, which must be a goal; *once*^{o} ensures that when the goal is run it produces at most a single answer.

```
(define onceo
  (λ (g)
    (condu
      (g))))
```

For example, (**run*** (*q*) (*once*^{o} *always*^{o})) produces (_$_0$).

copy-term^o creates a copy of its first argument, consistently replacing unassociated logic variables with new variables; the resulting copy is then associated with the second argument.

```
(run* (q)
  (exist (w x y z)
    (≡ (a x 5 y x) w)
    (copy-termo w z)
    (≡ (w z) q))) ⇒
```

```
(((a _0 5 _1 _0) (a _2 5 _3 _2)))
```

A major theme of Chapter 10 is how *copy-term*^{o} can be replaced with a relational combination of nominal unification and tagging, at least in certain cases.

Chapter 3

Implementation I: Core miniKanren

In this chapter we present the implementation of the core miniKanren operators described in Chapter 2. Later chapters describe additions or modifications to this core implementation; unless otherwise stated, these later chapters only present the definitions that differ from those of the core implementation.

This chapter is organized as follows. In section 3.1 we describe our representation of variables and substitutions, and define the *unify* function, which uses the *walk* function to look up variables in a triangular substitution. Section 3.2 presents our reification algorithm, which converts miniKanren terms into regular Scheme values without logic variables. Finally, in section 3.3, we discuss miniKanren goals, which map substitutions to (potentially infinite) streams of substitutions. We then define the core miniKanren goal constructors ≡, **exist**, and **cond**^{e}, along with the interface operator **run**.

3.1 Variables, Substitutions, and Unification

We represent logic variables as vectors of length one[1].

```
(define-syntax var
  (syntax-rules ()
    ((_ x) (vector x))))

(define-syntax var?
  (syntax-rules ()
    ((_ x) (vector? x))))
```

[1] R6RS Scheme supports records, which arguably provide a better abstraction for logic variables. We use vectors for compatibility with R5RS Scheme—one consequence is that vectors should not appear in arguments passed to *unify*.

The single argument to the **var** constructor is a symbol representing the name of the variable[2].

A *substitution* s is a mapping between logic variables and values (also called *terms*). We represent a substitution as an *association list*, which is a list of pairs associating vectors to values; we construct an empty substitution using *empty-s*

```
(define empty-s ())
```

and extend an existing substitution s with a new association between a variable x and a value v using *ext-s-no-check*

```
(define ext-s-no-check (λ (x v s) (cons (x . v) s)))
```

If x, y, and z are logic variables constructed using **var**, then the association list ((*x* . 5) (*y* . #t)) represents a substitution that associates x with 5, y with #t, and leaves z unassociated.

The right-hand-side (*rhs*) of an association may itself be a logic variable. In the substitution ((*y* . 5) (*x* . *y*)), x is associated with y, which in turn is associated with 5. Thus, both x and y are associated with 5. This representation is known as a "triangular" substitution, as opposed to the more common "idempotent" representation[3] of ((*y* . 5) (*x* . 5)). (See Baader and Snyder (2001) for more on substitutions.) One advantage of triangular substitutions is that they can be easily extended using *cons*, without side-effecting or rebuilding the substitution. This lack of side-effects permits sharing of substitutions, while substitution extension remains a constant-time operation. This sharing, in turn, gives us backtracking for free—we just "forget" irrelevant associations by using an older version of the substitution, which is always a suffix of the current substitution.

Triangular substitution representation is well-suited for functional implementations of logic programming, since it allows sharing of substitutions. Unfortunately, there are several significant disadvantages to the triangular representation. The major disadvantage is that variable lookup is both more complicated and more expensive[4] than with idempotent substitutions. With idempotent substitutions, variable lookup can be defined as follows, where *rhs*[5] returns the right-hand-side of an association.

[2] This name is useful for debugging. More importantly, we must ensure that the vectors created with **var** are non-empty. This is because we use Scheme's *eq?* test to distinguish between variables, and *eq?* is not guaranteed to distinguish between two non-empty vectors.

[3] In an *idempotent* substitution, a variable that appears on the left-hand-side of an association never appears on the rhs.

[4] In Chapter 4 we will explore several ways to improve the efficiency of variable lookup when using triangular substitutions.

[5] *rhs* is just defined to be *cdr*.

```
(define lookup
  (λ (v s)
    (cond
      ((var? v)
       (let ((a (assq v s)))
         (cond
           (a (rhs a))
           (else v))))
      (else v))))
```

If *v* is an unassociated variable, or a non-variable term, *lookup*[6] just returns *v*.

When looking up a variable in a triangular substitution, we must instead use the more complicated *walk* function.

```
(define walk
  (λ (v s)
    (cond
      ((var? v)
       (let ((a (assq v s)))
         (cond
           (a (walk (rhs a) s))
           (else v))))
      (else v))))
```

If, when walking a variable *x* in a substitution *s*, we find that *x* is bound to another variable *y*, we must then walk *y* in the original substitution *s*. *walk* is therefore not primitive recursive (Kleene 1952)—in fact, *walk* can diverge if used on a substitution containing a circularity; for example, when walking *x* in either the substitution **((***x* **.** *x***))** or **((***y* **.** *x***) (***x* **.** *y***))**. miniKanren's unification function, *unify*, ensures that these kinds of circularities are never introduced into a substitution. In addition, *unify* prohibits circularities of the form **((***x* **. (***x***)))** from being added to the substitution. Although this circularity will not cause *walk* to diverge, it can cause divergence during reification (described in section 3.2). To prevent circularities from being introduced, we extend the substitution using *ext-s* rather than *ext-s-no-check*.

```
(define ext-s
  (λ (x v s)
    (cond
      ((occurs√ x v s) #f)
      (else (ext-s-no-check x v s)))))
```

[6] For fans of syntactic sugar, this definition can be shortened using **cond**'s arrow notation.

```
(define lookup
  (λ (v s)
    (cond
      ((and (var? v) (assq v s)) ⇒ rhs)
      (else v))))
```

```
(define occurs√
  (λ (x v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) (eq? v x))
        ((pair? v) (or (occurs√ x (car v) s) (occurs√ x (cdr v) s)))
        (else #f)))))
```

ext-s calls the *occurs√* predicate, which returns #t if adding an association between *x* and *v* would introduce a circularity. If so, *ext-s* returns #f instead of an extended substitution, indicating that unification has failed.

unify unifies two terms *u* and *v* with respect to a substitution *s*, returning a (potentially extended) substitution if unification succeeds, and returning #f if unification fails or would introduce a circularity[7].

```
(define unify
  (λ (u v s)
    (let ((u (walk u s))
          (v (walk v s)))
      (cond
        ((eq? u v) s)
        ((var? u)
         (cond
           ((var? v) (ext-s-no-check u v s))
           (else (ext-s u v s))))
        ((var? v) (ext-s v u s))
        ((and (pair? u) (pair? v))
         (let ((s (unify (car u) (car v) s)))
           (and s (unify (cdr u) (cdr v) s))))
        ((equal? u v) s)
        (else #f)))))
```

The call to *occurs√* from within *ext-s* is potentially expensive, since it must perform a complete tree walk on its second argument. Therefore, we also define *unify-no-check*, which performs *unsound* unification but is more efficient than *unify*[8].

[7] Observe that *unify* calls *ext-s-no-check* rather than *ext-s* if *u* and *v* are distinct unassociated variables, thereby avoiding an unnecessary call to *walk* from inside *occurs√*.

[8] Apt and Pellegrini (1992) point out that, in practice, omission of the occurs check is usually not a problem. However, the type inferencer presented in section 9.3 requires sound unification to prevent self-application from typechecking.

```
(define unify-no-check
  (λ (u v s)
    (let ((u (walk u s))
          (v (walk v s)))
      (cond
        ((eq? u v) s)
        ((var? u) (ext-s-no-check u v s))
        ((var? v) (ext-s-no-check v u s))
        ((and (pair? u) (pair? v))
         (let ((s (unify-no-check (car u) (car v) s)))
           (and s (unify-no-check (cdr u) (cdr v) s))))
        ((equal? u v) s)
        (else #f)))))
```

3.2 Reification

Reification is the process of turning a miniKanren term into a Scheme value that does not contain logic variables. The *reify* function takes a substitution *s* and an arbitrary value *v*, perhaps containing variables, and returns the reified value of *v*.

```
(define reify
  (λ (v s)
    (let ((v (walk* v s)))
      (walk* v (reify-s v empty-s)))))
```

For example, (*reify* (5 *x* (#t *y* *x*) *z*) *empty-s*) returns (5 _$_0$ (#t _$_1$ _$_0$) _$_2$).

reify uses *walk** to deeply walk a term with respect to a substitution. If *s* is the substitution ((*z* . 6) (*y* . 5) (*x* . (*y* *z*))), then (*walk* *x* *s*) returns (*y* *z*) while (*walk** *x* *s*) returns (5 6)[9].

```
(define walk*
  (λ (v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) v)
        ((pair? v) (cons (walk* (car v) s) (walk* (cdr v) s)))
        (else v)))))
```

reify also calls *reify-s*, which is the heart of the reification algorithm.

```
(define reify-s
  (λ (v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) (ext-s v (reify-name (length s)) s))
        ((pair? v) (reify-s (cdr v) (reify-s (car v) s)))
        (else s)))))
```

[9] If *s* is idempotent, *walk** is equivalent to *walk*.

reify-s takes a *walk**ed term as its first argument; its second argument starts out as *empty-s*. The result of invoking *reify-s* is a *reified name* substitution, associating logic variables to distinct symbols of the form $\_{}_n$.

reify-s in turn relies on *reify-name* to produce the actual symbol.

```
(define reify-name
  (λ (n)
    (string→symbol (string-append "_." (number→string n)))))
```

3.3 Goals and Goal Constructors

A goal g is a function that maps a substitution s to an ordered sequence of zero or more values—these values are almost always substitutions. (For clarity, we notate λ as λ_G when creating such a function g.) Because the sequence of values may be infinite, we represent it not as a list but as a special kind of stream, a^∞.

Such streams contain either zero, one, or more values (Kiselyov et al. 2005; Spivey and Seres 2003). We use (**mzero**) to represent the empty stream of values. If a is a value, then (**unit** a) represents the stream containing just a. To represent a non-empty stream we use (**choice** a f), where a is the first value in the stream, and where f is a function of zero arguments. (For clarity, we notate λ as λ_F when creating such a function f.) Invoking the function f produces the remainder of the stream. (**unit** a) can be represented as (**choice** a (λ_F () (**mzero**))), but the **unit** constructor avoids the cost of building and taking apart pairs and invoking functions, since many goals return only singleton streams. To represent an incomplete stream, we create an f using (**inc** e), where e is an *expression* that evaluates to an a^∞.

```
(define-syntax mzero
  (syntax-rules ()
    ((_) #f)))

(define-syntax unit
  (syntax-rules ()
    ((_ a) a)))

(define-syntax choice
  (syntax-rules ()
    ((_ a f) (cons a f))))

(define-syntax inc
  (syntax-rules ()
    ((_ e) (λF () e))))
```

To ensure that streams produced by these four a^∞ constructors can be distinguished, we assume that a singleton a^∞ is never #f, a function, or a pair whose *cdr* is a function. To discriminate among these four cases, we define **case**$^\infty$.

```
(define-syntax case∞
  (syntax-rules ()
    ((_ e (() e0) ((f̂) e1) ((â) e2) ((a f) e3))
     (let ((a∞ e))
       (cond
         ((not a∞) e0)
         ((procedure? a∞)  (let ((f̂ a∞)) e1))
         ((and (pair? a∞) (procedure? (cdr a∞)))
          (let ((a (car a∞)) (f (cdr a∞))) e3))
         (else (let ((â a∞)) e2)))))))
```

The simplest goal constructor is ≡, which returns either a singleton stream or an empty stream, depending on whether the arguments unify with the implicit substitution. As with the other goal constructors, ≡ always expands to a goal, even if an argument diverges.

```
(define-syntax ≡
  (syntax-rules ()
    ((_ u v)
     (λG (a)
       (cond
         ((unify u v a) ⇒ (λ (a) (unit a)))
         (else (mzero)))))))
```

We can also define ≡*-no-check*, which performs unsound unification without the occurs check.

```
(define-syntax ≡-no-check
  (syntax-rules ()
    ((_ u v)
     (λG (a)
       (cond
         ((unify-no-check u v a) ⇒ (λ (a) (unit a)))
         (else (mzero)))))))
```

cond^e is a goal constructor that combines successive **cond**^{e}-clauses using **mplus***. To avoid unwanted divergence, we treat the **cond**^{e}-clauses as a single **inc** stream. Also, we use the same implicit substitution for each **cond**^{e}-clause. **mplus*** relies on *mplus*, which takes an a^∞ and an f and combines them (a kind of *append*). Using **inc**, however, allows an argument to *become* a stream, thus leading to a relative fairness because all of the stream values will be interleaved.

```
(define-syntax conde
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λG (a)
       (inc
         (mplus* (bind* (g0 a) g ...) (bind* (g1 a) ĝ ...) ...))))))
```

```
(define-syntax mplus*
  (syntax-rules ()
    ((_ e) e)
    ((_ e0 e ...) (mplus e0 (λF () (mplus* e ...))))))
```

```
(define mplus
  (λ (a∞ f)
    (case∞ a∞
      (() (f))
      ((f̂) (inc (mplus (f) f̂)))
      ((a) (choice a f))
      ((a f̂) (choice a (λF () (mplus (f) f̂)))))))
```

If the body of **cond**^{e} were just the **mplus*** expression, then the **inc** clauses of *mplus*, *bind*, and *take* (defined below) would never be reached, and there would be no interleaving of values.

exist is a goal constructor that first lexically binds its variables (created by **var**) and then, using **bind***, combines successive goals. **bind*** is short-circuiting: since the empty stream (**mzero**) is represented by #f, any failed goal causes **bind*** to immediately return #f. **bind*** relies on *bind* (Moggi 1991; Wadler 1992), which applies the goal g to each element in a^∞. These a^∞'s are then merged together with *mplus* yielding an a^∞. (*bind* is similar to Lisp's *mapcan*, with the arguments reversed.)

```
(define-syntax exist
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λG (a)
       (inc
         (let ((x (var x)) ...)
           (bind* (g0 a) g ...)))))))
```

```
(define-syntax bind*
  (syntax-rules ()
    ((_ e) e)
    ((_ e g0 g ...) (bind* (bind e g0) g ...))))
```

```
(define bind
  (λ (a∞ g)
    (case∞ a∞
      (() (mzero))
      ((f) (inc (bind (f) g)))
      ((a) (g a))
      ((a f) (mplus (g a) (λF () (bind (f) g)))))))
```

To minimize heap allocation we create a single λ_{G} closure for each goal constructor, and we define **bind*** and **mplus*** to manage sequences, not lists, of goal-like expressions.

run, and therefore *take*, converts an f to a list.

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x) g0 g ...)
     (take n
       (λF ()
         ((exist (x) g0 g ...
            (λG (a)
              (cons (reify x a) ())))
          empty-s))))))
```

```
(define take
  (λ (n f)
    (if (and n (zero? n))
      ()
      (case∞ (f)
        (() ())
        ((f) (take n f))
        ((a) a)
        ((a f) (cons (car a) (take (and n (− n 1)) f)))))))
```

We wrap the result of (*reify x s*) in a list so that the **case**$^\infty$ in *take* can distinguish a singleton a^∞ from the other three a^∞ types. We could simplify **run** by using **var** to create the unassociated variable x, but we prefer that **exist** be the only operator that calls **var**[10]. If the first argument to *take* is #f, we get the behavior of **run***. It is trivial to write a read-eval-print loop that uses **run***'s interface by redefining *take*.

3.4 Impure Operators

We conclude this chapter by defining the impure operators introduced in section 2.3: **project**, which can be used to access the values of variables, **cond**^{a} and **cond**^{u}, which can be used to prune the search tree of a program, and *copy-term*^{o}, which copies a term, consistently replacing unassociated logic variables with new variables.

project applies the implicit substitution to zero or more lexical variables, rebinds those variables to the values returned, and then evaluates the goal expressions in its body. The body of a **project** typically includes at least one **begin** expression—any expression is a goal expression if its value is a miniKanren goal.

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g g* ...)
     (λG (s)
       (let ((x (walk* x s)) ...)
         ((exist () g g* ...) s))))))
```

[10] This becomes important in Chapter 4, when we redefine the way **exist** uses **var**.

copy-term^o creates a copy of its first argument, consistently replacing unassociated variables with new logic variables in the copy. The term u is projected before copying, to avoid accidentally replacing associated variables with new variables.

```
(define copy-term^o
  (λ (u v)
    (project (u)
      (≡ (walk* u (build-s u ())) v))))
```

```
(define build-s
  (λ (u s)
    (cond
      ((var? u) (if (assq u s) s (cons (cons u (var ignore)) s)))
      ((pair? u) (build-s (cdr u) (build-s (car u) s)))
      (else s))))
```

Unlike **cond**^{e}, only one **cond**^{a}-clause or **cond**^{u}-clause can return an a^∞: the first clause whose first goal succeeds. With **cond**^{a}, the entire stream returned by the first goal is passed to **bind***. With **cond**^{u}, a singleton stream is passed to **bind***—this stream contains the first value of the stream returned by the first goal.

```
(define-syntax cond^a
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λ_G (a)
       (inc
         (if^a ((g0 a) g ...)
               ((g1 a) ĝ ...) ...))))))
```

```
(define-syntax cond^u
  (syntax-rules ()
    ((_ (g0 g ...) (g1 ĝ ...) ...)
     (λ_G (a)
       (inc
         (if^u ((g0 a) g ...)
               ((g1 a) ĝ ...) ...))))))
```

```
(define-syntax if^a
  (syntax-rules ()
    ((_) (mzero))
    ((_ (e g ...) b ...)
     (let loop ((a∞ e))
       (case∞ a∞
         (() (if^a b ...))
         ((f) (inc (loop (f))))
         ((a) (bind* a∞ g ...))
         ((a f) (bind* a∞ g ...)))))))
```

```
(define-syntax if^u
  (syntax-rules ()
    ((_) (mzero))
    ((_ (e g ...) b ...)
     (let loop ((a^∞ e))
       (case^∞ a^∞
         (() (if^u b ...))
         ((f) (inc (loop (f))))
         ((a) (bind* a^∞ g ...))
         ((a f) (bind* (unit a) g ...)))))))
```

Chapter 4

Implementation II: Optimizing *walk*

In this chapter we examine the efficiency of the *walk* algorithm presented in Chapter 3, which is the heart of the unification algorithm. We present various optimizations to *walk*, which significantly improve performance of unification, and indeed the entire miniKanren implementation.

This chapter is organized as follows. In section 4.1 we examine the worst-case performance of the *walk* algorithm, with emphasis on the cost of looking up an unassociated variable. Section 4.2 introduces an optimization using *birth records*, which can greatly increase the speed of looking up an unassociated variable. In section 4.3 we look at an additional optimization that requires we rewrite *walk* using explicit recursion instead of *assq*. Finally, section 4.4 shows how we can further improve on the birth-records optimization by storing the current substitution in a variable when it is first introduced.

4.1 Why *walk* is Expensive

In the worst case, the number of cdrs and tests performed by *walk* is quadratic in the length of the substitution. This happens when looking up a variable at the beginning of a long "unification chain"—for example, when looking up v in the "perfectly triangular" substitution $((y \,.\, z)\ (x \,.\, y)\ (w \,.\, x)\ (v \,.\, w))$. Contrast this with the linear cost of looking up v in the equivalent idempotent substitution $((y \,.\, z)\ (x \,.\, z)\ (w \,.\, z)\ (v \,.\, z))$.

Fortunately, extremely long unification chains rarely occur in real logic programs. Rather, the major cost of variable lookups is in walking unassociated variables. When using triangular substitutions (or even idempotent substitutions), the entire substitution must be examined to determine that a variable in unassociated[1].

[1] Prolog implementations based on the Warren Abstract Machine (Aït-Kaci 1991) do not use

One solution to this problem is to use a more sophisticated data structure to represent triangular substitutions—for example, we might use a trie (Fredkin 1960) instead of a list, to ensure logarithmic cost when looking up an unassociated variable[2]. For simplicity we will retain our association list representation of substitutions. Instead of changing the substitution representation, we will use a trick to determine if a variable is unassociated without having to look at the entire substitution.

4.2 Birth Records

To avoid examining the entire substitution when walking an unassociated variable, we will add a *birth record* to the substitution whenever we introduce a variable using **exist**. For example, to run the goal (**exist** (*x* *y*) (≡ 5 *x*)) we would add the birth records **(***x* **.** *x***)** and **(***y* **.** *y***)** to the current substitution, then run (≡ 5 *x*) in the extended substitution. Unifying *x* with 5 requires us to walk *x*: when we do so, we immediately encounter the birth record **(***x* **.** *x***)**, indicating *x* is unassociated. Unification then succeeds, adding the association **(***x* **.** 5**)** to the substitution to produce **((***x* **.** 5**)** **(***x* **.** *x***)** **(***y* **.** *y***)** ...**)**.

Here are **exist** and *walk*, modified to use birth records.

```
(define-syntax exist
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λG (s)
       (inc
         (let ((x (var x)) ...)
           (let* ((s (ext-s x x s))
                  ...)
             (bind* (g0 s) g ...))))))))
```

explicit substitutions to represent variable associations. Instead, they represent each variable as a mutable box, and side-effect the box during unification. This makes variable lookup extremely fast, but requires remembering and undoing these side-effects during backtracking. In addition, this simple model assumes a depth-first search strategy, whereas our purely functional representation can be used with interleaving search without modification.

[2] Abdulaziz Ghuloum has implemented miniKanren using a trie-based representation of triangular substitutions. David Bender and Lindsey Kuper have extended this work, using a variety of purely functional data structures to represent triangular substitutions. These more sophisticated representations of substitutions can result in much faster walking of variables, which can greatly speed up many miniKanren programs. The best performance for their benchmarks was achieved using a skew binary number representation within a random access list (Okasaki 1995).

```
(define walk
  (λ (v s)
    (cond
      ((var? v)
       (let ((a (assq v s)))
         (cond
           (a (if (eq? (rhs a) v) v (walk (rhs a) s)))
           (else v))))
      (else v))))
```

Technically, birth records ensure that we need not examine the entire substitution to determine a variable is unassociated. However, in the worst case our situation has not improved[3]: if a variable is introduced at the beginning of a program, but is not unified until the end of the program, the birth record will occur at the very end of the substitution, and lookup will still take linear time. Fortunately, in most real-world programs variables are unified shortly after they have been introduced. This locality of reference means that, in practice, birth records significantly reduce the cost of walking unassociated variables.

4.3 Eliminating *assq* and Checking the *rhs*

We can optimize *walk* in another way, although we will need to eliminate our call to *assq*, and introduce a recursion using "named" **let** [4]. Here is the standard *walk*, without birth records.

```
(define walk
  (λ (v ŝ)
    (let loop ((s ŝ))
      (cond
        ((var? v)
         (cond
           ((null? s) v)
           ((eq? v (lhs (car s))) (walk (rhs (car s)) ŝ))
           (else (loop (cdr s)))))
        (else v)))))
```

We can optimize *walk* by exploiting an important property of the triangular substitutions produced by *unify*: in the substitution $((x \mathrel{.} y) \mathrel{.} \hat{s})$, the variable y will never appear in the left-hand-side (lhs) of any binding in $\hat{s}$. Therefore, when walking a variable y we can look for y in both the lhs and rhs of each association.

[3] Indeed, the situation is even worse, since the birth records more than double the length of the substitution that must be walked.

[4] This chapter assumes miniKanren is run under an optimizing compiler, such as Ikarus Scheme or Chez Scheme. When run under an interpreter, the "named"-**let** based *walk* described in this section may run much slower than the *assq*-based version, since *assq* is often hand-coded in C. When running under an interpreter, the *assq*-based *walk* with birth records will probably be fastest.

If y is the lhs, we found the variable we are looking for, and need to walk the rhs in the original substitution. However, if we find y in the rhs of an association, we know that y is unassociated.

Here is the optimized version of *walk*

```
(define walk
  (λ (v ŝ)
    (let loop ((s ŝ))
      (cond
        ((var? v)
         (cond
           ((null? s) v)
           ((eq? v (rhs (car s))) v)
           ((eq? v (lhs (car s))) (walk (rhs (car s)) ŝ))
           (else (loop (cdr s)))))
        (else v)))))
```

where *lhs* and *rhs*[5] return the left-hand-side and right-hand-side of an association, respectively[6].

Once we make a recursive call to *walk*, the *null?* test becomes superfluous, so we redefine *walk* using the *step* helper function.

```
(define walk
  (λ (v ŝ)
    (let loop ((s ŝ))
      (cond
        ((var? v)
         (cond
           ((null? s) v)
           ((eq? v (rhs (car s))) v)
           ((eq? v (lhs (car s))) (step (rhs (car s)) ŝ))
           (else (loop (cdr s)))))
        (else v)))))
```

[5] *lhs* is just defined to be *car*; *rhs* is just defined to be *cdr*.

[6] By checking the rhs before the lhs, we ensure that *walk* always terminates, even with substitutions that contain circularities. If the substitution contains a circularity of the form $(x \,.\, x)$ (a birth record), then walking x clearly terminates, since the *rhs* test will find x before performing the recursion. If the substitution contains associations $(x \,.\, y)$ and $(y \,.\, x)$, walking x still terminates despite the circularity. Assume $(y \,.\, x)$ appears after $(x \,.\, y)$ (which will never happen for substitutions returned by *unify*); then when we walk x, we will end up walking y in the recursion. But we will then find y on the rhs of $(x \,.\, y)$, which will end the walk. The only other possibility is that $(y \,.\, x)$ appears before $(x \,.\, y)$. In this case, walking x does not result in a recursive call, since we find x on the rhs of $(y \,.\, x)$. Similar reasoning applies for arbitrarily complicated circularity chains.

```
(define step
  (λ (v ŝ)
    (let loop ((s ŝ))
      (cond
        ((var? v)
         (cond
           ((eq? v (rhs (car s))) v)
           ((eq? v (lhs (car s))) (step (rhs (car s)) ŝ))
           (else (loop (cdr s)))))
        (else v)))))
```

4.4 Storing the Substitution in the Variable

We now combine the birth records optimization presented in section 4.2 with checking for the walked variable in the rhs of each association, described in section 4.3. However, we wish to avoid polluting the substitution with birth records, which not only lengthen the substitution but also violate important invariants of our substitution representation[7]. Instead of adding birth records to the substitution, we will add a “birth substitution” to each variable by storing the current substitution in the variable when it is created.

```
(define-syntax exist
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λG (s)
       (inc
         (let ((x (var s)) ...)
           (bind* (g0 s) g ...)))))))
```

Now, instead of checking for the birth records as we walk down the substitution, we check if the entire substitution is *eq?* to the substitution stored in the walked variable; if so, we know the variable is unassociated[8].

[7] Namely, that a variable never appears on the lhs of more than one association, and that substitutions never contain circularities of the form $(x . x)$.

[8] It should be noted that none of these optimizations avoid the $n + 1$ passes that might be required when looking up a variable in a perfectly triangular substitution of length n.

Here, then, is the most efficient definition of *walk*[9].

```
(define walk
  (λ (v ŝ)
    (let loop ((s ŝ))
      (cond
        ((var? v)
         (cond
           ((eq? (vector-ref v 0) s) v)
           ((eq? v (rhs (car s))) v)
           ((eq? v (lhs (car s))) (step (rhs (car s)) ŝ))
           (else (loop (cdr s)))))
        (else v)))))
```

[9] Exercise for the reader: show that this definition of *walk* works correctly on the renaming substitution used in reification (section 3.2)

Chapter 5

A Slight Divergence

In this chapter we explore the divergence of relational programs. We present several divergent miniKanren programs; for each program we consider different techniques that can be used to make the program terminate.

By their very nature, relational programs are prone to divergence. As relational programmers, we may ask for an infinite number of answers from a program, or we may look for a non-existent answer in an infinite search tree. In fact, miniKanren programs can (and do!) diverge for a variety of reasons. A frustration common to beginning miniKanren programmers is that of carefully writing or deriving a program, only to have it diverge on even simple test cases. Learning to recognize the sources of divergence in a program, and which techniques can be used to achieve termination, is a critical stage in the evolution of every relational programmer.

To help miniKanren programmers write relations that terminate, this chapter presents several divergent example programs; for each program, we discuss why it diverges, and how the divergence can be avoided.

It is important to remember that a single relational program may contain multiple, and completely different, causes of divergence; such programs may require a variety of techniques in order to terminate[1]. Also, a single technique may be useful for avoiding multiple causes of divergence, as will be made clear in the examples below. miniKanren does not currently support all of these techniques (such as operators on cyclic terms)—unsupported techniques are clearly identified in the text. Even techniques not yet supported by miniKanren are of value, however, since they may be supported by other programming languages.

We now present the divergent example programs, along with techniques for avoiding divergence.

[1] Challenge for the reader: construct a single miniKanren program that contains every cause of divergence discussed in this chapter. Then use the techniques from this chapter to “fix” the program.

Example 1

Consider the divergent **run*** expression

```
(run* (q)
  (exist (x y z)
    (plus^o x y z)
    (≡ (x y z) q)))
```

where *plus*^{o} is the ternary addition relation defined in Chapter 6. This expression diverges because (*plus*^{o} x y z) succeeds an unbounded number of times; therefore, the **run*** never stops producing answers. Although it could be argued that this is a “good” infinite loop, and that we got what we asked for, presumably we want to see some of these answers. Also, the user has no way of knowing that the system is producing any answers, since the divergence might be due to one of the other causes described below. (Not to mention that, in general, the user cannot tell whether the program is diverging or merely taking a very long time to produce an answer.)

We can avoid this divergence in several different ways:

1. We could replace the **run*** with **run**^{n}, where n is some positive integer. This will return the n answers, although miniKanren’s interleaving search makes the order in which answers are produced difficult to predict.

2. Instead of using the **run** interface, we could directly manipulate the answer stream passed as the second argument to *take* (Chapter 3), and examine the answers one at a time. This the “read-eval-print loop” approach is used by Prolog systems, and is trivial to implement in miniKanren by redefining *take*.

3. We can use *once*^{o} or **cond**^{u} to ensure that goals that might succeed an unbounded number of times succeed only once. Of course, these operators are non-declarative, so we reject this approach. Instead, it would be better to use a **run**$^{\mathbf{1}}$.

4. A more sophisticated approach is to represent infinitely many answers as a single answer by using constraints. For example, one way to express that x is a natural number other than 2 is to associate x with $0, 1, 3, \ldots$. Clearly, there are infinitely many such associations, and enumerating them can lead to an unbounded number of answers. Instead, we might represent the same information using the single disequality constraint ($\neq$ 2 x).

   Similarly, we might use a clever data representation rather than a constraint to represent infinitely many answers as a single term. For example, using the little-endian binary representation of natural numbers presented in Chapter 6, the term (1 . x) represents any one of the infinitely many odd naturals.

   Using this technique, programs that previously produced infinitely many answers may fail finitely, proving that no more answers exist. Unfortunately, it is not always possible to find a constraint or data representation to concisely represent infinitely many terms. For example, although the data representation from Chapter 6 makes it easy to express every odd natural as a single term, there is no little-endian binary list that succinctly represents every prime number. Similarly, disequality constraints are not sufficient to concisely express that some term does not appear in an uninstantiated tree[2].

Example 2

Consider the divergent **run¹** expression

(**run¹** (*q*) (≡-*no-check* (*q*) *q*))

The unification of *q* with (*q*) results in a substitution containing a circularity[3]: ((q . (*q*))). However, it is not unification that diverges, or subsequent calls to *walk*. Rather, the reification of *q* at the end of the computation calls *walk** (Chapter 3), which diverges[4].

We can avoid this divergence in several different ways:

1. We can use ≡ rather than ≡-*no-check* to perform sound unification with the occurs check. The goal (≡ (*q*) *q*) violates the occurs check and therefore fails; hence, (**run¹** (*q*) (≡ (*q*) *q*)) returns () rather than diverging[5]. Since the occurs check can be expensive, we may wish to restrict ≡ to only those unifications that might introduce a circularity, such as in the application line of a type inferencer; this requires reasoning about the program. Alternatively, we can always be safe by using only ≡ rather than ≡-*no-check*[6].

2. Since the reification of *q* causes divergence in this example, the **run** expression will terminate if we do not reify the variable associated with the circularity. For example,

   (**run¹** (*q*) (**exist** (*x*) (≡-*no-check* (*x*) *x*)))

   returns ($\_\__0$). Although the **run** expression terminates, the resulting substitution is still circular: ((*x* . (*x*))). However, unless we allow infinite terms, the unification (≡-*no-check* (*x*) *x*) is *unsound*. This is a problem for the type inferencers based on the simply typed λ-calculus, for example, since

[2] However, the freshness constraint (#) described in Chapter 9 allows us to express a similar constraint.

[3] The ≠-*no-check* disequality operator (Chapter 7) suffers from the same problem, since it can add circularities to the constraint store.

[4] The non-logical operator **project** also calls *walk**, and can therefore diverge on circular substitutions.

[5] Similarly, we can use ≠ rather than ≠-*no-check* when introducing disequality constraints.

[6] As pointed out by Apt and Pellegrini (1992) this approach may be overly conservative. However, since our primary interest is in avoiding divergence, this approach seems reasonable.

self-applications such as (*f f*) should not type check (see the inferencer in section 9.3). If we do not perform the occurs check, and the circular term is not reified, the type inference will succeed instead of failing. Clearly this is not an acceptable way to avoid divergence. However, it is important to understand why the program above terminates, since it is possible to unintentionally write programs that abuse unsound unification, unless we use ≡ everywhere.

3. Since reification is the cause of divergence in this example, we can just avoid reification entirely and return the raw substitution. The user must determine which associations in the substitution are of interest; furthermore, the user must check the substitution for circularities introduced by unsound unification. There is one more problem with both this approach and the previous one: the occurs check can prevent divergence by making the program fail early, which may avoid an unbounded number of successes or a futile search for a non-existent answer in an infinite search space.

4. Another approach to avoiding divergence is to allow infinite (or *cyclic*) terms, as introduced by Prolog II (Colmerauer 1985, 1984, 1982). Then the unification (≡*-no-check* (*q*) *q*) is sound, even though it returns a circular substitution. miniKanren does not currently support infinite terms; however, it would not be difficult to extend the reifier to handle cyclic terms, just as many Scheme implementations can print circular lists.

Example 3

Consider the divergent **run**$^{\mathbf{1}}$ expression[7]

(**run**$^{\mathbf{1}}$ (*q*) *always*^{o} *fail*)

where *fail* is defined as (≡ #t #f). Recall that the body of a **run** is an implicit conjunction[8]. In order for the **run** expression to succeed, both *always*^{o} and *fail* must succeed. First, *always*^{o} succeeds, then *fail* fails. We then backtrack into *always*^{o},

[7] Recall that *always*^{o} was defined in Chapter 2 as (**define** *always*^{o} (*any*^{o} (≡ #f #f))). However, for the purposes of this chapter we define *always*^{o} as

```
(define alwayso
  (letrec ((alwayso (λ ()
                       (conde
                         ((≡ #f #f))
                         ((alwayso))))))
    (alwayso)))
```

This is because tabling (Chapters 12 and 13) uses reification to determine if a call is a variant of a previously tabled call. Since all procedures have the same reified form (#<procedure> under Chez Scheme, for example), and since *any*^{o} takes a goal (a procedure) as its argument, tabling *any*^{o} can lead to unsound behavior.

[8] (**run**$^{\mathbf{1}}$ (*q*) g_1 g_2) expands into an expression containing (**exist** () g_1 g_2).

which succeeds again, followed once again by failure of the *fail* goal. Since *always*^{o} succeeds an unbounded number of times, we repeat the cycle forever, resulting in divergence.

We can avoid this divergence in several different ways:

1. We could simply reorder the goals: (**run^1** (q) *fail always*^{o}). This expression returns **()** rather than diverging, since *fail* fails before *always*^{o} is even tried. miniKanren's conjunction operator (**exist**) is commutative, but only if an answer exists. If no answer exists, then reordering goals within an **exist** may result in divergence rather than failure[9].

   However, reordering goals has its disadvantages. For many programs, no ordering of goals will result in finite failure (see the remaining example in this chapter). Also, by committing to a certain goal ordering we are giving up on the declarative nature of relational programming: we are specifying *how* the program computes, rather than only *what* it computes. For these reasons, we should consider alternative solutions.

2. We may be able to use constraints or clever data structures to represent infinitely many terms as a single term (as described in Example 1). If we can use these techniques to make all the conjuncts succeed finitely many times, then the program will terminate regardless of goal ordering.

3. Another approach to making the conjuncts succeed finitely many times is to use tabling, described in Chapter 12. Tabling is a form of memoization—we remember every distinct call to the tabled goal, along with the answers produced. When a tabled goal is called, we check whether the goal has previously been called with similar arguments—if so, we use the tabled answers.

   In addition to potentially making goals more efficient by avoiding duplicate work, tabling can improve termination behavior by cutting off infinite recursions. For example, the tabled version of *always*^{o} succeeds exactly once rather than an unbounded number of times. Therefore, (**run^1** (q) *always*^{o} *fail*) returns **()** rather than diverging when *always*^{o} is tabled.

   Unfortunately, tabling has a major disadvantage: it does not work if one or more of the arguments to a tabled goal changes with each recursive call[10].

4. We could perform a *dependency analysis* on the conjuncts—if the goals do not share any logic variables, they cannot affect each other. Therefore we can run the goals in parallel, passing the original substitution to each goal. If either goal fails, the entire conjunction fails. If both goals succeed, we take the Cartesian product of answers from the goals, and use those new associations to extend the original substitution.

[9] We say that conjunction is commutative, *modulo divergence versus failure.*

[10] As demonstrated by the *gen*^{o} example in a later footnote.

miniKanren does not currently support this technique; however, miniKanren's interleaving search should make it straightforward to run conjuncts in parallel. A run-time dependency analysis would also be easy to implement[11].

5. We could address the problem directly by trying to make our conjunction operator commutative. For example, we could run both goal orderings in parallel[12], (**exist** () always^o fail) and (**exist** () always^o fail), and see if either ordering converges. If so, we could commit to this goal ordering. Unfortunately, this commitment may be premature, since the goal ordering we picked might diverge when we ask for a second answer, while the other ordering may fail finitely after producing a single answer.

   We could try *all* possible goal orderings, but this is prohibitively expensive for all but the simplest programs. In particular, recursive goals containing conjunctions will result in an exponential explosion in the number of orderings.

   For these reasons, miniKanren does not currently provide a commutative conjunction operator. However, future versions of miniKanren may include an operator that *simulates* full commutative conjunction using a combination of tabling, parallel goal evaluation, and continuations (see the Future Work chapter).

Example 4

Consider the **run**$^{\mathbf{1}}$ expression (**run**$^{\mathbf{1}}$ (x) (plus^o **2** x **1**)). If plus^o represents the ternary addition relation over natural numbers, there is no value for q that satisfies (plus^o **2** x **1**) (since $2 + x = 1$ has no solution in the naturals). Ideally, the **run**$^{\mathbf{1}}$ expression will return **()**. However, a naive implementation of plus^o that enumerates values for x will diverge, since it will keep associating x with larger numbers without bound. Since x grows with each recursive call, tabling plus^o will not help.

We can avoid this divergence in several different ways:

1. We can relax the domain of x to include negative integers—then the **run**$^{\mathbf{1}}$ expression will return **(-1)**. However, changing **run**$^{\mathbf{1}}$ to **run**$^{\mathbf{2}}$ still results in divergence, since $2 + x = 1$ has only a single solution in the integers.

2. We could use a domain-specific constraint system. For example, instead of writing an addition goal, we could use Constraint Logic Programming over the integers (also known as "CLP(Z)"). If we restrict the sizes of our numbers, we could use CLP(FD) (Constraint Logic Programming over finite domains).

[11] Ciao Prolog (Hermenegildo and Rossi 1995) performs dependency analysis of conjuncts, along with many other analyses, to support efficient parallel logic programming.

[12] We might do this by wrapping the goals in a fern (Chapter 14).

Alas, no single constraint system can express every interesting relation in a non-trivial application. We could try to create a custom constraint system for each application we write, but this may be a very difficult task, especially since constraints may interact with other language features in complex ways.

miniKanren currently supports four kinds of constraints: unification and disunification constraints using $\equiv$ and $\neq$ (Chapters 2 and 7); α-equivalence constraints using nominal unification (Chapter 9); and freshness constraints using $\#$ (Chapter 9)[13]. Future versions of miniKanren will likely support more sophisticated constraints.

3. Another approach is to bound the size of the terms in the recursive calls to $plus^o$. For example, if we represent numbers as binary lists, we know that the lengths of the first two arguments to $plus^o$ (the summands) should never exceed the length of the third argument (the sum). By encoding these bounds on term size in our $plus^o$ relation, the call ($plus^o$ **2** x **1**) will fail finitely. We use exactly this technique when defining $plus^o$ in Chapter 6.

   Bounding term sizes is a very powerful technique, as is demonstrated in the relational arithmetic chapter of this dissertation. But as with the other techniques presented in this chapter, it has its limitations. Establishing relationships between argument sizes may require considerable insight into the relation being expressed. In fact, the arithmetic definitions in Chapter 6, including the bounds on term size, were derived from mathematical equations; this code would be almost impossible to write otherwise[14].

   Furthermore, overly-eager bounds on term size can themselves cause divergence. For example, assume that we know arguments x and y represent lists, which must be of the same length. We might be tempted to first determine the length of x, then determine the length of y, and finally compare the result. However, if x is an unassociated logic variable, it has no fixed length: we could cdr down x forever, inadvertently lengthening x as we go. Instead, we must *simultaneously* compare the lengths of x and y. To make the task more difficult, we want to enforce the bounds while we are performing the primary computation of the relation (for example, while performing addition in the case of $plus^o$). In fact, lazily enforcing complex bounds between multiple arguments is likely to be more difficult than writing the underlying relation.

   Another problem with bounds on term sizes is that they may not help when arguments share logic variables. For example, consider the $lessl^o$ relation: ($lessl^o$ x y) succeeds if x and y are lists, and y is longer than x. We can easily implement $lessl^o$ by simultaneously cdring down x and y:

[13] Some non-published versions of miniKanren have also supported *pa/ir* constraints: (*pa/ir* x) expresses that x can never be instantiated as a pair. Uses of *pa/ir* can typically be removed through careful use of tagging, however, so we do not include the constraint in this dissertation.

[14] For example, see the definition of log^o in section 6.6.

```
(define lessl^o
  (λ^e (x y)
    ((() (_ . _)))
    (((_ . xd) (_ . yd))
     (lessl^o xd yd))))
```

However, consider the call (*lessl*^{o} *x x*). The first λ^e clause fails, while the second clause results in a recursive call where both arguments are the same uninstantiated variable. Therefore (*lessl*^{o} *x x*) diverges.

If we were to table *lessl*^{o}, (*lessl*^{o} *x x*) would fail instead of diverging. Unfortunately, sharing of arguments in more complicated relations may result in arguments growing with each recursive call, which would defeat tabling.

In this section we have examined several divergent miniKanren programs, investigated the causes of their divergence[15], and considered techniques we can use to make these programs converge. As miniKanren programmers, divergence, and how to avoid it, should never be far from our minds. Indeed, every extension to the core miniKanren language can be viewed as a new technique for avoiding divergence[16].

In the next chapter we present a relational arithmetic system that uses bounds on term size to establish strong termination guarantees.

[15] miniKanren's interleaving search avoids some forms of divergence that afflict Prolog, which uses an incomplete search strategy equivalent to depth-first search. For example, the left-recursive *swappend*^{o} relation from Chapter 2 is equivalent to the standard *append*^{o} relation in miniKanren. In Prolog, however, *swappend*^{o} diverges in many cases that *append*^{o} terminates, even when answers exist. (Although tabling can be used to avoid divergence for left-recursive Prolog goals—indeed, this is one of the main reasons for including tabling in a Prolog implementation.)

[16] For example, the freshness constraints of nominal logic allow us to express that a nom *a* does not occur free within a variable *x*. Without such a constraint, we would need to instantiate *x* to a potentially unbounded number of ground terms to establish that *a* does not appear in the term.

Chapter 6

Applications I: Pure Binary Arithmetic

This chapter presents relations for arithmetic over the non-negative integers: addition, subtraction, multiplication, division, exponentiation, and logarithm. Importantly, these relations are refutationally complete—if an individual arithmetic relation is called with arguments that do not satisfy the relation, the relation will fail in finite time rather than diverge. The conjunction of two or more arithmetic relations may not fail finitely, however. This is because the conjunction of arithmetic relations can express Diophantine equations; were such conjunctions guaranteed to terminate, we would be able to solve Hilbert's 10^{th} problem, which is undecidable (Matiyasevich 1993). We also do not guarantee termination if the goal's arguments share variables, since sharing can express the conjunction of sharing-free relations.

Kiselyov et al. (2008) gives proofs of refutational completeness for these relations. Friedman et al. (2005) and Kiselyov et al. (2008) give additional examples and exposition of these arithmetic relations[1].

This chapter is organized as follows. Section 6.1 describes our representation of numbers. In section 6.2 we present a naive implementation of addition and show its limitations. Section 6.3 presents a more sophisticated implementation of addition, inspired by the half-adders and full-adders of digital hardware. Sections 6.4 and 6.5 present the multiplication and division relations, respectively. Finally in section 6.6 we define relations for logarithm and exponentiation.

[1] The definition of log^o in the first printing of Friedman et al. (2005) contains an error, which has been corrected in the second printing and in section 6.6.

6.1 Representation of Numbers

Before we can write our arithmetic relations, we must decide how we will represent numbers. For simplicity, we restrict the domain of our arithmetic relations to non-negative integers[2]. We might be tempted to use Scheme's built-in numbers for our arithmetic relations. Unfortunately, unification cannot decompose Scheme numbers. Instead, we need an inductively defined representation of numbers that can be constructed and deconstructed using unification. We will therefore represent numbers as lists.

The simplest approach would be to use a unary representation[3]; however, for efficiency we will represent numbers as lists of binary digits. Our lists of binary digits are *little-endian*: the *car* of the list contains the least-significant-bit, which is convenient when performing arithmetic. We can define the *build-num* helper function, which constructs binary little-endian lists from Scheme numbers.

```
(define build-num
  (λ (n)
    (cond
      ((zero? n) ())
      ((and (not (zero? n)) (even? n))
       (cons 0 (build-num (quotient n 2))))
      ((odd? n)
       (cons 1 (build-num (quotient (− n 1) 2)))))))
```

For example (*build-num* 6) returns (0 1 1), while (*build-num* 19) returns (1 1 0 0 1).

To ensure there is a unique representation of every number, we suppress trailing 0's. Thus (0 1) is the unique representation of the number two; both (0 1 0) and (0 1 0 0) are illegal. Similarly, () is the unique representation of zero; (0) is illegal. Lists representing numbers may be partially instantiated: (1 . *x*) represents any odd integer, while (0 . *y*) represents any *positive* even number. We must ensure that our relations never instantiate variables representing numbers to illegal values—in these examples, *x* can be instantiated to any legal number, while *y* can be instantiated to any number *other* than zero to avoid creating the illegal value (0).

We can now define the simplest useful arithmetic relations, pos^o and $>\mathbf{1}^o$. The pos^o relation is satisfied if its argument represents a positive integer.

```
(define pos^o
  (λ^e (n)
    (((a . d)))))
```

The $>\mathbf{1}^o$ relation is satisfied if its argument represents an integer greater than one.

[2] We could extend our treatment to negative integers by adding a sign tag to each number.

[3] Even when using unary numbers, defining refutationally complete arithmetic relations is non-trivial, as demonstrated by Kiselyov et al. (2008).

```
(define >1^o
  (λ^e (n)
    (((a b . d)))))
```

We will use *pos*^{o} and >**1**^{o} in more sophisticated arithmetic relations, starting with addition.

6.2 Naive Addition

Now that we have decided on a representation for numbers, we can define the addition relation, *plus*^{o}.

```
(define plus^o
  (λ^e (n m s)
    ((x () x))
    ((() y y))
    (((0 . x) (b . y) (b . res))
     (plus^o x y res))
    (((b . x) (0 . y) (b . res))
     (plus^o x y res))
    (((1 . x) (1 . y) (0 . res))
     (exist (res-1)
       (plus^o x y res-1)
       (plus^o (1) res-1 res)))))
```

The first two clauses handle when n or m is zero. The next two clauses handle when both n and m are positive integers, at least one of which is even. The final clause handles when n and m are both positive odd integers.

At first glance, our definition of *plus*^{o} seems to work fine.

```
(run^1 (q) (plus^o (1 1) (0 1 1) q)) ⇒ ((1 0 0 1))
```

As expected, adding three and six yields nine. However, replacing **run**^{1} with **run*** results in the answer ((1 0 0 1) (1 0 0 1)). The duplicate value is due to the overlapping of clauses in *plus*^{o}—for example, both of the first two clauses succeed when n, m, and s are all zero. Even worse, (**run*** (q) (*plus*^{o} (0 1) q (1 0 1))) returns ((1 1) (1 1) (1 1 0) (1 1 0)). The last two values are not even legal representations of a number, since the most-significant bit is zero.

We can fix these problems by making the clauses of *plus*^{o} non-overlapping, and by adding calls to *pos*^{o} to ensure the most-significant bit of a positive number is never instantiated to zero.

```
(define plus^o
  (λ^e (n m k)
    ((x () x))
    ((() (x . y) (x . y)))
    (((0 . x) (0 . y) (0 . res)) (pos^o x) (pos^o y)
     (plus^o x y res))
    (((0 . x) (1 . y) (1 . res)) (pos^o x)
     (plus^o x y res))
    (((1 . x) (0 . y) (1 . res)) (pos^o y)
     (plus^o x y res))
    (((1 . x) (1 . y) (0 . res))
     (exist (res-1)
       (plus^o x y res-1)
       (plus^o (1) res-1 res)))))
```

We separated the third clause of the original *plus*^{o} into two clauses, so we can use *pos*^{o} to avoid illegal instantiations of numbers.

The improved definition of *plus*^{o} no longer produces duplicate or illegal values.

```
(run* (q) (plus^o (1 1) (0 1 1) q)) ⇒ ((1 0 0 1))
(run* (q) (plus^o (0 1) q (1 0 1))) ⇒ ((1 1))
```

It may appear that our new *plus*^{o} is refutationally complete, since attempting to add eight to some number q to produce six fails finitely:

```
(run* (q) (plus^o (0 0 0 1) q (0 1 1))) ⇒ ()
```

Unfortunately, this example is misleading—*plus*^{o} is not refutationally complete. The expression (**run**$^{\mathbf{1}}$ (q) (*plus*^{o} q (1 0 1) (0 0 0 1))) returns ((1 1)) as expected, but replacing **run**$^{\mathbf{1}}$ with **run**$^{\mathbf{2}}$ results in divergence. Similarly,

```
(run^6 (q)
  (exist (x y)
    (plus^o x y (1 0 1))
    (≡ (x y) q)))
```

returns

```
(((1 0 1) ())
 (() (1 0 1))
 ((0 0 1) (1))
 ((1) (0 0 1))
 ((0 1) (1 1))
 ((1 1) (0 1)))
```

but **run**$^{\mathbf{7}}$ diverges. If we were to swap the recursive calls in last clause of *plus*^{o}, the previous expressions would converge when using **run***; unfortunately, many previously convergent expressions would then diverge[4]. If we want *plus*^{o} to be refutationally complete, we must reconsider our approach.

[4] These examples demonstrate why an efficient implementation (or simulation) of commutative conjunction would be useful.

6.3 Arithmetic Revisited

In this section we develop a refutationally complete definition of *plus*^{o}, inspired by the half-adders and full-adders of digital logic[5].

We first define *half-adder*^{o}, which, when given the binary digits x, y, r, and c, satisfies the equation $x + y = r + 2 \cdot c$.

```
(define half-adder^o
  (λ (x y r c)
    (exist ()
      (bit-xor^o x y r)
      (bit-and^o x y c))))
```

half-adder^o is defined using bit-wise relations for logical `and` and `exclusive-or`.

```
(define bit-and^o
  (λ^e (x y r)
    ((0 0 0))
    ((1 0 0))
    ((0 1 0))
    ((1 1 1))))
```

```
(define bit-xor^o
  (λ^e (x y r)
    ((0 0 0))
    ((0 1 1))
    ((1 0 1))
    ((1 1 0))))
```

Now that we have defined *half-adder*^{o}, we can define *full-adder*^{o}. *full-adder*^{o} is similar to *half-adder*^{o}, but takes a carry-in bit b; given bits b, x, y, r, and c, *full-adder*^{o} satisfies $b + x + y = r + 2 \cdot c$.

```
(define full-adder^o
  (λ (b x y r c)
    (exist (w xy wz)
      (half-adder^o x y w xy)
      (half-adder^o w b r wz)
      (bit-xor^o xy wz c))))
```

half-adder^o and *full-adder*^{o} add individual bits. We now define *adder*^{o} in terms of *full-adder*^{o}; *adder*^{o} adds a carry-in bit d to arbitrarily large numbers n and m to produce a number r.

[5] See Hennessy and Patterson (2002) for a description of hardware adders.

```
(define adder^o
  (λ (d n m r)
    (match^e (d n m)
      ((0 _ ()) (≡ n r))
      ((0 () _) (≡ m r) (pos^o m))
      ((1 _ ())
       (adder^o 0 n (1) r))
      ((1 () _)
       (pos^o m)
       (adder^o 0 (1) m r))
      ((_ (1) (1))
       (exist (a c)
         (≡ (a c) r)
         (full-adder^o d 1 1 a c)))
      ((_ (1) _)
       (gen-adder^o d n m r))
      ((_ _ (1))
       (>1^o n) (>1^o r)
       (adder^o d (1) n r))
      ((_ _ _)
       (>1^o n)
       (gen-adder^o d n m r)))))
```

The last clause of *adder*^{o} calls *gen-adder*^{o}; given the bit d and numbers n, m, and r, *gen-adder*^{o} satisfies $d + n + m = r$, provided that m and r are greater than one and n is positive.

```
(define gen-adder^o
  (λ (d n m r)
    (match^e (n m r)
      (((a . x) (b . y) (c . z))
       (exist (e)
         (pos^o y) (pos^o z)
         (full-adder^o d a b c e)
         (adder^o e x y z))))))
```

We are finally ready to redefine *plus*^{o}.

```
(define plus^o (λ (n m k) (adder^o 0 n m k)))
```

As proved by Kiselyov et al. (2008), this definition of *plus*^{o} is refutationally complete. Using the new *plus*^{o} all the addition examples from the previous section terminate, even when using **run***. We can also generate triples of numbers, where the sum of the first two numbers equals the third.

```
(run^9 (q)
  (exist (x y r)
    (plus^o x y r)
    (≡ (x y r) q))) ⇒

((_0 () _0)
 (() (_0 . _1) (_0 . _1))
 ((1) (1) (0 1))
 ((1) (0 _0 . _1) (1 _0 . _1))
 ((1) (1 1) (0 0 1))
 ((0 _0 . _1) (1) (1 _0 . _1))
 ((1) (1 0 _0 . _1) (0 1 _0 . _1))
 ((0 1) (0 1) (0 0 1))
 ((1) (1 1 1) (0 0 0 1)))
```

We can take advantage of the flexibility of the relational approach by defining subtraction in terms of addition.

```
(define minus^o (λ (n m k) (plus^o m k n)))
```

minus^o works as expected:

```
(run* (q) (minus^o (0 0 0 1) (1 0 1) q)) ⇒ ((1 1))
```

eight minus five is indeed three. *minus^o* is also refutationally complete:

```
(run* (q) (minus^o (0 1 1) q (0 0 0 1))) ⇒ ()
```

there is no non-negative integer q that, when subtracted from six, produces eight.

6.4 Multiplication

Next we define the multiplication relation *mul^o*, which satisfies $n \cdot m = p$.

```
(define mul^o
  (λ (n m p)
    (match^e (n m)
      ((() _) (≡ () p))
      ((_ ()) (≡ () p) (pos^o n))
      (((1) _) (≡ m p) (pos^o m))
      ((_ (1)) (≡ n p) (>1^o n))
      (((0 . x) _)
       (exist (z)
         (≡ (0 . z) p)
         (pos^o x) (pos^o z) (>1^o m)
         (mul^o x m z)))
      (((1 . x) (0 . y))
       (pos^o x) (pos^o y)
       (mul^o m n p))
      (((1 . x) (1 . y))
       (pos^o x) (pos^o y)
       (odd-mul^o x n m p)))))
```

mul^o is defined in terms of the helper relation *odd-mul*^{o}.

```
(define odd-mul^o
  (λ (x n m p)
    (exist (q)
      (bound-mul^o q p n m)
      (mul^o x m q)
      (plus^o (0 . q) m p))))
```

For detailed descriptions of *mul*^{o} and *odd-mul*^{o}, see (Friedman et al. 2005) and (Kiselyov et al. 2008). From a refutational-completeness perspective, the definition of *bound-mul*^{o} is most interesting.

bound-mul^o ensures that the product of n and m is no larger than p by enforcing that the length[6] of n plus the length of m is an upper bound for the length of p. In the process of enforcing this bound, *bound-mul*^{o} length-instantiates q—that is, q becomes a list of fixed length containing uninstantiated variables representing binary digits. The length of q, written $\|q\|$, satisfies $\|q\| < \min(\|p\|, \|n\| + \|m\| + 1)$.

```
(define bound-mul^o
  (λ (q p n m)
    (match^e (q p)
      ((() (_ . _)))
      (((_ . x) (_ . y))
       (exist (a z)
         (cond^e
           ((≡ () n)
            (≡ (a . z) m)
            (bound-mul^o x y z ()))
           ((≡ (a . z) n)
            (bound-mul^o x y z m))))))))
```

mul^o works as expected:

```
(run* (p) (mul^o (1 0 1) (1 1) p)) ⇒ (1 1 1 1)
```

multiplying five by three yields fifteen. Thanks to the bounds on term sizes enforced by *bound-mul*^{o}, *mul*^{o} is refutationally complete:

```
(run* (q) (mul^o (0 1) q (1 1))) ⇒ ()
```

there exists no non-negative integer q that, when multiplied by two, yields three.

As we expect of all our relations, *mul*^{o} is flexible—it can even be used to factor numbers. For example, this **run*** expression returns all the factors of twelve.

[6] More correctly, the length of the list representing the number.

```
(run* (q)
  (exist (m)
    (mul^o q m (0 0 1 1)))) ⇒

((1) (0 0 1 1) (0 1) (0 0 1) (1 1) (0 1 1))
```

6.5 Division

Next we define a relation that performs division with remainder. We will need additional bounds on term sizes to define division (and logarithm in section 6.6).

The relation $=l^o$ ensures that the lists representing the numbers n and m are the same length. As before, we must take care to avoid instantiating either number to an illegal value like (0).

```
(define =l^o
  (λ^e (n m)
    ((() ()))
    (((1) (1)))
    (((a . x) (b . y)) (pos^o x) (pos^o y)
     (=l^o x y))))
```

$<l^o$ ensures that the length of the list representing n is less than that of m.

```
(define <l^o
  (λ^e (n m)
    ((() _) (pos^o m))
    (((1) _) (>1^o m))
    (((a . x) (b . y)) (pos^o x) (pos^o y)
     (<l^o x y))))
```

We can now define $\leqslant l^o$ by combining $=l^o$ and $<l^o$.

```
(define ⩽l^o
  (λ (n m)
    (cond^e
      ((=l^o n m))
      ((<l^o n m)))))
```

Using $<l^o$ and $=l^o$ we can define $<^o$, which ensures that the value of n is less than that of m.

```
(define <^o
  (λ (n m)
    (cond^e
      ((<l^o n m))
      ((=l^o n m)
       (exist (x)
         (pos^o x)
         (plus^o n x m))))))
```

Combining $<^o$ and $\equiv$ leads to the definition of $\leqslant^o$.

```
(define ⩽o
  (λ (n m)
    (conde
      ((≡ n m))
      ((<o n m)))))
```

With the bounds relations in place, we can define division with remainder. The div^o relation takes numbers n, m, q, and r, and satisfies $n = m{\cdot}q{+}r$, with $0 \leq r < m$; this is equivalent to the equation $\frac{n}{m} = q$ with remainder r, with $0 \leq r < m$. A simple definition of div^o is

```
(define divo
  (λ (n m q r)
    (exist (mq)
      (<o r m)
      (⩽lo mq n)
      (mulo m q mq)
      (pluso mq r n))))
```

Unfortunately, (**run*** (*m*) (**exist** (*r*) (*div^o* (1 0 1) *m* (1 1 1) *r*))) diverges. Because we want refutational completeness, we instead use the more sophisticated definition

```
(define divo
  (λ (n m q r)
    (matche q
      (() (≡ r n) (<o n m))
      ((1) (=lo n m) (pluso r m n) (<o r m))
      (__ (<lo m n) (<o r m) (poso q)
       (exist (nh nl qh ql qlm qlmr rr rh)
         (splito n r nl nh)
         (splito q r ql qh)
         (conde
           ((≡ () nh)
            (≡ () qh)
            (minuso nl r qlm)
            (mulo ql m qlm))
           ((poso nh)
            (mulo ql m qlm)
            (pluso qlm r qlmr)
            (minuso qlmr nl rr)
            (splito rr r () rh)
            (divo nh m qh rh))))))))
```

The refutational completeness of div^o is largely due to the use of $<^o$, $<l^o$, and $=l^o$ to establish bounds on term sizes. div^o is described in detail in Friedman et al. (2005).

div^o relies on the relation *$split^o$* to 'split' a binary numeral at a given length: (*$split^o$* n r l h) holds if $n = 2^{s+1} \cdot l + h$ where $s = \|r\|$ and $h < 2^{s+1}$. *$split^o$* can construct n by combining the lower-order bits[7] of l with the higher-order bits of h, inserting *padding* bits as specified by the length of r—*$split^o$* is essentially a specialized version of *$append^o$*. *$split^o$* ensures that illegal values like (0) are not constructed by removing the rightmost zeros after splitting the number n into its lower-order bits and its higher-order bits.

```
(define $split^o$
  ($\lambda^e$ ($n$ $r$ $l$ $h$)
    ((() _ () ()))
    (((0 $b$ . $\hat{n}$) () () ($b$ . $\hat{n}$)))
    (((1 . $\hat{n}$) () (1) $\hat{n}$))
    (((0 $b$ . $\hat{n}$) ($a$ . $\hat{r}$) () _)
     ($split^o$ ($b$ . $\hat{n}$) $\hat{r}$ () $h$))
    (((1 . $\hat{n}$) ($a$ . $\hat{r}$) (1) _)
     ($split^o$ $\hat{n}$ $\hat{r}$ () $h$))
    ((($b$ . $\hat{n}$) ($a$ . $\hat{r}$) ($b$ . $\hat{l}$) _)
     ($pos^o$ $\hat{l}$)
     ($split^o$ $\hat{n}$ $\hat{r}$ $\hat{l}$ $h$))))
```

6.6 Logarithm and Exponentiation

We end this chapter by defining relations for logarithm with remainder and exponentiation.

[7] The lowest bit of a positive number n is the car of n.

```
(define log^o
  (λ^e (n b q r)
    (((1) _ () ()) (pos^o b))
    ((_ _ () _) (<^o n b) (plus^o r (1) n))
    ((_ _ (1) _) (>1^o b) (=l^o n b) (plus^o r b n))
    ((_ (1) _ _) (pos^o q) (plus^o r (1) n))
    ((_ () _ _) (pos^o q) (≡ r n))
    (((a b̂ . dd) (0 1) _ _) (pos^o dd)
     (exp2^o n () q)
     (exist (s) (split^o n dd r s)))
    ((_ _ _ _)
     (exist (a b̂ add ddd)
       (cond^e
         ((≡ (1 1) b))
         ((≡ (a b̂ add . ddd) b))))
     (<l^o b n)
     (exist (bw1 bw nw nw1 ql1 ql s)
       (exp2^o b () bw1)
       (plus^o bw1 (1) bw)
       (<l^o q n)
       (exist (q̂ bwq1)
         (plus^o q (1) q̂)
         (mul^o bw q̂ bwq1)
         (<^o nw1 bwq1))
       (exp2^o n () nw1)
       (plus^o nw1 (1) nw)
       (div^o nw bw ql1 s)
       (plus^o ql (1) ql1)
       (⩽l^o ql q)
       (exist (bql qh s qdh qd)
         (repeated-mul^o b ql bql)
         (div^o nw bw1 qh s)
         (plus^o ql qdh qh)
         (plus^o ql qd q)
         (⩽^o qd qdh)
         (exist (bqd bq1 bq)
           (repeated-mul^o b qd bqd)
           (mul^o bql bqd bq)
           (mul^o b bq bq1)
           (plus^o bq r n)
           (<^o n bq1)))))))
```

Given numbers n, b, q, and r, *log*^{o} satisfies $n = b^q + r$, where $0 \leq n$ and where q is the largest number that satisfies the equation. The *log*^{o} definition is similar to *div*^{o}, but uses exponentiation rather than multiplication[8].

[8] A line-by-line description of the Prolog version of *log*^{o} and its helper relations can be found at `http://okmij.org/ftp/Prolog/Arithm/pure-bin-arithm.prl`

log^o relies on helpers $\mathit{exp2}^o$ and $\mathit{repeated\text{-}mul}^o$. $\mathit{exp2}^o$ is a simplified version of exponentiation; given our binary representation of numbers, exponentiation using base two is particularly simple. ($\mathit{exp2}^o$ n **()** q) satisfies $n = 2^q$; the more general ($\mathit{exp2}^o$ n b q) satisfies $n = (\|b\| + 1)^q + r$ for some r, where q is the largest such number and $0 \leq 2 \cdot r < n$, provided that b is length-instantiated and $\|b\| + 1$ is a power of two.

```
(define exp2^o
  (λ (n b q)
    (match^e (n q)
      (((1) ()))
      ((_ (1))
       (>1^o n)
       (exist (s)
         (split^o n b s (1))))
      ((_ (0 . q̂))
       (exist (b̂)
         (pos^o q̂)
         (<l^o b n)
         (append^o b (1 . b) b̂)
         (exp2^o n b̂ q̂)))
      ((_ (1 . q̂))
       (exist (nh b̂ s)
         (pos^o q̂)
         (pos^o nh)
         (split^o n b s nh)
         (append^o b (1 . b) b̂)
         (exp2^o nh b̂ q̂))))))
```

($\mathit{repeated\text{-}mul}^o$ n q nq) satisfies $nq = n^q$ provided n is length-instantiated and q is fully instantiated.

```
(define repeated-mul^o
  (λ (n q nq)
    (match^e q
      (() (≡ (1) nq) (pos^o n))
      ((1) (≡ n nq))
      (_
        (>1^o q)
        (exist (q̂ nq1)
          (plus^o q̂ (1) q)
          (repeated-mul^o n q̂ nq1)
          (mul^o nq1 n nq))))))
```

This simple log^o example shows that $14 = 2^3 + 6$.

```
(run* (q) (log^o (0 1 1 1) (0 1) (1 1) q)) ⇒ (0 1 1)
```

A more sophisticated example of log^o is

```
(run^9 (s)
  (exist (b q r)
    (log^o (0 0 1 0 0 0 1) b q r)
    (>1^o q)
    (≡ (b q r) s))) ⇒
```

```
((() (_0 _1 . _2) (0 0 1 0 0 0 1))
 ((1) (_0 _1 . _2) (1 1 0 0 0 0 1))
 ((0 1) (0 1 1) (0 0 1))
 ((1 1) (1 1) (1 0 0 1 0 1))
 ((0 0 1) (1 1) (0 0 1))
 ((0 0 0 1) (0 1) (0 0 1))
 ((1 0 1) (0 1) (1 1 0 1 0 1))
 ((0 1 1) (0 1) (0 0 0 0 0 1))
 ((1 1 1) (0 1) (1 1 0 0 1))),
```

which shows that:
$68 = 0^n + 68$ where n is greater than one,
$68 = 1^n + 67$ where n is greater than one,
$68 = 2^6 + 4$,
$68 = 3^3 + 59$,
$68 = 4^3 + 4$,
$68 = 8^2 + 4$,
$68 = 5^2 + 43$,
$68 = 6^2 + 32$, and
$68 = 7^2 + 19$.

We can define the exponentiation relation in terms of log^o.

```
(define exp^o (λ (b q n) (log^o n b q ())))
```

We can use exp^o to show that three to the fifth power is 243:

```
(run* (q) (exp^o (1 1) (1 0 1) q)) ⇒ (1 1 0 0 1 1 1 1).
```

The code in this chapter demonstrates the difficulty of achieving refutational completeness, even for relatively simple relations. Bounding the sizes of terms is a very powerful technique for ensuring termination, but can be tricky to apply. The definitions in this chapter were derived from equations defining arithmetic operators, and from the design of hardware half-adders and full-adders. It would have been extremely difficult to write this code from first principles.

Part II

Disequality Constraints

Chapter 7

Techniques I: Disequality Constraints

In this chapter we naively translate a Scheme program to miniKanren, and observe that the miniKanren relation exhibits undesirable behavior. This behavior is due to our inability to express negation in core miniKanren. We improve our miniKanren relation through the use of disequality constraints, which can express a limited form of negation.

This chapter is organized as follows. In section 7.1 we translate the Scheme function *rember* into the miniKanren relation *rember*^{o}. In section 7.2 we observe that *rember*^{o} produces unexpected answers that do not correspond to answers produced by *rember*. In section 7.3 we show that the unexpected answers are due to our failure to translate implicit tests in the *rember* function. Section 7.4 introduces disequality constraints, which allow us to express a limited form of negation. In section 7.5 we fix our definition of *rember*^{o} by adding a disequality constraint, thereby eliminating the unexpected answers. Finally in section 7.6 we point out several disadvantages of disequality constraints, and discuss when these constraints should be used.

7.1 Translating *rember* into miniKanren

We begin by naively translating the *rember* function into miniKanren. *rember* takes two arguments: a symbol x and a list of symbols *ls*, and removes the first occurrence of x from *ls*.

(*rember* b (a b c b d)) ⇒ (a c b d)

(*rember* d (a b c)) ⇒ (a b c)

Here is *rember*

```
(define rember
  (λ (x ls)
    (cond
      ((null? ls) ())
      ((eq? (car ls) x) (cdr ls))
      (else (cons (car ls) (rember x (cdr ls)))))))
```

To translate *rember* into the miniKanren relation *rember*^{o} we add a third argument *out*, change **cond** to **cond**^{e}, and replace uses of *null?*, *eq?*, *cons*, *car*, and *cdr* with calls to ≡. We also unnest the recursive call, using a temporary variable *res* to hold the “output” value of the recursive call.

```
(define rember^o
  (λ (x ls out)
    (cond^e
      ((≡ () ls) (≡ () out))
      ((exist (a d)
         (≡ (a . d) ls)
         (≡ a x)
         (≡ d out)))
      ((exist (a d res)
         (≡ (a . d) ls)
         (≡ (a . res) out)
         (rember^o x d res))))))
```

7.2 The Trouble with *rember*^{o}

For simple tests, it may seem that *rember*^{o} works as expected, mimicking the behavior of *rember*.

```
(run^1 (q) (rember^o b (a b c b d) q)) ⇒ ((a c b d))
```

```
(run^1 (q) (rember^o d (a b c) q)) ⇒ ((a b c))
```

However, we notice a problem if we replace the **run**^{1} with **run***.

```
(run* (q) (rember^o b (a b c b d) q)) ⇒ ((a c b d) (a b c d) (a b c b d))
```

Now there are multiple answers. The first answer is expected, but in the second answer *rember*^{o} removes the second occurrence of b rather than the first occurrence. The last answer is even worse—*rember*^{o} does not remove either b, as is evidenced by the **run** expression

```
(run* (q) (rember^o b (b) (b))) ⇒ (_0)
```

7.3 Reconsidering *rember*

Where did we go wrong? Is our miniKanren translation not faithful to the original Scheme program?

Not quite. The problem is that **cond** tries its clauses in order, stopping at the first clause whose test evaluates to a true value, while **cond**^{e} tries every possible clause. But isn't there only one **cond** clause that matches any given values of x and *ls*? Actually, no.

Let us examine the definition of *rember* once again.

```
(define rember
  (λ (x ls)
    (cond
      ((null? ls) ())
      ((eq? (car ls) x) (cdr ls))
      (else (cons (car ls) (rember x (cdr ls)))))))
```

Consider the call (*rember* a (a b c)). Clearly the *null?* test keeps the first clause from returning an answer, while the *eq?* test allows the second clause to produce an answer. But the test of the final clause, the "always-true" **else** keyword, is equivalent to the trivial #t test.

```
(define rember
  (λ (x ls)
    (cond
      ((null? ls) ())
      ((eq? (car ls) x) (cdr ls))
      (#t (cons (car ls) (rember x (cdr ls)))))))
```

If it were not for the second clause, the third clause would produce an answer for the call (*rember* a (a b c)). In fact, if we swap the last two clauses

```
(define rember
  (λ (x ls)
    (cond
      ((null? ls) ())
      (#t (cons (car ls) (rember x (cdr ls))))
      ((eq? (car ls) x) (cdr ls)))))
```

the call (*rember* a (a b c)) returns (a b c) rather than (b c).

What does the **else** test really mean in the original definition of *rember*? It means that the tests in all the above clauses must evaluate to #f. Similar reasoning holds for the *eq?* test of the second clause—the test implies that the *null?* test in the first clause returned #f. We can therefore redefine *rember* to make the implicit tests explicit.

```
(define rember
  (λ (x ls)
    (cond
      ((null? ls) ())
      ((and (not (null? ls)) (eq? (car ls) x))
       (cdr ls))
      ((and (not (null? ls)) (not (eq? (car ls) x)))
       (cons (car ls) (rember x (cdr ls)))))))
```

rember now produces the same answers no matter how we reorder the clauses; the clauses are now *non-overlapping*, since only a single clause can produce an answer for any specific call to *rember*[1].

```
(define rember
  (λ (x ls)
    (cond
      ((and (not (null? ls)) (not (eq? (car ls) x)))
       (cons (car ls) (rember x (cdr ls))))
      ((and (not (null? ls)) (eq? (car ls) x))
       (cdr ls))
      ((null? ls) ()))))
```

Even though we have reordered the **cond** clauses, *rember* works as expected.

(*rember* a (a b c)) ⇒ (b c)

7.4 Disequality Constraints

Now we can reconsider our definition of *rember*^{o}, adding the equivalent of the explicit tests to make our **cond**^{e} clauses non-overlapping[2].

Unfortunately, we do not have a way to express negation in core miniKanren[3]. However, we do not need full negation to express the test (*not* (*null? ls*)), since if *ls* is not null it must be a pair[4]. In fact, we are already expressing the (*not* (*null? ls*)) test implicitly, through the unification (≡ (*a* . *d*) *ls*) that appears in the last two **cond**^{e} clauses.

The only remaining test is (*not* (*eq?* (*car ls*) *x*)) in the last clause. How might we express that the car of *ls* is not *x*? We could attempt to unify the car of *ls* with every symbol other than *x*. Even if *x* were instantiated, to the symbol a for

[1] Throughout this dissertation we strive to write programs that adhere to the *non-overlapping principle*, to avoid duplicate or misleading answers. Such programs are similar to the guarded command programs described in Dijkstra (1975, 1997).

[2] More than one **cond**^{e} clause may succeed if *rember*^{o} is passed fresh variables. However, only one clause will succeed if the first two arguments to *rember*^{o} are fully ground.

[3] The impure operators **cond**^{a} and **cond**^{u} from section 2.3 can be used to express "negation as failure", as is commonly done in Prolog programs, but we eschew this non-declarative approach.

[4] This assumes, of course, that the second argument to *rember*^{o} can be unified with a proper list. Passing in 5 as the *ls* argument makes no more sense for *rember*^{o} than it does for *rember*.

example, we would have to unify *x* with every symbol *other* than a, of which there are infinitely many. Clearly this is problematic: enumerating an infinite domain can easily lead to divergent behavior[5].

Compare the tests in the second and third *rember* clauses: (*eq?* (*car ls*) *x*) and (*not* (*eq?* (*car ls*) *x*)). We use (≡ *a x*) to express that the car of *ls* (which is *a*) is equal to *x*. What we need is the ability to express the *disequality constraint*[6] (≠ *a x*)[7], which asserts that *a* and *x* are not equal, and can never be made equal through unification.

Before we add a disequality constraint to $rember^o$, let us examine some simple uses of ≠. In the first example, we unify *q* with 5, then specify that *q* can never be 5. As expected, the call to ≠ fails.

```
(run* (q) (≡ 5 q) (≠ 5 q)) ⇒ ()
```

If we swap the goals, the program behaves the same.

```
(run* (q) (≠ 5 q) (≡ 5 q)) ⇒ ()
```

≠ can take arbitrary expressions, as shown in the next two examples.

```
(run* (q) (≠ (+ 2 3) 5)) ⇒ ()
(run* (q) (≠ (* 2 3) 5)) ⇒ (_0)
```

In this **run*** expression we assert that *q* can never be 5 or 6. We express the latter constraint indirectly, by constraining *x*.

```
(run* (q)
  (exist (x)
    (≠ 5 q)
    (≡ x q)
    (≠ 6 x))) ⇒

((_0 : (never-equal ((_0 . 5)) ((_0 . 6)))))
```

The answer includes two reified constraints indicating that the output variable (*q*) can never be 5 or 6.

[5] It is possible to enumerate some infinite domains using a finite number of cases, through the use of clever data representation. For example, using the binary list notation from Chapter 6 we can express that a natural number *x* is not 5 by unifying *x* with the patterns (), (1), (*a* 1), (0 *a* 1), (1 1 1), and (*a b c d* . *rest*). Although this approach avoids divergence, it requires us to know the domain and representation of *x*. Furthermore, this approach may result in duplicate answers even for programs that adhere to the non-overlapping principle, which can be a problem even when enumerating finite domains.

[6] As opposed to an *equality constraint*, such as (≡ *a x*). Disequality is also known as *disunification*.

[7] We may also wish to introduce an operator *≠-no-check* that performs *unsound* disunification, to avoid the cost of the occurs check.

Consider this **run*** expression.

```
(run* (q)
  (exist (y z)
    (≠ (y . z) q))) ⇒
```

```
(_0)
```

It may seem that the constraint on q should be reified. However, this constraint can only be violated if q is unified with $(y \, . \, z)$. Since y and z are not reified, the constraint is not *relevant* and is therefore not reified.

To reify a constraint, we must reify all of the variables involved in the constraint.

```
(run* (q)
  (exist (x y z)
    (≠ (y . z) x)
    (≡ (x y z) q))) ⇒
```

```
(((_0 _1 _2) : (never-equal ((_0 _1 . _2)))))
```

The constraint is easier to interpret if we remember that (never-equal ((_0 _1 . _2))) is equivalent to (never-equal ((_0 . (_1 . _2)))).

Here is a slightly more complicated example of ≠.

```
(run* (q)
  (exist (x y z)
    (≡ (y . z) x)
    (≠ (5 . 6) x)
    (≡ 5 y)
    (≡ (x y z) q))) ⇒
```

```
((((5 . _0) 5 _0) : (never-equal ((_0 . 6)))))
```

Here is the same program, but with (≡ 6 y) instead of (≡ 5 y).

```
(run* (q)
  (exist (x y z)
    (≡ (y . z) x)
    (≠ (5 . 6) x)
    (≡ 6 y)
    (≡ (x y z) q))) ⇒
```

```
(((6 . _0) 6 _0))
```

Since y cannot be 5, (≠ (5 . 6) x) cannot be violated and is therefore discarded.

We end this section with a final example, to demonstrate how to interpret more complicated reified constraints.

```
(run* (q)
  (exist (x y z)
    (≠ 5 x)
    (≠ 6 x)
    (≠ (y 1) (2 z))
    (≡ (x y z) q))) ⇒
```

```
(((_0 _1 _2) : (never-equal ((_1 . 2) (_2 . 1)) ((_0 . 6)) ((_0 . 5)))))
```

The constraints `((_0 . 6))` and `((_0 . 5))` are independent of each other, and indicate that *x* can never be 5 or 6. However, `((_1 . 2) (_2 . 1))` represents a *single* constraint, indicating that *y* cannot be 2 if *z* is 1[8].

7.5 Fixing *rember*^{o}

Now that we understand ≠, and how to interpret reified constraints, we are ready to add the disequality constraint (≠ *a x*) to the last clause of *rember*^{o}.

```
(define rember^o
  (λ (x ls out)
    (cond^e
      ((≡ () ls) (≡ () out))
      ((exist (a d)
         (≡ (a . d) ls)
         (≡ a x)
         (≡ d out)))
      ((exist (a d res)
         (≡ (a . d) ls)
         (≠ a x)
         (≡ (a . res) out)
         (rember^o x d res))))))
```

If we re-run the programs from section 7.2 we see that *rember*^{o}'s behavior is consistent with that of *rember*.

```
(run* (q) (rember^o b (a b c b d) q)) ⇒ ((a c b d))
```

```
(run* (q) (rember^o b (b) (b))) ⇒ ()
```

Of course, *rember*^{o} is more flexible than *rember*.

```
(run* (q)
  (exist (x out)
    (rember^o x (a b c) out)
    (≡ (x out) q))) ⇒
```

```
((a (b c))
 (b (a c))
 (c (a b))
 ((_0 (a b c)) : (never-equal ((_0 . c)) ((_0 . b)) ((_0 . a)))))
```

[8] Reifying constraints in a friendly manner is non-trivial, as we will see in Chapter 8.

The final answer indicates that removing a symbol x from the list (a b c) results in the original list, provided that x is not a, b, or c.

7.6 Limitations of Disequality Constraints

Disequality constraints add expressive power to core miniKanren[9], allowing us to express a limited form of negation. However, disequality constraints have several limitations and disadvantages.

First, the $\neq$ operator can only express that two terms are never the same. This is much more limited than the ability to express full negation. For example, consider the test (**and** (*not* (*null? ls*)) (*not* (*eq?* (*car ls*) *x*))) from the version of *rember* in section 7.3. By de Morgan's law, this test is logically equivalent to (*not* (**or** (*null? ls*) (*eq?* (*car ls*) *x*))). We can use disequality constraints to express the first version of the test, but not the second.

Answers containing reified disequality constraints can be more difficult to interpret than answers without constraints. Also, it is not always obvious why a constraint was *not* reified (whether it was not relevant or could not be violated).

Disequality constraints also complicate the implementation of the unifier, and especially the reifier. Disequality constraints can also be expensive, since every constraint must be checked after each successful unification.

Because of these disadvantages, it is preferable to use $\equiv$ rather than $\neq$ whenever practical. For example, it is better to express the test (*not* (*null? ls*)) as ($\equiv$ (*a* . *d*) *ls*) rather than as ($\neq$ () *ls*).

Still, disequality constraints add expressive power to core miniKanren, and are generally preferable to enumerating infinite (or even finite) domains.

[9] It seems that disequality constraints were present in a very early version of Prolog (Colmerauer and Roussel 1996), although they were apparently removed after several years. Prolog II (Colmerauer 1985) reintroduced disequality constraints, which are now standard in most Prolog systems.

Chapter 8

Implementation III: Disequality Constraints

In this chapter we implement the $\neq$ disequality constraint operator described in Chapter 7. We implement disequality constraints using unification, which results in remarkably concise and elegant code. The mathematics of this approach were described by Comon in the 1980's[1]—to our knowledge, our implementation is the first to use this technique, for which triangular substitutions (section 3.1) are a perfect match. We also present a sophisticated reifier that removes irrelevant and redundant constraints.

This chapter is organized as follows. In section 8.1 we describe our representation of the constraint store, which is passed to every goal as part of a *package* that also contains the substitution. Section 8.2 presents the constraint solving algorithm, which is based on unification, while section 8.3 defines the $\neq$ and $\equiv$ operators and related helpers. Finally in section 8.4 we present a sophisticated reifier that produces human-friendly representations of constraints.

8.1 Constraints, Constraint Lists, and Packages

We represent a constraint c as a list of pairs associating variables with terms. For example, the constraint ($\neq$ 5 x) would be represented as ((x . 5)), while the constraint ($\neq$ (5 6) (y z)) would be represented as ((y . 5) (z . 6)). In fact, our representation of disequality constraints is identical to our representation of substitutions—indeed, a constraint can be viewed as a mini-substitution that indicates which simultaneous variable associations would violate the constraint.

A program can introduce many constraints, which requires that we introduce the notion of a *constraint store* that will be passed to every goal, along with the substitution. We represent our constraint store c^* as a list of constraints (that is, a

[1] See Comon (1991) and Comon and Lescanne (1989).

list of substitutions). For example, after running the goal

```
(exist (x y z) (≠ 5 x) (≠ (5 6) (y z)))
```

the constraint store would be `(((y . 5) (z . 6)) ((x . 5)))`.

We define *empty-c** to be the empty list: (**define** *empty-c** ()). We extend c^* using *cons*.

We must pass the constraint store to every goal. We could add an extra c^* argument to each goal, but instead we pass around the substitution and constraint store as a single value, which we call a *package*. Most goal constructors just pass around the substitution—their definitions need not change. We only need to modify goal constructors that extend or inspect the substitution (such as ≡). (We will use the package abstraction whenever we need to pass around constraint information, such as the freshness constraints of nominal logic in Chapter 11.)

Here are our package constructors and deconstructors[2].

```
(define make-a (λ (s c*) (cons s c*)))
(define s-of (λ (a) (car a)))
(define c*-of (λ (a) (cdr a)))
(define empty-a (make-a empty-s empty-c*))
```

8.2 Solving Disequality Constraints

In this section we will use unification in a clever way to solve disequality constraints after a call to ≠ or ≡, and to keep these constraints in simplified form. First, observe that unifying terms t_1 and t_2 in a substitution s has three possible outcomes:

1. unification can fail, indicating there is no extension to s that will make t_1 and t_2 equal;

2. unification can succeed *without* extending s—this implies that t_1 and t_2 are already equal;

3. unification can succeed, returning an extended substitution containing new associations—in this case, the "mini-substitution" $\hat{s}$ containing only these new

[2] The *s-of* deconstructor, which returns a package's substitution, is all we need to update our definition of the impure operator **project**.

```
(define-syntax project
  (syntax-rules ()
    ((_ (x ...) g g* ...)
     (λG (a)
       (let ((s (s-of a)))
         (let ((x (walk* x s)) ...)
           ((exist () g g* ...) a)))))))
```

associations represents the most general substitution that makes t_1 and t_2 equal[3].

Now let us consider disequality constraints: instead of determining if t_1 and t_2 can be made equal, we wish to determine if t_1 and t_2 can be made *disequal* with respect to s. Fortunately, this requires only a slight change in perspective. We unify t_1 and t_2 with respect to s, but we interpret result of the unification differently:

1. if unification fails, t_1 and t_2 can never be made equal, and the disequality constraint can never be violated—therefore, we can throw the constraint away;

2. if unification succeeds *without* extending s, then t_1 and t_2 are already equal—the disequality constraint has been violated;

3. if unification succeeds and returns an extended substitution containing new associations, then the constraint has not been violated, but could still be violated through future calls to ≡—in this case, the "mini-substitution" $\hat{s}$ that contains the new associations represents the updated disequality constraint in simplified form.

A few examples should clarify how unification can be used to solve disequality constraints.

1. Running the goal (≠ 5 6) corresponds to the first case above: 5 and 6 fail to unify in any substitution, which means the constraint can never be violated. Therefore (≠ 5 6) succeeds, without extending the constraint store.

2. The goal (≠ 5 5) corresponds to the second case above: 5 unifies with itself, without extending the current substitution, which means the disequality constraint has been violated. Therefore (≠ 5 5) fails.

3. The goal (≠ (5 6) (x y)) corresponds to the third case above: (5 6) and (x y) unify in the empty substitution (let's say), resulting in a substitution extended with the associations (x . 5) and (y . 6). This means the constraint was not violated, but could be violated in the future (if x is unified with 5 and y with 6). Therefore (≠ (5 6) (x y)) succeeds, extending the constraint store with the simplified constraint ((x . 5) (y . 6)).

Let us consider a final, more complicated example that uses both ≠ and ≡.

```
(exist (p x y)
  (≠ (5 6) p)
  (≡ (x y) p)
  (≡ 5 x)
  (≡ 7 y))
```

[3] The technical term for this substitution is the *most general unifier* or *mgu*.

Let us assume that we run this goal in the empty package, containing the empty substitution $s =$ () and the empty constraint store $c^* =$ (). First we run the goal (≠ (5 6) p); p unifies with (5 6) in the empty substitution, extending the substitution with the association ((p . (5 6))). Therefore (≠ (5 6) p) succeeds, returning a package with $s = ()$ and $c^* =$ (((p . (5 6)))).

Next we run (≡ (x y) p); p unifies with (x y) in the empty substitution, returning the extended substitution $s =$ ((p . (x y))). But after the successful unification we must verify all of the constraints in the constraint store. We have only the single constraint ((p . (5 6))), which we verify by unifying p and (5 6) in the new substitution ((p . (x y))). This unification succeeds, extending the substitution with the associations (x . 5) and (y . 6). Therefore (≡ (x y) p) succeeds, returning a new package with $s =$ ((p . (x y))) and $c^* =$ (((x . 5) (y . 6))).

Next we run (≡ 5 x); x unifies with 5 in the substitution ((p . (x y))), returning the extended substitution $s =$ ((x . 5) (p . (x y))). Since the unification was successful, we must verify our constraints. We still have only a single constraint, ((x . 5) (y . 6)), which we verify by simultaneously unifying x with 5 and y with 6 in the new substitution $s =$ ((x . 5) (p . (x y))). This unification succeeds, extending s with the association (y . 6). Therefore (≡ 5 x) succeeds, returning a new package with $s =$ ((x . 5) (p . (x y))) and $c^* =$ (((y . 6))).

Finally we run (≡ 7 y); y unifies with 7 in the substitution ((x . 5) (p . (x y))), returning the extended substitution $s =$ ((y . 7) (x . 5) (p . (x y))). We then check the constraint ((y . 6)) by unifying y and 6 in the new substitution; this unification fails, indicating that the constraint can never be violated, and can therefore be discarded. The goal (≡ 7 y) succeeds, as does the entire **exist**, returning the package $s =$ ((y . 7) (x . 5) (p . (x y))) and $c^* = ()$.

Had we replaced the final goal (≡ 7 y) with (≡ 6 y), y and 6 would have succeeded without extending the substitution; the constraint would therefore have been violated, and the entire **exist** would fail.

8.3 Implementing ≠ and ≡

Now that we understand how to solve disequality constraints using unification, we are ready to define ≠. ≠ just unifies its arguments in the current substitution, then passes the result of the unification, along with original package, to *≠-verify*.

```
(define-syntax ≠
  (syntax-rules ()
    ((_ u v)
     (λG (a)
       (≠-verify (unify u v (s-of a)) a)))))
```

≠-verify performs a case analysis on the result of the unification, $\hat{s}$, as described in section 8.2. If unification failed, the constraint cannot be violated; therefore ≠

succeeds, and just returns the package passed to it. Since we are using triangular substitutions, we can use a single *eq?* test to determine if unification succeeded without extending the substitution (the second **cond** clause); if so, the constraint has been violated, and ≠ returns (**mzero**) to indicate failure. Otherwise, unification returned an extended substitution. We therefore call the *prefix-s* helper (below), which returns a mini-substitution c containing only the new associations added during unification. We then construct a new package containing both the extended substitution $\hat{s}$ and the simplified constraint c.

```
(define ≠-verify
  (λ (ŝ a)
    (cond
      ((not ŝ) (unit a))
      ((eq? (s-of a) ŝ) (mzero))
      (else (let ((c (prefix-s ŝ (s-of a))))
              (unit (make-a (s-of a) (cons c (c*-of a))))))))))
```

Here is *prefix-s*, which returns the new associations in s that do not occur in the older substitution $<s$. Our use of triangular substitutions makes it trivial to define *prefix-s*, since the new substitutions always form a prefix of s.

```
(define prefix-s
  (λ (s <s)
    (cond
      ((eq? s <s) empty-s)
      (else (cons (car s) (prefix-s (cdr s) <s))))))
```

We can now define ≡, which must check every constraint in the constraint store after a successful unification. Constraint checking also ensures the constraints are kept in simplified form, making future constraint checking more efficient. This simplified form also simplifies reification[4].

```
(define-syntax ≡
  (syntax-rules ()
    ((_ u v)
     (λG (a)
       (≡-verify (unify u v (s-of a)) a)))))
```

≡-verify is similar to, but slightly more complicated than *≠-verify*, since upon successful unification we need to verify all the constraints in c^*.

[4] We keep each individual constraint in simplified form. However, the constraint store itself is not simplified, and may contain redundant constraints. Determining if a constraint subsumes another is expensive, so we only remove redundant constraints at reification time.

```
(define ≡-verify
  (λ (ŝ a)
    (cond
      ((not ŝ) (mzero))
      ((eq? (s-of a) ŝ) (unit a))
      ((verify-c* (c*-of a) empty-c* ŝ)
       ⇒ (λ (c*) (unit (make-a ŝ c*))))
      (else (mzero)))))
```

verify-c* verifies all the constraints in c^* with respect to the current substitution s, accumulating the verified (and simplified) constraints in $\hat{c}^*$. verify-c* uses unify* (below) to simultaneously unify the left- and right-hand-sides of all the associations within a given constraint.

```
(define verify-c*
  (λ (c* ĉ* s)
    (cond
      ((null? c*) ĉ*)
      ((unify* (car c*) s)
       ⇒ (λ (ŝ)
            (cond
              ((eq? s ŝ) #f)
              (else (let ((c (prefix-s ŝ s)))
                      (verify-c* (cdr c*) (cons c ĉ*) s))))))
      (else (verify-c* (cdr c*) ĉ* s)))))
```

```
(define unify*
  (λ (p* s)
    (cond
      ((null? p*) s)
      ((unify (lhs (car p*)) (rhs (car p*)) s)
       ⇒ (λ (s) (unify* (cdr p*) s)))
      (else #f))))
```

For completeness, here is ≡-no-check[5].

```
(define-syntax ≡-no-check
  (syntax-rules ()
    ((_ u v)
     (λG (a)
       (≡-verify (unify-no-check u v (s-of a)) a)))))
```

[5] We can also define ≠-no-check, which performs *unsound disunification*, allowing circular constraints such as **((**x **. (**x**)))**.

```
(define-syntax ≠-no-check
  (syntax-rules ()
    ((_ u v)
     (λG (a)
       (≠-verify (unify-no-check u v (s-of a)) a)))))
```

Reifying a circular constraint introduced by ≠-no-check can result in divergence.

8.4 Reification

We want our reified constraints to be as concise and readable as possible; we therefore eliminate *irrelevant* constraints, which contain one or more variables that are not themselves reified (see section 7.4). We also remove redundant constraints that are subsumed by other reified constraints. Our subsumption check uses unification and is potentially expensive, so we perform this check only during reification.

A relevant constraint contains no unreified variables. *purify* takes the constraint store c^* and the reified name substitution r (section 3.2), and returns a constraint store containing only relevant constraints.

```
(define purify
  (λ (c* r)
    (cond
      ((null? c*) empty-c*)
      ((anyvar? (car c*) r)
       (purify (cdr c*) r))
      (else (cons (car c*)
                  (purify (cdr c*) r))))))
```

purify calls *anyvar?* on each constraint, which returns #t if the constraint contains a variable that is unassociated in the reified name substitution. (The constraint store is walk^*ed in the package's normal substitution before purification, so that variables associated with ground terms do not affect purification.)

```
(define anyvar?
  (λ (v r)
    (cond
      ((var? v) (var? (walk v r)))
      ((pair? v) (or (anyvar? (car v) r) (anyvar? (cdr v) r)))
      (else #f))))
```

In addition to removing irrelevant constraints, we also want to remove any constraint that is subsumed by another reified constraint. For example, after running the goal (**exist** (*x y*) (≠ (5 6) (*x y*)) (≠ 5 *x*)) the constraint store will be (((*x* . 5)) ((*x* . 5) (*y* . 6))). Although the individual constraints are simplified, the constraint ((*x* . 5)) subsumes the constraint ((*x* . 5) (*y* . 6)) (since it is not possible to violate the latter constraint without also violating the former).

We can determine if a constraint c is subsumed by another constraint $\hat{c}$ through yet another clever use of unification. We use unify^* to perform simultaneous unification of the left- and right-hand-sides of all the associations in $\hat{c}$, with respect to the "substitution" c (see section 8.3); if unify^* succeeds without extending the substitution, then c is subsumed by $\hat{c}$. For example, to determine if the constraint $c =$ ((*x* . 5) (*y* . 6)) is subsumed by $\hat{c} =$ ((*x* . 5)), we unify x and 5 in the substitution ((*x* . 5) (*y* . 6)). This unification succeeds without extending c: therefore, ((*x* . 5) (*y* . 6)) is subsumed by ((*x* . 5)), and can be discarded.

The *subsumed?* predicate returns #t if the constraint c is subsumed by any constraint in c^*.

```
(define subsumed?
  (λ (c c*)
    (and (not (null? c*))
         (or (eq? (unify* (car c*) c) c)
             (subsumed? c (cdr c*))))))
```

rem-subsumed takes a list of *unseen* constraints c^* and previously seen constraints $\hat{c}^*$ (initially empty), and returns a new constraint store containing independent constraints, none of which are subsumed by any other. As *rem-subsumed* cdrs down c^*, it checks if the car of c^* is subsumed by any of the other constraints, either in the rest of the unseen constraints in c^*, or the already seen constraints accumulated in $\hat{c}^*$. If so, the car of c^* is thrown away; otherwise, it is added to the list of already seen constraints.

```
(define rem-subsumed
  (λ (c* ĉ*)
    (cond
      ((null? c*) ĉ*)
      ((or (subsumed? (car c*) ĉ*) (subsumed? (car c*) (cdr c*)))
       (rem-subsumed (cdr c*) ĉ*))
      (else (rem-subsumed (cdr c*) (cons (car c*) ĉ*))))))
```

Here is the updated definition of *reify*, which *walk**s the constraint store in the package's substitution before calling *purify* and *rem-subsumed. reify* returns only the reified value if there are no relevant constraints; otherwise, *reify* returns a list containing the reified value, followed by a tagged list of relevant, and independent, reified constraints.

```
(define reify
  (λ (v a)
    (let ((s (s-of a)))
      (let ((v (walk* v s))
            (c* (walk* (c*-of a) s)))
        (let ((r (reify-s v empty-s)))
          (let ((v (walk* v r))
                (c* (walk* (rem-subsumed (purify c* r) empty-c*) r)))
            (cond
              ((null? c*) v)
              (else (v : (never-equal . c*))))))))))
```

Part III

Nominal Logic

Chapter 9

Techniques II: Nominal Logic

In this chapter we introduce αKanren, which extends core miniKanren with operators for *nominal logic programming*. αKanren was inspired by αProlog (Cheney 2004a; Cheney and Urban 2004) and MLSOS (Lakin and Pitts 2008), and their use of nominal logic (Pitts 2003) to solve a class of problems more elegantly than is possible with conventional logic programming.

Like αProlog and MLSOS, αKanren allows programmers to explicitly manage variable names and bindings, making it easier to write interpreters, type inferencers, and other programs that must reason about scope. αKanren also eases the burden of implementing a language from its structural operational semantics, since the requisite side-conditions can often be trivially encoded in nominal logic.

A standard class of such side conditions is to state that a certain variable name cannot occur free in a particular expression. It is a simple matter to check for free occurrences of a variable name in a fully-instantiated term, but in a logic program the term might contain unbound logic variables. At a later point in the program those variables might be instantiated to terms containing the variable name in question. Also, when the writer of semantics employs the equality symbol, what they really mean is that the two terms are the same *up to α-equivalence*, as in the variable hygiene convention popularized by Barendregt (1984). As functional programmers, we would never quibble with the statement: $\lambda x.x = \lambda y.y$, yet without the implicit assumption that one can rename variables using α-conversion, we would have to forgo this obvious equality. And again, if either expression contains an unbound logic variable, it is impossible to perform a full parallel tree walk to determine if the two expressions are α-equivalent: at least part of the tree walk must be deferred until one or both expressions are fully instantiated.

This chapter is organized as follows. Section 9.1 introduces the αKanren operators, and provides trivial examples of their use. Section 9.2 provides a concise but useful αKanren program that performs capture-avoiding substitution. Section 9.3 presents a second αKanren program: a type inferencer for a subset of Scheme.

9.1 Introduction to αKanren

αKanren extends miniKanren with two additional operators, **fresh** and # (entered as `hash`), and one term constructor, ⋈ (entered as `tie`).

fresh, which syntactically looks like **exist**, introduces new *noms* into its scope. (Noms are also called "names" or "atoms", overloaded terminology which we avoid.) Conceptually, a nom represents a variable name[1]; however, a nom behaves more like a constant than a variable, since it only unifies with itself or with an unassociated variable.

```
(run* (q) (fresh (a) (≡ a a))) ⇒ (_0)
(run* (q) (fresh (a) (≡ a 5))) ⇒ ()
(run* (q) (fresh (a b) (≡ a b))) ⇒ ()
(run* (q) (fresh (b) (≡ b q))) ⇒ (a_0)
```

A reified nom is subscripted in the same fashion as a reified variable, but a is used instead of an underscore (_)—hence the (a_0) in the final example above. **fresh** forms can be nested, which may result in noms being shadowed.

```
(run* (q)
  (exist (x y z)
    (fresh (a)
      (≡ x a)
      (fresh (a b)
        (≡ y a)
        (≡ (x y z a b) q))))) ⇒

((a_0 a_1 _0 a_1 a_2))
```

Here a_0, a_1, and a_2 represent different noms, which will not unify with each other.

⋈ is a *term constructor* used to limit the scope of a nom within a term.

```
(define-syntax ⋈
  (syntax-rules ()
    ((_ a t) (tie a t))))
```

Terms constructed using ⋈ are called *binders*. In the term created by the expression (⋈ a t), all occurrences of the nom a within term t are considered bound. We refer to the term t as the *body* of (⋈ a t), and to the nom a as being in *binding position*. The ⋈ constructor does not create noms; rather, it delimits the scope of noms, already introduced using **fresh**.

[1] Less commonly, a nom may represent a non-variable entity. For example, a nom may represent a channel name in the π-calculus—see Cheney (2004a) for details.

For example, consider this **run*** expression.

```
(run* (q)
  (fresh (a b)
    (≡ (⋈ a (foo a 3 b)) q))) ⇒

((tie a_0 (foo a_0 3 a_1)))
```

The tagged list (tie a_0 (foo a_0 3 a_1)) is the reified value of the term constructed using ⋈. (The tag name tie is a pun—the bowtie ⋈ is the "tie that binds.") The nom whose reified value is a_0 occurs bound within the term (tie a_0 (foo a_0 3 a_1)) while a_1 occurs free in that same term.

# introduces a *freshness constraint* (henceforth referred to as simply a *constraint*). The expression (# a t) asserts that the nom a does *not* occur free in term t—if a occurs free in t, then (# a t) fails. Furthermore, if t contains an unbound variable x, and some later unification involving x results in a occurring free in t, then that unification fails.

```
(run* (q) (fresh (a) (≡ (3 a #t) q) (# a q))) ⇒ ()
(run* (q) (fresh (a) (# a q) (≡ (3 a #t) q))) ⇒ ()
(run* (q) (fresh (a b) (# a (⋈ b a)))) ⇒ ()
(run* (q) (fresh (a) (# a (⋈ a a)))) ⇒ (_0)
(run* (q)
  (exist (x y z)
    (fresh (a)
      (# a x)
      (≡ (y z) x)
      (≡ (x a) q)))) ⇒

((((_0 _1) a_0) : ((a_0 . _0) (a_0 . _1))))
```

In the fourth example, the constraint (# a (⋈ a a)) is not violated because a does not occur free in (⋈ a a). In the final example, the partial instantiation of x causes the constraint introduced by (# a x) to be "pushed down" onto the unbound variables y and z. The answer comprises two parts, separated by a colon and enclosed in an extra set of parentheses: the reified value of ((y z) a) and a list of reified constraints indicating that a cannot occur free in either y or z.

The notion of a constraint is prominent in the standard definition of α-equivalence (Stoy 1979):

$$\lambda a.M \equiv_\alpha \lambda b.[b/a]M \text{ where } b \text{ does not occur free in } M.$$

In αKanren this constraint is expressed as (# b M). We shall revisit the connection between constraints and α-equivalence shortly.

We now extend the standard notion of unification to that of *nominal unification* (Urban et al. 2004), which equates α-equivalent binders. Consider this **run*** expression: (**run*** (q) (**fresh** (a b) (≡ (⋈ a a) (⋈ b b)))) ⇒ ($\_{0}$). Although a and b

are distinct noms, $(\equiv (\bowtie a\ a)\ (\bowtie b\ b))$ succeeds. According to the rules of nominal unification, the binders $(\bowtie a\ a)$ and $(\bowtie b\ b)$ represent the same term, and therefore unify.

The reader may suspect that, as in the definition of α-equivalence given above, nominal unification uses substitution to equate binders

$$(\bowtie a\ a) \equiv_\alpha (\bowtie b\ [b/a]a)$$

however, this is not the case.

Unfortunately, naive substitution does not preserve α-equivalence of terms, as shown in the following example given by Urban et al. (2004). Consider the α-equivalent terms $(\bowtie a\ b)$ and $(\bowtie c\ b)$; replacing all free occurrences of b with a in both terms yields $(\bowtie a\ a)$ and $(\bowtie c\ a)$, which are no longer α-equivalent.

Rather than using capture-avoiding substitution to address this problem, nominal logic uses the simple and elegant notion of a *nom swap*. Instead of performing a uni-directional substitution of a for b, the unifier exchanges all occurrences of a and b within a term, regardless of whether those noms appear free, bound, or in the binding position of a $\bowtie$-constructed binder. Applying the swap $\boldsymbol{(}a\ b\boldsymbol{)}$ to $(\bowtie a\ b)$ and $(\bowtie c\ b)$ yields the α-equivalent terms $(\bowtie b\ a)$ and $(\bowtie c\ a)$.

When unifying $(\bowtie a\ a)$ and $(\bowtie b\ b)$ in the **run*** expression above, the nominal unifier first creates the swap $\boldsymbol{(}a\ b\boldsymbol{)}$ containing the noms in the binding positions of the two terms. The unifier then applies this swap to $(\bowtie a\ a)$, yielding $(\bowtie b\ b)$ (or equivalently, applies the swap to $(\bowtie b\ b)$, yielding $(\bowtie a\ a)$). Obviously $(\bowtie b\ b)$ unifies with itself, according to the standard rules of unification, and thus the nominal unification succeeds.

Of course, the terms being unified might contain unbound variables. In the simple example

$$(\mathbf{run}^*\ (q)\ (\mathbf{fresh}\ (a\ b)\ (\equiv (\bowtie a\ q)\ (\bowtie b\ b)))) \Rightarrow \boldsymbol{(}\mathsf{a}_0\boldsymbol{)}$$

the swap $\boldsymbol{(}a\ b\boldsymbol{)}$ can be applied to $(\bowtie b\ b)$, yielding $(\bowtie a\ a)$. The terms $(\bowtie a\ a)$ and $(\bowtie a\ q)$ are then unified, associating q with a. However, in some cases a swap cannot be performed until a variable has become at least partially instantiated. For example, in the first call to $\equiv$ in

```
(run* (q)
  (fresh (a b)
    (exist (x y)
      (≡ (⋈ a (⋈ a x)) (⋈ a (⋈ b y)))
      (≡ (x y) q))))
```

the unifier cannot apply the swap $\boldsymbol{(}a\ b\boldsymbol{)}$ to either x or y, since they are both unbound. (The unifier does not generate a swap for the outer binders, since they have the same nom in their binding positions.)

Nominal unification solves this problem by introducing the notion of a *suspension*, which is a record of *delayed swaps* that may be applied later. We represent a suspension using the susp data structure, which comprises a list of suspended swaps and a variable.

$$(\mathsf{susp}\ ((a_n\ b_n)\ \ldots\ (a_1\ b_1))\ x)$$

The swaps are deferred until the variable x is instantiated (at least partially); at this point the swaps are applied to the instantiated portion of the term associated with x. Swaps are applied from right to left; that is, the result of applying the swaps to a term t can be determined by first exchanging all occurrences of noms a_1 and b_1 within t, then exchanging a_2 and b_2 within the resulting term, and continuing in this fashion until finally exchanging a_n with b_n.

Now that we have the notion of a suspension, we can define equality on binders (adapted from Urban et al. 2004):

> $(\bowtie\ a\ M)$ and $(\bowtie\ b\ N)$ are α-equivalent if and only if a and b are the same nom and M is α-equivalent to N, or if $(\mathsf{susp}\ ((a\ b))\ M)$ is α-equivalent to N and $(\#\ b\ M)$.

The side condition $(\#\ b\ M)$ is necessary, since if b occurred free in M, then b would be inadvertently captured (and replaced with a) by the suspension $(\mathsf{susp}\ ((a\ b))\ M)$.

Having defined equality on binders, we can examine the result of the previous **run*** expression.

```
(run* (q)
  (fresh (a b)
    (exist (x y)
      (≡ (⋈ a (⋈ a x)) (⋈ a (⋈ b y)))
      (≡ (x y) q)))) ⇒
```

```
((((susp ((a_0 a_1)) _0) _0) : ((a_0 . _0))))
```

The first call to ≡ applies the swap $(a\ b)$ to the unbound variable y, and then associates the resulting suspension $(\mathsf{susp}\ ((a\ b))\ y)$ with x. Of course, the unifier could have applied the swap to x instead of y, resulting in a symmetric answer. The freshness constraint states that the nom a can never occur free within y, as required by the definition of binder equivalence.

Here is a translation of a quiz presented in Urban et al. (2004), demonstrating some of the finer points of nominal unification.

```
(run* (q)
  (fresh (a b)
    (exist (x y)
      (cond^e
        ((≡ (⋈ a (⋈ b (x b))) (⋈ b (⋈ a (a x)))))
        ((≡ (⋈ a (⋈ b (y b))) (⋈ b (⋈ a (a x)))))
        ((≡ (⋈ a (⋈ b (b y))) (⋈ b (⋈ a (a x)))))
        ((≡ (⋈ a (⋈ b (b y))) (⋈ a (⋈ a (a x))))))
      (≡ (x y) q)))) ⇒
```

```
((a_0 a_1)
 (_0 (susp ((a_0 a_1)) _0))
 ((_0 (susp ((a_1 a_0)) _0)) : ((a_1 . _0))))
```

The first **cond**^{e} clause fails, since x cannot be associated with both a and b. The second clause succeeds, associating x with a and y with b. The third clause applies the swap $(a\ b)$ to $(\bowtie a\ (a\ x))$, yielding (tie b (b (susp ((a b)) x))). This term is then unified with $(\bowtie b\ (b\ y))$, associating y with the suspension (susp ((b a)) x). The fourth clause should look familiar—it is similar to the previous **run*** expression.

We can interpret the successful unification of binders $(\bowtie a\ a)$ and $(\bowtie b\ b)$ as showing that the λ-calculus terms $\lambda a.a$ and $\lambda b.b$ are identical, up to α-equivalence. We need not restrict our interpretation to λ terms, however, since other scoping mechanisms have similar properties. For example, the same successful unification also shows that $\forall a.a$ and $\forall b.b$ are equivalent in first-order logic, and similarly, that $\exists a.a$ and $\exists b.b$ are equivalent.

We can tag terms in order to disambiguate their interpretation. For example, this program shows that $\lambda a.\lambda b.a$ and $\lambda c.\lambda d.c$ are equivalent.

```
(run* (q)
  (exist (t u)
    (fresh (a b c d)
      (≡ (lam (tie a (lam (tie b (var a))))) t)
      (≡ (lam (tie c (lam (tie d (var c))))) u)))) ⇒
```

```
(_0)
```

Of course, not all λ-calculus terms are equivalent.

```
(run* (q)
  (exist (t u)
    (fresh (a b c d)
      (≡ (lam (tie a (lam (tie b (var a))))) t)
      (≡ (lam (tie c (lam (tie d (var d))))) u)))) ⇒
```

```
()
```

Here (≡ (lam t_2) (lam u_2)) fails, showing that terms $\lambda a.\lambda b.a$ and $\lambda c.\lambda d.d$ are not α-equivalent.

9.2 Capture-avoiding Substitution

We now consider a simple, but useful, nominal logic program adapted from Cheney and Urban (2004) that performs capture-avoiding substitution (that is, β-substitution). $subst^o$ implements the relation $[new/a]e = out$ where e, new, and out are tagged lists representing λ-calculus terms, and where a is a nom representing a variable name. (We refer the interested reader to Cheney and Urban for a full description of $subst^o$.)

```
(define subst^o
  (λ (e new a out)
    (match^e (e out)
      (((var a) new))
      (((var y) (var y))
       (# a y))
      (((app rator rand) (app rator-res rand-res))
       (subst^o rator new a rator-res)
       (subst^o rand new a rand-res))
      (((lam (tie @c body)) (lam (tie @c body-res)))
       (# c a)
       (# c new)
       (subst^o body new a body-res)))))
```

The first $subst^o$ example shows that $[b/a]\lambda a.ab \equiv_\alpha \lambda c.cb$.

```
(run* (q)
  (fresh (a b)
    (subst^o (lam (tie a (app (var a) (var b)))) (var b) a q))) ⇒
```

```
((lam (tie a_0 (app (var a_0) (var a_1)))))
```

Naive substitution would have produced $\lambda b.bb$ instead.

This second example shows that $[a/b]\lambda a.b \equiv_\alpha \lambda c.a$.

```
(run* (x)
  (fresh (a b)
    (subst^o (lam (tie a (var b))) (var a) b x))) ⇒
```

```
((lam (tie a_0 (var a_1))))
```

Naive substitution would have produced $\lambda a.a$.

9.3 Type Inferencer

Let us consider a second non-trivial αKanren example: a type inferencer for a subset of Scheme[2]. We begin with the typing rule for integer constants, which are tagged with the symbol intc.

```
(define int-rel
  (λ (g exp t)
    (exist (n)
      (≡ (intc n) exp)
      (≡ int t))))
```

[2] This program is an extended and adapted version of the inferencer for the simply-type λ-calculus presented in Cheney and Urban (2004).

The ⊢ relation[3] relates an expression *exp* to its type *t* in the type environment *g*.

```
(define ⊢
  (λ (g exp t)
    (conde
      ((int-rel g exp t)))))
```

We can now infer the types of integer constants: `(run* (q) (⊢ () (intc 5) q))` returns `(int)`.

Inferring the types of integer constants is not very interesting. We therefore add typing rules for variables, λ expressions, and application.

```
(define var-rel
  (λ (g exp t)
    (exist (x)
      (≡ (var x) exp)
      (lookupo x t g))))

(define lambda-rel
  (λ (g exp t)
    (exist (body trand tbody)
      (fresh (a)
        (≡ (lam (⋈ a body)) exp)
        (≡ (→ trand tbody) t)
        (⊢ ((a . trand) . g) body tbody)))))

(define app-rel
  (λ (g exp t)
    (exist (rator rand trand)
      (≡ (app rator rand) exp)
      (⊢ g rator (→ trand t))
      (⊢ g rand trand))))
```

The *lookup*^{o} helper relation finds the type *tx* associated with the type variable *x* in the current type environment *g*.

```
(define lookupo
  (λ (x tx g)
    (exist (a d)
      (≡ (a . d) g)
      (conde
        ((≡ (x . tx) a))
        ((exist (x̂ tx̂)
           (≡ (x̂ . tx̂) a)
           (# x x̂)
           (lookupo x tx d)))))))
```

[3] ⊢ is entered as `!-` and is pronounced “turnstile”.

We redefine ⊢ to include the new typing rules.

```
(define ⊢
  (λ (g exp t)
    (conde
      ((var-rel g exp t))
      ((int-rel g exp t))
      ((lambda-rel g exp t))
      ((app-rel g exp t)))))
```

We can now show that (λ (*x*) (λ (*y*) *x*)) has type ($\alpha \rightarrow (\beta \rightarrow \alpha)$).

```
(run* (q) (⊢ () (parse (λ (x) (λ (y) x))) q)) ⇒ ((→ _0 (→ _1 _0)))
```

Here we use the parser from Appendix E to make the code more readable.

The next example shows that self-application doesn't type check, since the nominal unifier uses the occurs check (Lloyd 1987).

```
(run* (q) (⊢ () (parse (λ (x) (x x))) q)) ⇒ ()
```

This example is more interesting, since it searches for expressions that *inhabit* the type (→ int int).

```
(run5 (q) (⊢ () q (→ int int))) ⇒

((lam (tie a.0 (intc _0)))
 (lam (tie a.0 (var a.0)))
 (lam (tie a.0 (app (lam (tie a.1 (intc _0))) (intc _1))))
 (lam (tie a.0 (app (lam (tie a.1 (intc _0))) (var a.0))))
 (app (lam (tie a.0 (var a.0))) (lam (tie a.1 (intc _0)))))
```

These expressions are equivalent to (in order)

```
(λ (x) n)
(λ (x) x)
(λ (x) ((λ (y) n) m))
(λ (x) ((λ (y) n) x))
((λ (x) x) (λ (y) n))
```

where **n** and **m** are some integer constants. Each expression inhabits the type (int → int), although the principal type of the expression is either ($\alpha \rightarrow \alpha$) (for the identity function) or ($\alpha \rightarrow$ int) (for the remaining expressions).

We now extend the language even further, adding boolean constants, *zero?*, *sub1*, multiplication, **if**-expressions, and a fixed-point operator for defining recursive functions.

```
(define bool-rel
  (λ (g exp t)
    (exist (b)
      (≡ (boolc b) exp)
      (≡ bool t))))
```

```
(define zero?-rel
  (λ (g exp t)
    (exist (e)
      (≡ (zero? e) exp)
      (≡ bool t)
      (⊢ g e int))))

(define sub1-rel
  (λ (g exp t)
    (exist (e)
      (≡ (sub1 e) exp)
      (≡ t int)
      (⊢ g e int))))

 (define ∗-rel
  (λ (g exp t)
    (exist (e1 e2)
      (≡ (∗ e1 e2) exp)
      (≡ t int)
      (⊢ g e1 int)
      (⊢ g e2 int))))

(define if-rel
  (λ (g exp t)
    (exist (test conseq alt)
      (≡ (if test conseq alt) exp)
      (⊢ g test bool)
      (⊢ g conseq t)
      (⊢ g alt t))))

(define fix-rel
  (λ (g exp t)
    (exist (rand)
      (≡ (fix rand) exp)
      (⊢ g rand (→ t t)))))
```

We redefine ⊢ one last time.

```
(define ⊢
  (λ (g exp t)
    (conde
      ((var-rel g exp t))
      ((int-rel g exp t))
      ((bool-rel g exp t))
      ((zero?-rel g exp t))
      ((sub1-rel g exp t))
      ((fix-rel g exp t))
      ((∗-rel g exp t))
      ((lambda-rel g exp t))
      ((app-rel g exp t))
      ((if-rel g exp t)))))
```

We can now infer the type of the factorial function.

```
(run* (q)
  (⊢ () (parse ((fix (λ (!)
                       (λ (n)
                         (if (zero? n)
                             1
                             (∗ (! (sub1 n)) n))))) 5))
        q)) ⇒
```

```
(int)
```

We can also generate pairs of expressions and their types.

```
(run13 (q)
  (exist (exp t)
    (⊢ () exp t)
    (≡ (exp t) q))) ⇒
```

```
(((intc _0) int)
 ((boolc _0) bool)
 ((zero? (intc _0)) bool)
 ((sub1 (intc _0)) int)
 ((zero? (sub1 (intc _0))) bool)
 ((sub1 (sub1 (intc _0))) int)
 ((zero? (sub1 (sub1 (intc _0)))) bool)
 ((sub1 (sub1 (sub1 (intc _0)))) int)
 ((zero? (sub1 (sub1 (sub1 (intc _0))))) bool)
 ((∗ (intc _0) (intc _1)) int)
 ((lam (tie a.0 (intc _0))) (→ _1 int))
 ((zero? (∗ (intc _0) (intc _1))) bool)
 ((lam (tie a.0 (var a.0))) (→ _0 _0)))
```

For example, the last answer shows that the identity function has type $(\alpha \rightarrow \alpha)$.

This ends the introduction to αKanren. For additional simple examples of nominal logic programming, we suggest Cheney and Urban (2008), Cheney (2004a), Cheney and Urban (2004), Urban et al. (2004), and Lakin and Pitts (2008), which are also excellent choices for understanding the theory of nominal logic.

Chapter 10

Applications II: αlean*TAP*

In this chapter we examine a second application of nominal logic programming, a declarative theorem prover for first-order classical logic. We call this prover αlean*TAP*, since it is based on the lean*TAP* (Beckert and Posegga 1995) prover and written in αKanren. Our prover is a relation, without mode restrictions; given a logic variable as the theorem to be proved, αlean*TAP* *generates* valid theorems.

lean*TAP* is a lean tableau-based theorem prover for first-order logic due to Beckert and Posegga (1995). Written in Prolog, it is extremely concise and is capable of a high rate of inference. lean*TAP* uses Prolog's cut (`!`) in three of its five clauses in order to avoid nondeterminism, and uses `copy_term/2` to make copies of universally quantified formulas. Although Beckert and Posegga take advantage of Prolog's unification and backtracking features, their use of the impure cut and `copy_term/2` makes lean*TAP* non-declarative.

In this chapter we translate lean*TAP* from Prolog to impure miniKanren, using $\textbf{match}^a$ to mimic Prolog's cut, and $\textit{copy-term}^o$ to mimic `copy_term/2`. We then show how to eliminate these impure operators from our translation. To eliminate the use of $\textbf{match}^a$, we introduce a tagging scheme that makes our formulas unambiguous. To eliminate the use of $\textit{copy-term}^o$, we use substitution instead of copying terms. Universally quantified formulas are used as templates, rather than instantiated directly; instead of representing universally quantified variables with logic variables, we use the noms of nominal logic. We then use nominal unification to write a substitution relation that replaces quantified variables with logic variables, leaving the original template untouched.

The resulting declarative theorem prover is interesting for two reasons. First, because of the technique used to arrive at its definition: we use declarative substitution rather than $\textit{copy-term}^o$. To our knowledge, there is no method for copying arbitrary terms declaratively. Our solution is not completely general but is useful when a term is used as a template for copying, as in the case of lean*TAP*. Second, because of the flexibility of the prover itself: αlean*TAP* is capable of instantiating non-ground theorems during the proof process, and accepts non-ground *proofs*, as well. Whereas lean*TAP* is fully automated and either succeeds or fails to prove a given theorem, αlean*TAP* can accept guidance from the user in the form of a partially-instantiated proof, regardless of whether the theorem is ground.

We present an implementation of αlean*TAP* in section 10.3 , demonstrating our technique for eliminating cut and `copy_term/2` from lean*TAP*. Our implementation demonstrates our contributions: first, it illustrates a method for eliminating common impure operators, and demonstrates the use of nominal logic for representing formulas in first-order logic; second, it shows that the tableau process can be represented as a relation between formulas and their tableaux; and third, it demonstrates the flexibility of relational provers to mimic the full spectrum of theorem provers, from fully automated to fully dependent on the user.

This chapter is organized as follows. In section 10.1 we describe the concept of tableau theorem proving. In section 10.2 we motivate our declarative prover by examining its declarative properties and the proofs it returns. In section 10.3 we present the implementation of αlean*TAP*, and in section 10.4 we briefly examine αlean*TAP*'s performance. Familiarity with tableau theorem proving would be helpful; for more on this topic, see the references given in section 10.1. In addition, a reading knowledge of Prolog would be useful, but is not necessary; for readers unfamiliar with Prolog, carefully following the miniKanren and αKanren code should be sufficient for understanding all the ideas in this chapter.

10.1 Tableau Theorem Proving

We begin with an introduction to tableau theorem proving and its implementation in lean*TAP*.

Tableau is a method of proving first-order theorems that works by refuting the theorem's negation. In our description we assume basic knowledge of first-order logic; for coverage of this subject and a more complete description of tableau proving, see Fitting (1996). For simplicity, we consider only formulas in Skolemized *negation normal form* (NNF). Converting a formula to this form requires removing existential quantifiers through Skolemization, reducing logical connectives so that only $\wedge$, $\vee$, and $\neg$ remain, and pushing negations inward until they are applied only to literals—see section 3 of Beckert and Posegga (1995) for details.

To form a tableau, a compound formula is expanded into branches recursively until no compound formulas remain. The leaves of this tree structure are referred to as *literals*. lean*TAP* forms and expands the tableau according to the following rules. When the prover encounters a conjunction $x \wedge y$, it expands both x and y on the same branch. When the prover encounters a disjunction $x \vee y$, it splits the tableau and expands x and y on separate branches. Once a formula has been fully expanded into a tableau, it can be proved unsatisfiable if on each branch of the tableau there exist two complementary literals a and $\neg a$ (each branch is *closed*). In the case of propositional logic, syntactic comparison is sufficient to find complementary literals;

in first-order logic, sound unification must be used. A closed tableau represents a proof that the original formula is unsatisfiable.

The addition of universal quantifiers makes the expansion process more complicated. To prove a universally quantified formula $\forall x.M$, lean*TAP* generates a logic variable v and expands M, replacing all occurrences of x with v (i.e., it expands M' where $M' = M[v/x]$). If lean*TAP* is unable to close the current branch after this expansion, it has the option of generating another logic variable and expanding the original formula again. When the prover expands the universally quantified formula $\forall x.F(x) \wedge (\neg F(\mathsf{a}) \vee \neg F(\mathsf{b}))$, for example, $\forall x.F(x)$ must be expanded twice, since x cannot be instantiated to both a and b.

10.2 Introducing αlean*TAP*

We begin by presenting some examples of αlean*TAP*'s abilities, both in proving ground theorems and in generating theorems. We also explore the proofs generated by αlean*TAP*, and show how passing partially-instantiated proofs to the prover can greatly improve its performance.

10.2.1 Running Forwards

Both lean*TAP* and αlean*TAP* can prove ground theorems; in addition, αlean*TAP* produces a proof. This proof is a list representing the steps taken to build a closed tableau for the theorem; Paulson (1999) has shown that translation to a more standard format is possible. Since a closed tableau represents an unsatisfiable formula, such a list of steps proves that the negation of the formula is valid. If the list of steps is ground, the proof search becomes deterministic, and αlean*TAP* acts as a proof checker.

lean*TAP* encodes first-order formulas using Prolog terms. For example, the term `(p(b),all(X,(-p(X);p(s(X)))))` represents $p(\mathsf{b}) \wedge \forall x.\neg p(x) \vee p(s(x))$. In our prover, we represent formulas using Scheme lists with extra tags:

(and (pos (app p (app b))) (forall ($\bowtie$ a (or (neg (app p (var a)))
(pos (app p (app s (var a))))))))

Consider Pelletier Problem 18 (Pelletier 1986): $\exists y.\forall x.F(y) \Rightarrow F(x)$. To prove this theorem in αlean*TAP*, we transform it into the following *negation* of the NNF:

(forall ($\bowtie$ a (and (pos (app f (var a))) (neg (app f (app g_1 (var a)))))))

where (app g_1 (var a)) represents the application of a Skolem function to the universally quantified variable a. Passing this formula to the prover, we obtain the proof (univ conj savefml savefml univ conj close). This proof lists the steps the prover (presented in section 10.3.3) follows to close the tableau. Because both conjuncts of the formula contain the nom a, we must expand the universally quantified formula more than once.

Partially instantiating the proof helps αleanTAP prove theorems with similar subparts. We can create a non-ground proof that describes in general how to prove the subparts and have αleanTAP fill in the trivial differences. This can speed up the search for a proof considerably. By inspecting the negated NNF of Pelletier Problem 21, for example, we can see that there are at least two portions of the theorem that will have the same proof. By specifying the structure of the first part of the proof and constraining the identical portions by using the same logic variable to represent both, we can give the prover some guidance without specifying the whole proof. We pass the following non-ground proof to αleanTAP:

```
(conj univ split (conj savefml savefml conj split x x)
      (conj savefml savefml conj split (close) (savefml split y y)))
```

On our test machine, our prover solves the original problem with no help in 68 milliseconds (ms); given the knowledge that the later parts of the proof will be duplicated, the prover takes only 27 ms. This technique also yields improvement when applied to Pelletier Problem 43: inspecting the negated NNF of the formula, we see two parts that look nearly identical. The first part of the negated NNF—the part representing the theorem itself—has the following form:

```
(and (or (and (neg (app Q (app g4) (app g3)))
              (pos (app Q (app g3) (app g4))))
         (and (pos (app Q (app g4) (app g3)))
              (neg (app Q (app g3) (app g4))))) ...)
```

Since we suspect that the same proof might suffice for both branches of the theorem, we give the prover the partially-instantiated proof (conj split x x). Given just this small amount of help, αleanTAP proves the theorem in 720 ms, compared to 1.5 seconds when the prover has no help at all. While situations in which large parts of a proof are identical are rare, this technique also allows us to handle situations in which different parts of a proof are merely similar by instantiating as much or as little of the proof as necessary.

10.2.2 Running Backwards

Unlike leanTAP, αleanTAP can generate valid theorems. Some interpretation of the results is required since the theorems generated are negated formulas in NNF.[1] In the example

```
(run1 (q) (exist (x) (proveo q () () () x)))
⇒ ((and (pos (app _0)) (neg (app _0))))
```

[1] The full implementation of αleanTAP includes a simple declarative translator from negated NNF to a positive form.

the reified logic variable $\_\_{0}$ represents any first-order formula p, and the entire answer represents the formula $p \wedge \neg p$. Negating this formula yields the original theorem: $\neg p \vee p$, or the law of excluded middle. We can also generate more complicated theorems; here we use the "generate and test" idiom to find the first theorem matching the negated NNF of the inference rule *modus ponens*:

```
(run^1 (q)
  (exist (x)
    (prove^o x () () () q)
    (≡ (and (and (or (neg (app a)) (pos (app b))) (pos (app a))) (neg (app b)))
       x)))
⇒ ((conj conj split (savefml close) (savefml savefml close)))
```

This process takes about 5.1 seconds; *modus ponens* is the 173rd theorem to be generated, and the prover also generates a proof of its validity. When this proof is given to αlean*TAP*, *modus ponens* is the sixth theorem generated, and the process takes only 20 ms.

Thus the declarative nature of αlean*TAP* is useful both for generating theorems and for producing proofs. Due to this flexibility, αlean*TAP* could become the core of a larger proof system. Automated theorem provers like lean*TAP* are limited in the complexity of the problems they can solve, but given the ability to accept assistance from the user, more problems become tractable.

As an example, consider Pelletier Problem 47: Schubert's Steamroller. This problem is difficult for tableau-based provers like lean*TAP* and αlean*TAP*, and neither can solve it automatically (Beckert and Posegga 1995). Given some help, however, αlean*TAP* can prove the Steamroller. Our approach is to prove a series of smaller lemmas that act as stepping stones toward the final theorem; as each lemma is proved, it is added as an assumption in proving the remaining ones. The proof process is automated—the user need only specify which lemmas to prove and in what order. Using this strategy, αlean*TAP* proves the Steamroller in about five seconds; the proof requires twenty lemmas.

αlean*TAP* thus offers an interesting compromise between large proof assistants and smaller automated provers. It achieves some of the capabilities of a larger system while maintaining the lean deduction philosophy introduced by lean*TAP*. Like an automated prover, it is capable of proving simple theorems without user guidance. Confronted with a more complex theorem, however, the user can provide a partially-instantiated proof; αlean*TAP* can then check the proof and fill in the trivial parts the user has left out. Because αlean*TAP* is declarative, the user may even leave required axioms out of the theorem to be proved and have the system derive them. This flexibility comes at no extra cost to the user—the prover remains both concise and reasonably efficient.

The flexibility of αlean*TAP* means that it could be made interactive through the addition of a read-eval-print loop and a simple proof translator between αlean*TAP*'s proofs and a more human-readable format. Since the proof given to αlean*TAP* may be partially instantiated, such an interface would allow the user to conveniently guide αlean*TAP* in proving complex problems. With the addition of equality and the ability to perform single beta steps, this flexibility would become more interesting—in addition to reasoning about programs and proving properties about them, αlean*TAP* would instantiate non-ground programs during the proof process.

10.3 Implementation

We now present the implementation of αlean*TAP*. We begin with a translation of lean*TAP* from Prolog into αKanren. We then show how to eliminate the translation's impure features through a combination of substitution and tagging.

lean*TAP* implements both expansion and closing of the tableau. When the prover encounters a conjunction, it uses its argument `UnExp` as a stack (Figure 10.1): lean*TAP* expands the first conjunct, pushing the second onto the stack for later expansion. If the first conjunct cannot be refuted, the second is popped off the stack and expansion begins again. When a disjunction is encountered, the split in the tableau is reflected by two recursive calls. When a universal quantifier is encountered, the quantified variable is replaced by a new logic variable, and the formula is expanded. The `FreeV` argument is used to avoid replacing the free variables of the formula. lean*TAP* keeps a list of the literals it has encountered on the current branch of the tableau in the argument `Lits`. When a literal is encountered, lean*TAP* attempts to unify its negation with each literal in `Lits`; if any unification succeeds, the branch is closed. Otherwise, the current literal is added to `Lits` and expansion continues with a formula from `UnExp`.

10.3.1 Translation to αKanren

While αKanren is similar to Prolog with the addition of nominal unification, αKanren uses a variant of interleaving depth-first search (Kiselyov et al. 2005), so the order of **cond**^{e} or **match**^{e} clauses in αKanren is irrelevant. Because of Prolog's depth-first search, lean*TAP* must use `VarLim` to limit its search depth; in αKanren, `VarLim` is not necessary, and thus we omit it.

In Figure 10.1 we present mKlean*TAP*, our translation of lean*TAP* into αKanren; we label two clauses (①, ②), since we will modify these clauses later. To express Prolog's cuts, our definition uses **match**^{a}. The final two clauses of lean*TAP* do not contain Prolog cuts; in mKlean*TAP*, they are combined into a single clause containing a **cond**^{e}. In place of lean*TAP*'s recursive call to `prove` to check the membership of `Lit` in `Lits`, we call *member*^{o}, which performs a membership check using sound unification.[2]

[2] We define *member*^{o} in Figure 10.3; *member*^{o} *must* use sound unification, and cannot use $\equiv$*-no-check*.

```
prove((E1,E2),UnExp,Lits,
      FreeV,VarLim) :- !,
  prove(E1,[E2|UnExp],Lits,
        FreeV,VarLim).
prove((E1;E2),UnExp,Lits,
      FreeV,VarLim) :- !,
 prove(E1,UnExp,Lits,FreeV,VarLim),
 prove(E2,UnExp,Lits,FreeV,Varlim).
prove(all(X,Fml),UnExp,Lits,
      FreeV,VarLim) :- !,
  \+ length(FreeV,VarLim),
  copy_term((X,Fml,FreeV),
         (X1,Fml1,FreeV)),
 append(UnExp,[all(X,Fml)],UnExp1),
  prove(Fml1,UnExp1,Lits,
       [X1|FreeV],VarLim).

prove(Lit,_,[L|Lits],_,_) :-
  (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L);
     prove(Lit,[],Lits,_,_)).

        prove(Lit,[Next|UnExp],Lits,
              FreeV,VarLim) :-
        prove(Next,UnExp,[Lit|Lits],
                     FreeV,VarLim).
```

```
(define prove^o
  (λ (fml unexp lits freev)
    (match^a fml

      ((and e1 e2)
       (prove^o e1 (e2 . unexp) lits freev))

      ((or e1 e2)
       (prove^o e1 unexp lits freev)
       (prove^o e2 unexp lits freev))

   ①((forall x body)
       (exist (x1 body1 unexp1)
         (copy-term^o (x body freev)
                      (x1 body1 freev))
         (append^o unexp (fml) unexp1)
         (prove^o body1 unexp1 lits
                  (x1 . freev))))

   ②(fml
       (cond^e
         ((match^a (fml neg)
            (((not neg) neg))
            ((fml (not fml))))
          (member^o neg lits))

       ((exist (next unexp1)
          (≡ (next . unexp1) unexp)
          (prove^o next unexp1 (fml . lits)
                   freev))))))))
```

Figure 10.1: lean*TAP* and mKlean*TAP*: a translation from Prolog to αKanren

10.3.2 Eliminating *copy-term*^{o}

Since *copy-term*^{o} is an impure operator, its use makes *prove*^{o} non-declarative: reordering the goals in the prover can result in different behavior. For example, moving the call to *copy-term*^{o} after the call to *prove*^{o} causes the prover to diverge when given any universally quantified formula. To make our prover declarative, we must eliminate the use of *copy-term*^{o}.

Tagging the logic variables that represent universally quantified variables allows the use of a declarative technique that creates two pristine copies of the original term: one copy may be expanded and the other saved for later copying. Unfortunately, this copying examines the entire body of each quantified formula and instantiates the original term to a potentially invalid formula.

Another approach is to represent quantified variables with symbols or strings. When a new instantiation is needed, a new variable name can be generated, and the new name can be substituted for the old without affecting the original formula. This solution does not destroy the prover's input, but it is difficult to ensure that the provided data is in the correct form declaratively: if the formula to be proved is non-ground, then the prover must generate unique names. If the formula *does* contain these names, however, the prover must *not* generate new ones. This problem can be solved with a declarative preprocessor that expects a logical formula *without* names and puts them in place. If the preprocessor is passed a non-ground formula, it instantiates the formula to the correct form. The requirement of a preprocessor, however, means the prover itself is not declarative.

We use nominal logic to solve the *copy-term*^{o} problem. Nominal logic is a good fit for this problem, as it is designed to handle the complexities of dealing with names and binders declaratively. Since noms represent unique names, we achieve the benefits of the symbol or string approach without the use of a preprocessor. We can generate unique names each time we encounter a universally quantified formula, and use nominal unification to perform the renaming of the quantified variable. If the original formula is uninstantiated, our newly-generated name is unique and is put in place correctly; we no longer need a preprocessor to perform this function.

Using the tools of nominal logic, we can modify mKlean*TAP* to represent universally quantified variables using noms and to perform substitution instead of copying. When the prover reaches a literal, however, it must replace each nom with a logic variable, so that unification may successfully compare literals. To accomplish this, we associate a logic variable with each unique nom, and replace every nom with its associated variable before comparing literals. These variables are generated each time the prover expands a quantified formula.

To implement this strategy, we change our representation of formulas slightly. Instead of representing $\forall x.F(x)$ as (forall x (f x)), we use a nom wrapped in a var tag to represent a variable reference, and the term constructor $\bowtie$ to represent the $\forall$ binder: (forall ($\bowtie$ a (f (var a)))), where a is a nom. The var tag allows us to distinguish noms representing variables from other formulas. We now write a relation *subst-lit*^{o} to perform substitution of logic variables for tagged noms in a literal, and we modify the literal case of *prove*^{o} to use it. We also replace the clause handling forall formulas and define *lookup*^{o}. The two clauses of *lookup*^{o} overlap, but since each mapping in the environment is from a unique nom to a logic variable, a particular nom will never appear twice.

We present the changes needed to eliminate *copy-term*^{o} from mKlean*TAP* in Figure 10.2. Instead of copying the body of each universally quantified formula, we generate a logic variable x and add an association between the nom representing the quantified variable and x to the current environment. When we prepare to close a branch of the tableau, we call *subst-lit*^{o}, replacing the noms in the current literal with their associated logic variables.

```
①((forall (tie @a body))
   (exist (x unexp1)
     (append^o unexp (fml) unexp1)
     (prove^o body unexp1 lits
              ((a . x) . env))))

②(fml
  (exist (lit)
    (subst-lit^o fml env lit)
    (cond^e
      ((match^a (lit neg)
         (((not neg) neg))
         ((lit (not lit))))
       (member^o neg lits))
      ((exist (next unexp1)
         (≡ (next . unexp1) unexp)
         (prove^o next unexp1 (lit . lits)
                  env))))))
```

```
(define lookup^o
  (λ (a env out)
    (match^e env
      (((a . out) . rest))
      ((first . rest)
       (lookup^o a rest out)))))

(define subst-lit^o
  (λ (fml env out)
    (match^a (fml out)
      (((var a) out)
       (lookup^o a env out))
      (((e1 . e2) (r1 . r2))
       (subst-lit^o e1 env r1)
       (subst-lit^o e2 env r2))
      ((fml fml)))))
```

Figure 10.2: Changes to mKlean*TAP* to eliminate *copy-term^o*

The original `copy_term/2` approach used by lean*TAP* and mKlean*TAP* avoids replacing free variables by copying the list (*x body freev*). The copied version is unified with the list (x_1 $body_1$ *freev*), so that *only* the variable x will be replaced by a new logic variable—the free variables will be copied, but those copies will be unified with the original variables afterwards. Since our substitution strategy does not affect free variables, the *freev* argument is no longer needed, and so we have eliminated it.

10.3.3 Eliminating match^a

Both *prove^o* and *subst-lit^o* use **match^a** because the clauses that recognize literals overlap with the other clauses. To solve this problem, we have designed a tagging scheme that ensures that the clauses of our substitution and *prove^o* relations do not overlap. To this end, we tag both positive and negative literals, applications, and variables. Constants are represented by applications of zero arguments. Our prover thus accepts formulas of the following form:

Fml → (and *Fml Fml*) | (or *Fml Fml*) | (forall (⋈ *Nom Fml*)) | *Lit*
Lit → (pos *Term*) | (neg *Term*)
Term → (var *Nom*) | (app *Symbol Term**)

This scheme has been chosen carefully to allow unification to compare literals. In particular, the tags on variables *must* be discarded before literals are compared. Consider the two non-ground literals (not (f x)) and (f (p y)). These literals are complementary: the negation of one unifies with the other, associating x with (p y). When we apply our tagging scheme, however, these literals become (neg (app f (var x))) and (pos (app f (app p (var y)))), respectively, and are no longer complementary: their subexpressions (var x) and (app p (var y)) do not unify. To avoid this problem, our substitution relation discards the var tag when it replaces noms with logic variables.

```
(define prove^o
  (λ (fml unexp lits env proof)
    (match^e (fml proof)
      (((and e1 e2) (conj . prf))
       (prove^o e1 (e2 . unexp)
                lits env prf))
      (((or e1 e2) (split prf1 prf2))
       (prove^o e1 unexp lits env prf1)
       (prove^o e2 unexp lits env prf2))
      (((forall (tie @a body)) (univ . prf))
       (exist (x unexp1)
         (append^o unexp (fml) unexp1)
         (prove^o body unexp1 lits
                  ((a . x) . env) prf)))
      ((fml proof)
       (exist (lit)
         (subst-lit^o fml env lit)
         (cond^e
           ((≡ (close) proof)
            (match^e (lit neg)
              (((pos tm) (neg tm)))
              (((neg tm) (pos tm))))
            (member^o neg lits))
           ((exist (next unexp1 prf)
              (≡ (next . unexp1) unexp)
              (≡ (savefml . prf) proof)
              (prove^o next unexp1 (lit . lits)
                       env prf)))))))))
```

```
(define append^o
  (λ^e (ls s out)
    ((() s s))
    (((a . d) s (a . r))
     (append^o d s r))))

(define subst-lit^o
  (λ^e (fml env out)
    (((pos l) env (pos r))
     (subst-term^o l env r))
    (((neg l) env (neg r))
     (subst-term^o l env r))))

(define subst-term^o
  (λ^e (fml env out)
    (((var a) env out)
     (lookup^o a env out))
    (((app f . d) env (app f . r))
     (subst-term*^o d env r))))

(define subst-term*^o
  (λ^e (tm* env out)
    ((() _ ()))
    (((e1 . e2) env (r1 . r2))
     (subst-term^o e1 env r1)
     (subst-term*^o e2 env r2))))

(define member^o
  (λ (x ls)
    (exist (a d)
      (≡ (a . d) ls)
      (cond^e
        ((≡ a x))
        ((member^o x d))))))
```

Figure 10.3: Final definition of αleanTAP

Given our new tagging scheme, we can easily rewrite our substitution relation without the use of **match**^{a}. We simply follow the production rules of the grammar, defining a relation to recognize each.

Finally, we modify *prove*^{o} to take advantage of the same tags. We also add a *proof* argument to *prove*^{o}. We call this version of the prover αleanTAP, and present its definition in Figure 10.3. It is declarative, since we have eliminated the use of *copy-term*^{o} and every use of **match**^{a}. In addition to being a sound and complete theorem prover for first-order logic, αlean*TAP* can now generate valid first-order theorems.

10.4 Performance

Like the original lean*TAP*, αlean*TAP* can prove many theorems in first-order logic. Because it is declarative, αlean*TAP* is generally slower at proving ground theorems than mKlean*TAP*, which is slower than the original lean*TAP*. Figure 10.4 presents a summary of αlean*TAP*'s performance on the first 46 of Pelletier's 75 problems (Pelletier 1986), showing it to be roughly twice as slow as mKlean*TAP*.

These performance numbers suggest that while there is a penalty to be paid for declarativeness, it is not so severe as to cripple the prover. The advantage mKlean*TAP* enjoys over the original lean*TAP* in Problem 34 is due to αKanren's interleaving search strategy; as the result for mKlean*TAP* shows, the original lean*TAP* is faster than αlean*TAP* for any given search strategy.

Many automated provers now use the TPTP problem library (Sutcliffe and Suttner 1998) to assess performance. Even though it is faster than αlean*TAP*, lean*TAP* solves few of the TPTP problems. The Pelletier Problems, on the other hand, fall into the class of theorems lean*TAP* was designed to prove, and so we feel they provide a better set of tests for the comparison between lean*TAP* and αlean*TAP*.

10.5 Applicability of These Techniques

To avoid the use of *copy-term*^{o}, we have represented universally quantified variables with noms rather than logic variables, allowing us to perform substitution instead of copying. To eliminate **match**^{a}, we have enhanced the tagging scheme for representing formulas.

Both of these transformations are broadly applicable. When **match**^{a} is used to handle overlapping clauses, a carefully crafted tagging scheme can often be used to eliminate overlapping. When terms must be copied, substitution can often be used instead of *copy-term*^{o}—in the case of αlean*TAP*, we use a combination of nominal unification and substitution.

#	lean*TAP*	mKlean*TAP*	αlean*TAP*
1	0.1	0.7	2.0
2	0.0	0.1	0.3
3	0.0	0.2	0.5
4	0.0	1.0	1.7
5	0.1	1.2	2.5
6	0.0	0.1	0.2
7	0.0	0.1	0.2
8	0.0	0.3	0.8
9	0.1	4.3	9.7
10	0.3	5.5	10.2
11	0.0	0.3	0.6
12	0.6	17.7	31.9
13	0.1	3.7	8.2
14	0.1	4.2	9.7
15	0.0	0.8	1.9
16	0.0	0.2	0.6
17	1.1	9.2	18.1
18	0.1	0.5	1.2
19	0.3	15.1	33.5
20	0.5	8.1	12.7
21	0.4	22.1	38.7
22	0.1	3.4	6.4
23	0.1	2.5	5.4
24	1.7	31.9	60.3
25	0.2	7.5	14.1
26	0.8	130.9	187.5
27	2.3	40.4	79.3
28	0.3	19.1	29.6
29	0.1	27.9	57.0
30	0.1	4.2	9.6
31	0.3	13.2	23.1
32	0.2	23.9	42.4
33	0.1	15.9	39.2
34	199129.0	7272.9	8493.5
35	0.1	0.5	1.1
36	0.2	6.7	12.4
37	0.8	123.3	169.2
38	8.9	4228.8	8363.8
39	0.0	1.1	2.8
40	0.2	8.1	19.2
41	0.1	6.9	17.0
42	0.4	15.0	32.1
43	43.2	668.4	1509.6
44	0.3	15.1	35.7
45	3.4	145.3	239.7
46	7.7	505.5	931.2

Figure 10.4: Performance of lean*TAP*, mKlean*TAP*, and αlean*TAP* on the first 46 Pelletier Problems. All times are in milliseconds, averaged over 100 trials. All tests were run under Debian Linux on an IBM Thinkpad X40 with a 1.1GHz Intel Pentium-M processor and 768MB RAM. lean*TAP* tests were run under SWI-Prolog 5.6.55; mKlean*TAP* and αlean*TAP* tests were run under Ikarus Scheme 0.0.3+.

Chapter 11

Implementation IV: αKanren

In this chapter we present two implementations of αKanren based on two implementations of nominal unification: one using idempotent substitutions, and one using triangular substitutions. The idempotent implementation mirrors the mathematical description of nominal unification given by Urban et al. (2004), while the triangular implementation is more efficient.

This chapter is organized as follows. In section 11.1 we present our implementation of nominal unification using idempotent substitutions. In section 11.2 we implement αKanren's goal constructors, using the unifier of section 11.1, and in section 11.3 we implement reification. In section 11.4 we present a second implementation of nominal unification, using triangular substitutions.

11.1 Nominal Unification with Idempotent Substitutions

Nominal unification occurs in two distinct phases: the first processes equations, while the second processes constraints. The first phase takes a set of equations ϵ and transforms it into a substitution σ and a set of unresolved constraints δ. The second phase combines the unresolved constraints with the previously resolved constraints, which have both been brought up to date using *apply-subst*. Then, the unifier transforms these combined constraints into a set of resolved constraints ∇, and returns the list $(\sigma\ \nabla)$ as a package.

Nominal unification uses several data structures. A set of equations ϵ is represented as a list of pairs of terms. A substitution σ is represented as an association list of variables to terms. A set of constraints δ is represented as a list of pairs associating noms to terms; a ∇ is a δ in which all terms are unbound variables. In a substitution, a variable may have at most one association. In a δ (and therefore in a ∇) a nom may have multiple associations.

We represent a variable as a suspension containing an empty list of swaps. Several functions reconstruct suspensions that represent variables. However, our implementation of nominal unification assumes that variables can be compared using *eq?*.

In order to ensure that a variable is always *eq?* to itself, regardless of how many times it is reconstructed, we use a **letrec** trick: a suspension representing a variable contains a procedure of zero arguments (a *thunk*) that, when invoked, returns the suspension, thus maintaining the desired *eq?*-ness property. (In the text we conflate variables with their associated thunks.)

```
(define var
  (λ (ignore)
    (letrec ((s (list susp () (λ () s))))
      s)))
```

unify attempts to solve a set of equations ϵ in the context of a package **(**σ ∇**)**. *unify* applies σ to ϵ, and then calls *apply-σ-rules* on the resulting set of equations. *apply-σ-rules* either successfully completes the first phase of nominal unification by returning a new σ and δ, or invokes the failure continuation *fk*, a jump-out continuation similar to Lisp's *catch* (Steele Jr. 1990).

```
(define unify
  (λ (ϵ σ ∇ fk)
    (let ((ϵ (apply-subst σ ϵ)))
      (mv-let ((σ̂ δ) (apply-σ-rules ϵ fk))
        (unify# δ (compose-subst σ σ̂) ∇ fk)))))
```

mv-let, defined in Appendix B, deconstructs a list of values.

In the second phase of nominal unification, *unify#* calls *apply-subst* to bring ∇ and δ up to date, then passes their union to *apply-∇-rules.*

```
(define unify#
  (λ (δ σ ∇ fk)
    (let ((δ (apply-subst σ δ))
          (∇ (apply-subst σ ∇)))
      (let ((δ (δ-union ∇ δ)))
        (list σ (apply-∇-rules δ fk))))))
```

apply-σ-rules is a recursive function whose only task is to combine results returned by *σ-rules.* *σ-rules* takes two arguments: a single equation and the rest of the equations. If *σ-rules* fails, then *apply-σ-rules* invokes *fk*, and the result of *unify* is #f. Each successful call to *σ-rules* returns a new set of equations ϵ, a new σ, and a set of (unresolved) constraints δ. Successive calls to *σ-rules* resolve the equations in ϵ until there are no equations left.

```
(define apply-σ-rules
  (λ (ϵ fk)
    (cond
      ((null? ϵ) (empty-σ empty-δ))
      (else
       (let ((eqn (car ϵ)) (ϵ (cdr ϵ)))
         (mv-let ((ϵ σ δ) (or (σ-rules eqn ϵ) (fk)))
           (mv-let ((σ̂ δ̂) (apply-σ-rules ϵ fk))
             (list (compose-subst σ σ̂) (δ-union δ̂ δ))))))))))
```

apply-∇-rules is similar to *apply-σ-rules*, but takes constraints instead of equations, and combines the results returned by *∇-rules*.

```
(define apply-∇-rules
  (λ (δ fk)
    (cond
      ((null? δ) empty-∇)
      (else
       (let ((c (car δ)) (δ (cdr δ)))
         (mv-let ((δ ∇) (or (∇-rules c δ) (fk)))
           (δ-union ∇ (apply-∇-rules δ fk))))))))
```

empty-σ, *empty-δ*, and *empty-∇* are defined in section 11.2.

In both *σ-rules* and *∇-rules* we use *untagged?* to distinguish untagged pairs from specially tagged pairs that represent binders, noms, and suspensions.

```
(define untagged?
  (λ (x)
    (not (memv x (tie nom susp)))))
```

Here are the transformation rules of the nominal unification algorithm, derived from the rules in Urban et al. (2004). (*σ-rules* relies on **pmatch**, which is defined in Appendix B.)

```
(define σ-rules
  (λ (eqn ϵ)
    (pmatch eqn
      ((c . ĉ)
       (guard (not (pair? c)) (equal? c ĉ))
       (ϵ empty-σ empty-δ))
      (((tie a t) . (tie â t̂))
       (guard (eq? a â))
       (((t . t̂) . ϵ) empty-σ empty-δ))
      (((tie a t) . (tie â t̂))
       (guard (not (eq? a â)))
       (let ((û (apply-π ((a â)) t̂)))
         (((t . û) . ϵ) empty-σ ((a . t̂)))))
      (((nom _) . (nom _))
       (guard (eq? (car eqn) (cdr eqn)))
       (ϵ empty-σ empty-δ))
      (((susp π x) . (susp π̂ x̂))
       (guard (eq? (x) (x̂)))
       (let ((δ (map (λ (a) (cons a (x)))
                     (disagreement-set π π̂))))
         (ϵ empty-σ δ)))
      (((susp π x) . t)
       (guard (not (occurs√ (x) t)))
       (let ((x (x)) (t (apply-π (reverse π) t)))
         (let ((σ ((x . t))))
           (list (apply-subst σ ϵ) σ empty-δ))))
      ((t . (susp π x))
       (guard (not (occurs√ (x) t)))
       (let ((x (x)) (t (apply-π (reverse π) t)))
         (let ((σ ((x . t))))
           (list (apply-subst σ ϵ) σ empty-δ))))
      (((t1 . t2) . (t̂1 . t̂2))
       (guard (untagged? t1) (untagged? t̂1))
       (((t1 . t̂1) (t2 . t̂2) . ϵ) empty-σ empty-δ))
      (else #f))))
```

Clauses two and three in *σ-rules* implement α-equivalence of binders, as defined in section 9.1 of Chapter 9. Clause five unifies two suspensions that have the same variable; in this case, *σ-rules* creates as many new freshness constraints as there are noms in the *disagreement set* (defined below) of the suspensions' swaps. Clauses six and seven are similar: each clause unifies a suspension containing a variable x and a list of swaps π with a term t. *σ-rules* creates a substitution associating x with the result of applying the swaps in π to t in *reverse order*, with the newest swap in π applied first. This substitution is applied to the context ϵ.

apply-π, below, applies a list of swaps π to a term v.

```
(define apply-π
  (λ (π v)
    (pmatch v
      (c (guard (not (pair? c))) c)
      ((tie a t)
       (let ((a (apply-π π a))
             (t (apply-π π t)))
         (tie a t)))
      ((nom _)
       (let loop ((v v) (π π))
         (if (null? π)
             v
             (apply-swap (car π) (loop v (cdr π))))))
      ((susp π̂ x)
       (let ((π (append π π̂)))
         (if (null? π)
             (x)
             (susp π x))))
      ((a . d) (cons (apply-π π a) (apply-π π d))))))
```

If v is a nom, then π's swaps are applied, with the oldest swap applied first. If v is a suspension with a list of swaps $\hat{\pi}$ and variable x, then the swaps in π are added to the swaps in $\hat{\pi}$. If this list is empty, then x's suspension is returned; otherwise, a new suspension is created with those swaps.

```
(define apply-swap
  (λ (swap a)
    (pmatch swap
      ((a₁ a₂)
       (cond
         ((eq? a a₂) a₁)
         ((eq? a a₁) a₂)
         (else a))))))
```

The ∇*-rules* are much simpler than the σ*-rules.* In the second clause, the nom $\hat{a}$ in the binding position of the binder is the same as a, so a can never appear free in t. In the fifth clause, the list of swaps π in the suspension are applied, in reverse order, to the nom a, yielding another nom. ∇*-rules* then adds a new constraint associating this nom with the suspension's variable.

```
(define ∇-rules
  (λ (d δ)
    (pmatch d
      ((a . c)
       (guard (not (pair? c)))
       (δ empty-∇))
      ((a . (tie â t))
       (guard (eq? â a))
       (δ empty-∇))
      ((a . (tie â t))
       (guard (not (eq? â a)))
       (((a . t) . δ) empty-∇))
      ((a . (nom _))
       (guard (not (eq? a (cdr d))))
       (δ empty-∇))
      ((a . (susp π x))
       (let ((a (apply-π (reverse π) a)) (x (x)))
         (δ ((a . x)))))
      ((a . (t1 . t2))
       (guard (untagged? t1))
       (((a . t1) (a . t2) . δ) empty-∇))
      (else #f))))
```

Finding the disagreement set of two lists of swaps π and $\hat{\pi}$ requires forming a set of all the noms in those lists, then applying both π and $\hat{\pi}$ to each nom a in this set. If (*apply-*π π a) and (*apply-*π $\hat{\pi}$ a) produce different noms, then a is in the *dis*agreement set. (*filter* and *remove-duplicates* are defined in Appendix A.)

```
(define disagreement-set
  (λ (π π̂)
    (filter
      (λ (a) (not (eq? (apply-π π a) (apply-π π̂ a))))
      (remove-duplicates
        (append (apply append π) (apply append π̂))))))
```

The *occurs*√ is what one might expect.

```
(define occurs√
  (λ (x v)
    (pmatch v
      (c (guard (not (pair? c))) #f)
      ((tie _ t) (occurs√ x t))
      ((nom _) #f)
      ((susp _ x̂) (eq? (x̂) x))
      ((x̂ . ŷ) (or (occurs√ x x̂) (occurs√ x ŷ)))
      (else #f))))
```

11.1.1 Idempotent Substitutions

compose-subst's definition is taken from Lloyd (1987). It takes two substitutions σ and τ, and constructs a new substitution $\hat{\sigma}$ in which each association **(**x **.** v**)** in σ is replaced by **(**x **.** $\hat{v}$**)**, where $\hat{v}$ is the result of applying τ to v. Any association in τ whose variable has an association in $\hat{\sigma}$ is then filtered from τ. Also, any association of the form **(**x **.** x**)** is filtered from $\hat{\sigma}$. These filtered substitutions are then appended.

```
(define compose-subst
  (λ (σ τ)
    (let ((σ̂ (map
               (λ (a) (cons (car a) (apply-subst τ (cdr a))))
               σ)))
      (append
        (filter (λ (a) (not (assq (car a) σ̂))) τ)
        (filter (λ (a) (not (eq? (car a) (cdr a)))) σ̂)))))
```

Next we define *apply-subst*. In the suspension case, *apply-subst* applies the list of swaps π to a variable, or to its binding.

```
(define apply-subst
  (λ (σ v)
    (pmatch v
      (c (guard (not (pair? c))) c)
      ((tie a t)
       (let ((t (apply-subst σ t)))
         (tie a t)))
      ((nom _) v)
      ((susp π x) (apply-π π (get (x) σ)))
      ((x . y) (cons (apply-subst σ x) (apply-subst σ y))))))
```

get, which is defined in Appendix A, finds the binding of a variable in a substitution or returns the variable if no binding exists.

11.1.2 *δ-union*

Finally we define *δ-union*, which forms the union of two δ's.

```
(define δ-union
  (λ (δ δ̂)
    (pmatch δ
      (() δ̂)
      ((d . δ)
       (if (term-member? d δ̂)
           (δ-union δ δ̂)
           (cons d (δ-union δ δ̂)))))))
```

```
(define term-member?
  (λ (v v*)
    (pmatch v*
      (() #f)
      ((v̂ . v*)
       (or (term-equal? v̂ v) (term-member? v v*))))))
```

```
(define term-equal?
  (λ (u v)
    (pmatch (u v)
      ((c ĉ) (guard (not (pair? c)) (not (pair? ĉ)))
       (equal? c ĉ))
      (((tie a t) (tie â t̂))
       (and (eq? a â) (term-equal? t t̂)))
      (((nom _) (nom _)) (eq? u v))
      (((susp π x) (susp π̂ x̂))
       (and (eq? (x) (x̂)) (null? (disagreement-set π π̂))))
      (((x . y) (x̂ . ŷ))
       (and (term-equal? x x̂) (term-equal? y ŷ)))
      (else #f))))
```

Recall that δ denotes a set of unresolved constraints, where a constraint is a pair of a nom a and a term t. *δ-union* uses *term-member?*, which uses *term-equal?* when comparing two constraints. The definition of *term-equal?* is straightforward except when comparing two suspensions, in which case their variables must be the same, and the disagreement set of their lists of swaps must be empty.

11.2 Goal Constructors

In the core miniKanren implementation of Chapter 3, a goal is a function that maps a substitution s to an ordered sequence of zero or more substitutions (see section 3.3). In αKanren, a goal g is a function that maps a package p to an ordered sequence p^∞ of zero or more packages.

We represent the empty substitution, along with the empty unresolved and resolved constraint sets, as the empty list.

```
(define empty-σ ())       (define empty-δ ())       (define empty-∇ ())
```

≡ and # construct goals that return either a singleton stream or an empty stream.

```
(define-syntax ≡
  (syntax-rules ()
    ((_ u v)
     (unifier unify ((u . v))))))
```

```
(define-syntax #
  (syntax-rules ()
    ((_ a t)
     (unifier unify# ((a . t))))))

(define unifier
  (λ (fn set)
    (λG (p)
      (mv-let ((σ ∇) p)
        (call/cc (λ (fk) (fn set σ ∇ (λ () (fk #f)))))))))
```

The goal constructor **fresh** is identical to **exist**, except that it lexically binds noms instead of variables.

```
(define-syntax fresh
  (syntax-rules ()
    ((_ (a ...) g0 g ...)
     (λG (p)
       (inc
         (let ((a (nom a)) ...)
           (bind* (g0 p) g ...)))))))

(define nom
  (λ (a)
    (list nom (symbol→string a))))
```

11.3 Reification

As described in section 3.2, *reification* is the process of turning a miniKanren (or αKanren) value into a Scheme value.

αKanren's version of *reify* takes a variable x and a package p, and returns the value associated with x in p (along with any relevant constraints), first replacing all variables and noms with symbols representing those entities. A constraint $(a . y)$ is *relevant* if both a and y appear in the value associated with x.

The first **cond** clause in the definition of *reify* below returns only the reified value associated with x, when there are no relevant constraints. The **else** clause returns both the reified value of x and the reified set of relevant constraints; we have arbitrarily chosen the colon ':' to separate the reified value from the list of reified constraints.

```
(define reify
  (λ (x p)
    (mv-let ((σ ∇) p)
      (let* ((v (get x σ)) (s (reify-s v)) (v (walk* v s)))
        (let ((∇ (filter (λ (a) (and (symbol? (car a)) (symbol? (cdr a))))
                   (walk* ∇ s))))
          (cond
            ((null? ∇) v)
            (else (v : ∇))))))))
```

reify-s is the heart of the reifier. *reify-s* takes an arbitrary value *v*, and returns a substitution that maps every distinct nom and variable in *v* to a unique symbol. The trick to maintaining left-to-right ordering of the subscripts on these symbols is to process *v* from left to right, as can be seen in the last **pmatch** clause. When *reify-s* encounters a nom or variable, it determines if we already have a mapping for that entity. If not, *reify-s* extends the substitution with an association between the nom or variable and a new, appropriately subscripted symbol.

```
(define reify-s
  (letrec
    ((r-s (λ (v s)
            (pmatch v
              (c (guard (not (pair? c))) s)
              ((tie a t) (r-s t (r-s a s)))
              ((nom n)
               (cond
                 ((assq v s) s)
                 ((assp nom? s)
                  ⇒ (λ (p)
                       (let ((n (reify-n (cdr p))))
                         (cons (v . n) s))))
                 (else (cons (v . a.0) s))))
              ((susp () _)
               (cond
                 ((assq v s) s)
                 ((assp var? s)
                  ⇒ (λ (p)
                       (let ((n (reify-n (cdr p))))
                         (cons (v . n) s))))
                 (else (cons (v . _.0) s))))
              ((susp π x)
               (r-s (x) (r-s π s)))
              ((a . d) (r-s d (r-s a s)))))))
     (λ (v)
       (r-s v ()))))
```

*walk** applies a special substitution *s*, which maps noms and variables to symbols, to an arbitrary value *v*.

```
(define walk*
  (λ (v s)
    (pmatch v
      (c (guard (not (pair? c))) c)
      ((tie a t) (list tie (get a s) (walk* t s)))
      ((nom _) (get v s))
      ((susp () _) (get v s))
      ((susp π x) (list susp (walk* π s) (get (x) s)))
      ((a . d) (cons (walk* a s) (walk* d s))))))
```

```
(define var?
  (λ (x)
    (pmatch x
      ((susp () __) #t)
      (else #f))))
```

```
(define nom?
  (λ (x)
    (pmatch x
      ((nom __) #t)
      (else #f))))
```

reify-n returns a symbol representing an individual variable or nom; this symbol always ends with a period followed by a non-negative integer.

```
(define reify-n
  (λ (a)
    (let ((str* (string→list (symbol→string a))))
      (let ((c* (memv #\. str*)))
        (let ((rn (string→number (list→string (cdr c*)))))
          (let ((n-str (number→string (+ rn 1))))
            (string→symbol
              (string-append
                (string (car str*)) "." n-str))))))))
```

11.4 Nominal Unification with Triangular Substitutions

In this section we modify the idempotent nominal unification implementation to work with triangular substitutions, significantly improving the performance of αKanren[1]. We present only the definitions that differ from those already presented.

Like the core miniKanren implementation of Chapter 3, our triangular unifier relies on a *walk* function for looking up values in a triangular substitution. The nominal *walk* function is complicated by the need to handle suspensions and permutations.

```
(define walk
  (λ (x s)
    (let loop ((x x) (π ()))
      (pmatch x
        ((susp π̂ v)
         (let ((v (assq (v) s)))
           (cond
             (v (loop (cdr v) (append π̂ π)))
             (else (apply-π π x)))))
        (else (apply-π π x))))))
```

[1] This implementation of triangular nominal unification is due to Joseph Near. Ramana Kumar has implemented a somewhat faster triangular unifier; however, the resulting code bears little resemblance to the idempotent algorithm of (Urban et al. 2004).

We can now redefine *walk** in terms of *walk*.

```
(define walk*
  (λ (v s)
    (let ([v (walk v s)])
      (pmatch v
        ((tie a t) (list tie a (walk* t s)))
        ((a . d) (guard (untagged? a))
         (cons (walk* a s) (walk* d s)))
        (else v)))))
```

unify no longer uses *compose-subst* or *apply-subst*.

```
(define unify
  (λ (ϵ σ ∇ fk)
    (mv-let ((σ̂ δ) (apply-σ-rules ϵ σ fk))
      (unify# δ σ̂ ∇ fk))))
```

Similarly, *unify#* no longer uses *apply-subst*.

```
(define unify#
  (λ (δ σ ∇ fk)
    (let ((δ (δ-union ∇ δ)))
      (list σ (apply-∇-rules δ σ fk)))))
```

apply-σ-rules now takes σ as an additional argument, which it passes to *σ-rules*; also, *apply-σ-rules* no longer uses *compose-subst*.

```
(define apply-σ-rules
  (λ (ϵ σ fk)
    (cond
      ((null? ϵ) (σ empty-δ))
      (else
       (let ((eqn (car ϵ)) (ϵ (cdr ϵ)))
         (mv-let ((ϵ σ δ) (or (σ-rules eqn σ ϵ) (fk)))
           (mv-let ((σ̂ δ̂) (apply-σ-rules ϵ σ fk))
             (list σ̂ (δ-union δ̂ δ)))))))))
```

apply-∇-rules also takes σ as an additional argument, which it passes to *∇-rules*.

```
(define apply-∇-rules
  (λ (δ σ fk)
    (cond
      ((null? δ) empty-∇)
      (else
       (let ((c (car δ)) (δ (cdr δ)))
         (mv-let ((δ ∇) (or (∇-rules c σ δ) (fk)))
           (δ-union ∇ (apply-∇-rules δ σ fk))))))))
```

σ-rules no longer uses *apply-subst*, but now walks ϵ in σ, which is passed in as an additional argument.

```
(define σ-rules
  (λ (eqn σ ϵ)
    (let ((eqn (cons (walk (car eqn) σ) (walk (cdr eqn) σ))))
      (pmatch eqn
        ((c . ĉ)
         (guard (not (pair? c)) (equal? c ĉ))
         (ϵ σ empty-δ))
        (((tie a t) . (tie â t̂))
         (guard (eq? a â))
         (((t . t̂) . ϵ) σ empty-δ))
        (((tie a t) . (tie â t̂))
         (guard (not (eq? a â)))
         (let ((û (apply-π ((a â)) t̂)))
           (((t . û) . ϵ) σ ((a . t̂)))))
        (((nom _) . (nom _))
         (guard (eq? (car eqn) (cdr eqn)))
         (ϵ σ empty-δ))
        (((susp π x) . (susp π̂ x̂))
         (guard (eq? (x) (x̂)))
         (let ((δ (map (λ (a) (cons a (x)))
                       (disagreement-set π π̂))))
           (ϵ σ δ)))
        (((susp π x) . t)
         (guard (not (occurs√ (x) t)))
         (let ((σ (ext-s (x) (apply-π (reverse π) t) σ)))
           (ϵ σ empty-δ)))
        ((t . (susp π x))
         (guard (not (occurs√ (x) t)))
         (let ((σ (ext-s (x) (apply-π (reverse π) t) σ)))
           (ϵ σ empty-δ)))
        (((t1 . t2) . (t̂1 . t̂2))
         (guard (untagged? t1) (untagged? t̂1))
         (((t1 . t̂1) (t2 . t̂2) . ϵ) σ empty-δ))
        (else #f)))))
```

∇-rules also takes σ as an additional argument, which it uses to walk d.

```
(define ∇-rules
  (λ (d σ δ)
    (let ((d (cons (walk (car d) σ) (walk (cdr d) σ))))
      (pmatch d
        ((a . c)
         (guard (not (pair? c)))
         (δ empty-∇))
        ((a . (tie â t))
         (guard (eq? â a))
         (δ empty-∇))
        ((a . (tie â t))
         (guard (not (eq? â a)))
         (((a . t) . δ) empty-∇))
        ((a . (nom _))
         (guard (not (eq? a (cdr d))))
         (δ empty-∇))
        ((a . (susp π x))
         (let ((a (apply-π (reverse π) a)) (x (x)))
           (δ ((a . x)))))
        ((a . (t1 . t2))
         (guard (untagged? t1))
         (((a . t1) (a . t2) . δ) empty-∇))
        (else #f)))))
```

The redefinition of *reify* uses the new *apply-reify-s* function in place of some uses of *walk**.

```
(define reify
  (λ (x p)
    (mv-let ((σ ∇) p)
      (let* ((v (walk* x σ)) (s (reify-s v)) (v (apply-reify-s v s)))
        (let ((∇ (filter (λ (a) (and (symbol? (car a)) (symbol? (cdr a))))
                         (apply-reify-s ∇ s))))
          (cond
            ((null? ∇) v)
            (else (v : ∇))))))))
```

apply-reify-s is new, but is almost identical to the old definition of *walk** in section 11.3.

```
(define apply-reify-s
  (λ (v s)
    (pmatch v
      (c (guard (not (pair? c))) c)
      ((tie a t) (list tie (get a s) (apply-reify-s t s)))
      ((nom _) (get v s))
      ((susp () _) (get v s))
      ((susp π x)
       (list susp
         (map (λ (swap)
                (pmatch swap
                  ((a b) (list (get a s) (get b s)))))
              π)
         (get (x) s)))
      ((a . d) (cons (apply-reify-s a s) (apply-reify-s d s))))))
```

By using triangular rather than idempotent substitutions, unification is as much as ten times faster and is more memory efficient.

An important limitation of both the triangular and idempotent implementations is that neither currently supports disequality constraints.

Part IV

Tabling

Chapter 12

Techniques III: Tabling

This chapter introduces *tabling*, an extension of memoization to logic programming. We present a full implementation of tabling for miniKanren in Chapter 13.

This chapter is organized as follows. In section 12.1 we review memoization as used in functional programming. Section 12.2 introduces tabling, explains how tabling differs from memoization, and describes a few of the many applications of tabling. In section 12.3 we present the **tabled** form, used to create tabled relations. In section 12.4 we examine several examples of tabled relations, and in section 12.5 we discuss the limitations of tabling.

12.1 Memoization

Consider the naive Scheme implementation of the Fibonacci function.

```
(define fib
  (λ (n)
    (cond
      ((= 0 n) 0)
      ((= 1 n) 1)
      (else (+ (fib (− n 1)) (fib (− n 2)))))))
```

The call (*fib* 5) results in calls to (*fib* 4) and (*fib* 3); the resulting call to (*fib* 4) also calls (*fib* 3). The call (*fib* 5) therefore results in two calls to (*fib* 3), the second of which performs duplicate work. Similarly, (*fib* 5) results in three calls to (*fib* 2), five calls to (*fib* 1), and three calls to (*fib* 0). Due to these redundant calls, the time complexity of *fib* is exponential in n.

To avoid this duplicate work, we could record each distinct call to *fib* in a table, along with the answer returned by that call. Whenever a duplicate call to *fib* is made, *fib* would return the answer stored in the table instead of recomputing the result. This optimization technique, known as *memoization* (Michie 1968), can result in a lower complexity class for the running time of the memoized function. Indeed, the memoized version of *fib* runs in linear rather than exponential time.

Memoization is a common technique in functional programming, since it often improves performance of recursive functions. In this chapter we consider the related technique of *tabling*, which generalizes memoization to logic programming.

12.2 Tabling

Tabling is a generalization of memoization; tabling allows a relation to store and reuse its previously computed results. Tabling a relation is more complicated than memoizing a function, since a relation returns a potentially infinite stream of substitutions rather than a single value. Also, the arguments to a tabled relation can contain unassociated logic variables or partially instantiated terms, which complicates determining whether a call is a variant of a previously seen call.

Tabling, like memoization, can result in dramatic performance gains for some programs. For example, combining tabling with Prolog's Definite Clause Grammars (Pereira and Warren 1986) makes it trivial to write efficient recursive descent parsers that handle left-recursion[1] (Becket and Somogyi 2008)—these parsers are equivalent to "packrat" parsing (Ford 2002). Tabling is also useful for writing programs that must calculate fixed points, such as abstract interpreters and model checkers (Warren 1992; Guo and Gupta 2009). However, the real reason we are interested in tabling is that many relations that would otherwise diverge terminate under tabling, as we will see in section 12.4.

An excellent introduction to tabling and its uses is Warren's survey (Warren 1992).

12.3 The tabled Form

Tabled relations are constructed using the **tabled** form:

(**tabled** $(x \ldots)$ g $g^* \ldots$)

For example,

(**define** f^o (**tabled** (z) $(\equiv z$ 5))))

defines a top-level tabled goal constructor named f^o. Each tabled goal constructor has its own local table, which can be garbage collected once there are no live references to the goal constructor. Keep in mind that the table is associated with the goal constructor, not the goal returned by the goal constructor.

Calls to a tabled relation come in two flavors: master calls and slave calls. A *master call* is a call to a tabled relation whose arguments are not (yet) stored in

[1] One important use of tabling by Prolog systems is to handle left-recursive definitions of goals; due to Prolog's incomplete depth-first search, calls to left-recursive goals often diverge. Since miniKanren uses a complete search strategy, handling left-recursion is not a problem. However, we will see in section 12.4 that there are other programs we want to write that terminate under tabling but diverge otherwise.

the table. A *slave call* is a call whose arguments are found in the table; each slave call is a *variant* of some master call.

Two calls to the same tabled relation are variants of each other if their arguments are the same, up to consistent renaming of unassociated logic variables[2]. For example, consider the calls $(\mathit{mul}^o\ y\ z\ 5)$ and $(\mathit{mul}^o\ w\ w\ x)$ in the substitutions ((y . z)) and ((x . 5)), respectively. Taking the substitutions into account, these calls are equivalent to $(\mathit{mul}^o\ z\ z\ 5)$ and $(\mathit{mul}^o\ w\ w\ 5)$, which are variants of each other. However, the calls $(\mathit{mul}^o\ w\ w\ x)$ and $(\mathit{mul}^o\ y\ z\ z)$ are variants only if w is associated with x, and y is associated with z, respectively. For the same reason, $(\mathit{mul}^o\ w\ 5\ 6)$ and $(\mathit{mul}^o\ y\ z\ 6)$ are variants only if z is associated with 5 in the substitution in place for the second call.

12.4 Tabling Examples

We are now ready to examine examples of tabled relations. The canonical example relation, path^o[3], finds all paths between two nodes in a directed graph. The goal $(\mathit{path}^o\ x\ y)$ succeeds if there is a directed edge from x to y, or if there is an edge from x to some node z and there is a path from z to y.

```
(define path^o
  (λ (x y)
    (cond^e
      ((arc^o x y))
      ((exist (z)
         (arc^o x z)
         (path^o z y))))))
```

The goal $(\mathit{arc}^o\ x\ y)$ succeeds if there is a directed edge from node x to node y.

```
(define arc^o
  (λ (x y)
    (cond^e
      ((≡ a x) (≡ b y))
      ((≡ c x) (≡ b y))
      ((≡ b x) (≡ d y)))))
```

This definition of arc^o represents edges from a to b, c to b, and b to d.
The expression $(\mathbf{run}^*\ (q)\ (\mathit{path}^o\ \mathsf{a}\ q))$ returns (b d), indicating that only the nodes b and d are reachable from a.

Now let us redefine arc^o to represent a different set of directed edges, this time with a circularity between nodes a and b.

[2] In other words, the two lists of arguments to the relation, when reified with respect to their "current" substitutions, must be *equal?*.

[3] The path examples in this section are taken from Warren (1992).

```
(define arcᵒ
  (λ (x y)
    (condᵉ
      ((≡ a x) (≡ b y))
      ((≡ b x) (≡ a y))
      ((≡ b x) (≡ d y)))))
```

Using the new definition of arc^o, the expression (**run*** (q) ($path^o$ a q)) now diverges. We can understand the cause of this divergence if we replace **run*** with **run¹⁰**.

(**run¹⁰** (q) ($path^o$ a q)) ⇒ (b a d b a d b a d b)

Because of the circular path between a and b, ($path^o$ a q) keeps finding longer and longer paths between a and the nodes b, a, and d. To avoid this problem, we can table $path^o$.

```
(define pathᵒ
  (tabled (x y)
    (condᵉ
      ((arcᵒ x y))
      ((exist (z)
         (arcᵒ x z)
         (pathᵒ z y))))))
```

(**run*** (q) ($path^o$ a q)) then converges, returning (b a d).

Now let us consider a mutually recursive program.

```
(letrec ((fᵒ (λ (x)
               (condᵉ
                 ((≡ 0 x))
                 ((gᵒ x)))))
         (gᵒ (λ (x)
               (condᵉ
                 ((≡ 1 x))
                 ((fᵒ x))))))
  (run* (q) (fᵒ q)))
```

This expression diverges. If we replace **run*** with **run¹⁰** the program converges with the value (0 1 0 1 0 1 0 1 0 1). If we table either f^o, g^o, or both, (**run*** (q) (f^o q)) converges with the value (0 1).

12.5 Limitations of Tabling

Tabling is a remarkably useful addition to miniKanren, and can be used to improve efficiency of relations and (sometimes) avoid divergence. Unfortunately, tabling is not a panacea. In fact, tabling can be trivially defeated by changing one or more arguments in each call to a tabled relation. For example, consider the ternary multiplication relation mul^o from Chapter 6. The arguments in the call

(*mul^o* (1 1 . x) x (0 0 0 1 . x))[4]

all share the variable x. The resulting goal succeeds only if there exists a non-negative integer x that satisfies $(3+4x)\cdot x = 8+16x$. *mul^o* enumerates all non-negative integer values for x until it finds one that satisfies this equation. However, if no such x exists the call to *mul^o* will diverge. Tabling will not help, since the value of x keeps changing.

Another disadvantage of tabling is that it can greatly increase the memory consumption of a program. This is a problem with memoization in general. For example, consider the tail-recursive accumulator-passing-style Scheme definition of factorial[5].

```
(define !-aps
  (λ (n a)
    (cond
      ((zero? n) a)
      (else (!-aps (sub1 n) (∗ n a))))))
```

Other than the space used to represent numbers, this function uses a bounded amount of memory[6]. However, the memoized version of *!-aps* uses an unbounded amount of memory if n is negative, and otherwise uses an amount of memory linear in n.

Chapter 13 presents a complete implementation of tabling for miniKanren; this implementation has several limitations. The first limitation is that tabled relations must be closed; a tabled goal constructor cannot contain free logic variables, since associations for those variables would be thrown away. This is a consequence of not storing entire substitutions in a relation's table, as described in section 13.1.

Another limitation is that arguments passed to tabled relations must be "printable" (or "reifiable") values. For example, tabled relations should never be passed functions, including goals, since all functions reify to the same value[7].

The most significant limitation of our tabling implementation is that it does not currently support disequality constraints, nominal unification, or freshness constraints. How to best combine tabling and constraints is an open research problem (Schrijvers et al. 2008a).

[4] This example is due to Oleg Kiselyov (personal communication).

[5] The call (*!-aps* n 1) calculates the factorial of n.

[6] Scheme implementations are required to handle tail calls properly—thus *!-aps* uses a constant amount of stack space

[7] Pure relations should never take functions as arguments anyway, since miniKanren does not support higher-order unification, and cannot meaningfully construct functions when running backwards.

Chapter 13

Implementation V: Tabling

In this chapter we implement the tabling scheme described in Chapter 12. Our tabling implementation extends the streams-based implementation of miniKanren from Chapter 3, preserving the original implementation's interleaving search behavior.

This chapter is organized as follows. In section 13.1 we describe the core data structures used in the implementation. Section 13.2 gives a high-level description of the tabling algorithm. In section 13.3 we introduce a new type of *waiting* stream, which requires extending both **case**$^{\infty}$ and the operators that use it: *take*, *bind*, and *mplus*. Section 13.4 extends the reifier from Chapter 3 with a new function *reify-var*. Finally in section 13.5 we present the heart of the tabling implementation: the user-level **tabled** form, and the *master* and *reuse* functions to handle master and slave calls, respectively.

13.1 Answer Terms, Caches, and Suspended Streams

Like any goal, a goal returned by a tabled goal constructor is a function mapping a substitution to a stream of substitutions. The goal constructor's table does not store entire substitutions; rather, the table stores *answer terms*. An answer term is a list of the arguments from a master call, perhaps partially or fully instantiated as a result of running the goal's body. A *cache* associates each master call with a set of answer terms. A subsequent slave call reuses the master call's tabled answers by unifying each answer term in the cache with the slave call's actual parameters, producing a stream of answer substitutions.

There may be multiple slave calls associated with each master call; each slave call "consumes" *all* the tabled answer terms in the cache. Evaluation of the master call and its slave calls are interleaved—slave calls may start consuming answer terms before the master call has finished producing them. When a master call produces new answer terms, the consumption of these answers by associated slave calls can result in new master or slave calls. The algorithm reaches a fixed point when all master calls have finished producing answers, and each slave call has consumed every answer term produced by its associated master call.

To understand why we table answer terms rather than full substitutions, consider this **run*** expression.

```
(let ((f (tabled (z) (≡ z 6))))
  (run* (q)
    (exist (x y)
      (cond^e
        ((≡ x 5) (f y))
        ((f y)))
      (≡ (x y) q))))
```

Imagine that the first **cond**^{e} clause is evaluated completely before the second clause. When the master call (f y) in the first clause succeeds, the substitution will be ((y . 6) (x . 5)). If we were to table the full substitution, including the association for x, the slave call in the second clause would incorrectly associate x with 5. The **run*** expression would therefore return ((5 6) (5 6)) instead of the correct answer ((5 6) (_$_0$ 6)).

Since the table records answer terms rather than entire substitutions, a tabled goal constructor must be closed with respect to logic variables; values associated with free logic variables would be forgotten. For example, the **run*** expression

```
(run* (q)
  (exist (x y)
    (let ((f (λ (z) (exist () (≡ x 5) (≡ z 6)))))
      (cond^e
        ((f y) (≡ (x y) q))
        ((f y) (≡ (x y) q))))))
```

returns ((5 6) (5 6)), as expected. However, if we were to table f by replacing (λ (z) (**exist** () (≡ x 5) (≡ z 6))) with (**tabled** (z) (**exist** () (≡ x 5) (≡ z 6))), the **run*** expression would instead return ((5 6) (_$_0$ 6)).

Each tabled goal constructor has its own local table represented as a list of (*key* . *cache*) pairs, where *key* is a list of reified arguments from a master call, and where *cache* contains the set of answer terms for that master call.

A cache is represented as a tagged vector, and contains a list of tabled answer terms. Each master call is associated with a single cache.

```
(define make-cache (λ (ansv*) (vector cache ansv*)))
(define cache-ansv* (λ (c) (vector-ref c 1)))
(define cache-ansv*-set! (λ (c val) (vector-set! c 1 val)))
```

Each slave call is associated with a single *suspended stream*, or *ss*. Each suspended stream is represented as a tagged vector containing a cache, a list of tabled answer terms *ansv**, and a thunk that produces the remainder of the stream (an f, as described in section 3.3).

```
(define make-ss (λ (cache ansv* f) (vector ss cache ansv* f)))
(define ss? (λ (x) (and (vector? x) (eq? (vector-ref x 0) ss))))
(define ss-cache (λ (ss) (vector-ref ss 1)))
(define ss-ansv* (λ (ss) (vector-ref ss 2)))
(define ss-f (λ (ss) (vector-ref ss 3)))
```

The *ansv** list indicates which of the master call's answer terms the suspended stream has already processed—*ansv** is always a suffix of the list in *cache*. There may be many suspended streams associated with a single *cache*—each of these *ss*'s may contain a different *ansv** list, representing a different "already seen" suffix of answer terms from the cache.

The *ss-ready?* predicate indicates whether a suspended stream's cache contains new answer terms not yet consumed by the stream.

```
(define ss-ready? (λ (ss) (not (eq? (cache-ansv* (ss-cache ss)) (ss-ansv* ss)))))
```

13.2 The Tabling Algorithm

Now that we are familiar with the fundamental data structures, we can examine in detail the steps performed when a tabled goal constructor is called:

1. The goal constructor creates a list of the arguments passed to the call, *argv*, then returns a goal.

2. When passed a substitution *s*, the goal reifies *argv* in *s*, producing a list *key* of reified arguments.

3. The goal uses the reified list of arguments as the lookup key in the goal constructor's local table, which is an association list of (*key* . *cache*) pairs.

4. If the key is not in the table's association list we are making a new master call. The goal constructs a new cache containing the empty list. The goal then side-effects the local table, extending it with a pair containing the new key and cache. Next, a "fake" subgoal is added to the body of the goal. When passed a substitution, this "fake" goal checks if the answer term about to be cached is equivalent to an existing answer term in the cache; if so, the fake goal fails, keeping the master call from producing a duplicate answer. Otherwise, the fake goal extends the cache with the new answer term, then returns the answer substitution as a singleton stream[1].

5. If, on the other hand, the key is found in the table's association list, we are making a slave call. Instead of re-running the body of the goal, we reuse the tabled answers from the corresponding master call. The slave call produces a stream of answer substitutions by unifying, in the current substitution, *ansv**

[1] This singleton stream is actually a *waiting stream*, described in section 13.3.

with each cached answer term. Due to miniKanren's interleaving search, a master call may not produce all of its answers immediately. Therefore, the answer stream produced by a slave call may need to suspend periodically, "awakening" when the master call produces new answer terms for the slave to consume.

Recall that the algorithm reaches a fixed point when all the master calls have finished producing answers, and each slave call has consumed every answer term produced by its corresponding master call. In the process of consuming a cached answer term, a slave call might make a new master or slave call.

13.3 Waiting Streams

We extend the a^∞ stream datatype described in section 3.3 with a new variant: a *waiting* stream w is a non-empty proper list $(ss\ ss^* \ldots)$ of suspended streams. The waiting stream datatype allows us to express a disjunction of suspended streams; just as importantly, the datatype makes it easier to recognize when a fixed point has been reached, as described below.

```
(define w? (λ (x) (and (pair? x) (ss? (car x)))))
```

New singleton waiting streams are created in the *reuse* function described in section 13.5. The only way to create a waiting stream containing multiple suspended streams is through disjunction (see the definition of *mplus* below).

The addition of the waiting stream type requires us to extend the definition of **case**$^\infty$ from section 3.3 with a new w clause.

```
(define-syntax case∞
  (syntax-rules ()
    ((_ e (() e0) ((f̂) e1) ((w) ew) ((â) e2) ((a f) e3))
     (let ((a∞ e))
       (cond
         ((not a∞) e0)
         ((procedure? a∞) (let ((f̂ a∞)) e1))
         ((and (pair? a∞) (procedure? (cdr a∞)))
          (let ((a (car a∞)) (f (cdr a∞))) e3))
         ((w? a∞) (w-check a∞
                           (λ (f̂) e1)
                           (λ () (let ((w a∞)) ew))))
         (else (let ((â a∞)) e2)))))))
```

The new clause of **case**$^\infty$ expands into a call to *w-check*, which takes a waiting stream w, a success continuation *sk*, and a failure continuation *fk*. *w-check* plays a critical role in finding the fixed point of a program.

w-check looks in *w* for the first suspended stream *ss* whose cache contains new answer terms. If none of the suspended streams contain unseen answer terms, *w-check* invokes the failure continuation. Otherwise, *sk* is passed an *f*-type stream containing the new answers produced by *ss*, interleaved with answers from a new waiting stream containing the remaining suspended streams in *w*.

```
(define w-check
  (λ (w sk fk)
    (let loop ((w w) (a ()))
      (cond
        ((null? w) (fk))
        ((ss-ready? (car w))
         (sk (λF ()
               (let ((f (ss-f (car w)))
                     (w (append (reverse a) (cdr w))))
                 (if (null? w) (f) (mplus (f) (λF () w)))))))
        (else (loop (cdr w) (cons (car w) a)))))))
```

The *w* case of **case**$^\infty$ actually represents *two* cases: in the first case, the waiting stream can produce new answers; in the second case, the stream cannot produce new answers, although it may be able to in the future.

In the first case, *w* contains a suspended stream *ss* ready to produce new answers. *w-check* creates a new *f*-type stream encapsulating the answers from *ss*, along with the remainder of the *w* stream (if non-empty). **case**$^\infty$ then processes this stream as it would any other *f*: by evaluating the expression e_1 in an extended environment in which $\hat{f}$ is bound to the new stream. To see this more clearly, think of the *sk* passed to *w-check*, (λ ($\hat{f}$) e_1), as the equivalent (λ (a^∞) (**let** (($\hat{f}$ a^∞)) e_1)), which exactly mirrors the code produced in the $\hat{f}$ case of **case**$^\infty$.

In the second case, none of the suspended streams in *w* can produce new answers. We have therefore reached a fixed point, at least temporarily; this case is analogous to the () case of **case**$^\infty$. Unlike in the () case, however, *w* might produce answers later. *ew* is evaluated in an extended environment in which *w* is bound to the waiting stream—this is made most clear in the *w* case of *mplus*, below.

Since we have added a clause to **case**$^\infty$ we must redefine *take*, *bind*, and *mplus*. These functions differ from their definitions in section 3.3 only in the addition of the *w* case. However, the *w* case implicitly uses the expression specified for the *f* case as well[2], if *w* contains a suspended stream ready to produce new answers. In this event, a new *f*-type stream is constructed that contains not only these new answers but also the remaining suspended streams in *w*; this new stream is then handled by the *f* case of **case**$^\infty$.

[2] Or the $\hat{f}$ case, in the definition of *mplus*.

Here is the updated definition of *take*.

```
(define take
  (λ (n f)
    (if (and n (zero? n))
      ()
      (case∞ (f)
        (() ())
        ((f) (take n f))
        ((w) ())
        ((a) a)
        ((a f) (cons (car a) (take (and n (− n 1)) f)))))))
```

If *w* contains a suspended stream ready to produce a new stream of answers, this new stream is handled by the *f* case of **case**$^\infty$. Otherwise, we have reached a fixed point—therefore, *take* returns the empty list.

Here is the updated definition of *bind*.

```
(define bind
  (λ (a∞ g)
    (case∞ a∞
      (() (mzero))
      ((f) (inc (bind (f) g)))
      ((w) (map (λ (ss)
                  (make-ss (ss-cache ss) (ss-ansv* ss)
                           (λ_F () (bind ((ss-f ss)) g))))
                w))
      ((a) (g a))
      ((a f) (mplus (g a) (λ_F () (bind (f) g)))))))
```

If *w* contains a suspended stream ready to produce a new stream of answers, this new stream is handled by the *f* case of **case**$^\infty$. Otherwise, the binding of answer substitutions to *g* must be delayed, because the streams in *w* are all suspended. *bind* reconstructs the list *w*, pushing the bind operation into each rebuilt suspended stream. If a stream is awakened later, it will then bind its new answers to *g*.

Here is the updated definition of *mplus*.

```
(define mplus
  (λ (a∞ f)
    (case∞ a∞
      (() (f))
      ((f̂) (inc (mplus (f) f̂)))
      ((w) (λ_F () (let ((a∞ (f)))
                     (if (w? a∞)
                       (append a∞ w)
                       (mplus a∞ (λ_F () w))))))
      ((a) (choice a f))
      ((a f̂) (choice a (λ_F () (mplus (f) f̂)))))))
```

If w contains a suspended stream ready to produce a new stream of answers, this new stream is handled by the f case of **case**$^\infty$. Otherwise, *mplus* returns a new f-type stream. If the second argument to *mplus* produces a waiting stream $\hat{w}$, then *mplus* appends the lists $\hat{w}$ and w, creating a single combined waiting stream. If *mplus*'s second argument produces an a^∞ that is *not* a waiting stream, then w is "pushed" to the back of the new stream. Accumulating all suspended streams in a single waiting stream at the end of an f-type stream allows *w-check* to easily determine if a fixed point has been reached.

13.4 Extending and Abstracting Reification

To avoid prematurely instantiating logic variables, the *master* and *reuse* procedures in section 13.5 copy the list *argv* of arguments passed to the tabled goal constructor. This operation is performed by the *reify-var* function, which is similar in spirit to Prolog's `copy_term/2`, but is implemented by a function rather than a user-level goal constructor.

Our implementation of *reify-var* is identical to that of the *reify* function from Chapter 3, except that unassociated variables are consistently replaced with newly created logic variables rather than with symbols. We therefore abstract the reification operators, defining them in terms of the *make-reify* helper, which in turn uses an abstracted version of *reify-s*.

```
(define make-reify
  (λ (rep)
    (λ (v s)
      (let ((v (walk* v s)))
        (walk* v (reify-s rep v empty-s))))))

(define reify (make-reify reify-name))

(define reify-var (make-reify reify-v))

(define reify-v
  (λ (n)
    (var n)))

(define reify-s
  (λ (rep v s)
    (let ((v (walk v s)))
      (cond
        ((var? v) (ext-s-no-check v (rep (length s)) s))
        ((pair? v) (reify-s rep (cdr v) (reify-s rep (car v) s)))
        (else s)))))
```

13.5 Core Tabling Operators

We are now ready to define the core tabling operators. The **tabled** user-level form creates a tabled goal constructor, complete with an empty local association list *table* that will contain **(***key* **.** *cache***)** pairs. Section 13.2 describes the behavior of tabled goal constructors at a high level; most of the interesting work is performed in the *master* and *reuse* helpers, defined below.

```
(define-syntax tabled
  (syntax-rules ()
    ((_ (x ...) g g* ...)
     (let ((table ()))
       (λ (x ...)
         (let ((argv (list x ...)))
           (λG (s)
             (let ((key (reify argv s)))
               (cond
                 ((assoc key table)
                  ⇒ (λ (key.cache) (reuse argv (cdr key.cache) s)))
                 (else (let ((cache (make-cache ())))
                         (set! table (cons (key . cache) table))
                         ((exist () g g* ... (master argv cache)) s))))))))))))
```

The *master* function is invoked during a master call, and returns a "fake" goal run at the end of the body of the tabled goal. This fake goal checks if the answer term about to be cached is equivalent to an answer term already in the cache. If so, the call to the fake goal fails, to avoid producing a duplicate answer. Otherwise, the goal succeeds, caching the new answer term before returning the answer substitution.

```
(define master
  (λ (argv cache)
    (λG (s)
      (and
        (for-all
          (λ (ansv) (not (alpha-equiv? argv ansv s)))
          (cache-ansv* cache))
        (begin
          (cache-ansv*-set! cache (cons (reify-var argv s) (cache-ansv* cache)))
          s)))))
```

alpha-equiv? returns true if *x* and *y* represent the same term, modulo consistent replacement of unassociated logic variables.

```
(define alpha-equiv?
  (λ (x y s)
    (equal? (reify x s) (reify y s))))
```

reuse constructs a stream of answer substitutions for a slave call, using the cached answer terms from the corresponding master call. Like *w-check*, *reuse* plays a critical role in calculating the fixed point of a program. Each call to *loop* returns an **(***a* **.** *f***)**-type stream until all the answer terms in the cache have been consumed. *reuse* then returns a waiting stream[3] encapsulating a single suspended stream whose *f* calls the outer *fix* loop, consuming any answer terms produced by the master call while the stream was suspended. Invoking *f* restarts the search for a fixed point; to avoid divergence, *w-check* does not invoke the *f* of any suspended stream that does not contain unseen answer terms.

```
(define reuse
  (λ (argv cache s)
    (let fix ((start (cache-ansv* cache)) (end ()))
      (let loop ((ansv* start))
        (if (eq? ansv* end)
            (list (make-ss cache start (λF () (fix (cache-ansv* cache) start))))
            (choice (subunify argv (reify-var (car ansv*) s) s)
                    (λF () (loop (cdr ansv*)))))))))
```

reuse depends on *subunify* to unify the list of unreified arguments in the slave call with a copy of each cached answer term. Since we know that the unification will succeed, the definition of *subunify* is shorter and more efficient than the definition of *unify* from Chapter 3.

```
(define subunify
  (λ (arg ans s)
    (let ((arg (walk arg s)))
      (cond
        ((eq? arg ans) s)
        ((var? arg) (ext-s-no-check arg ans s))
        ((pair? arg) (subunify (cdr arg) (cdr ans)
                       (subunify (car arg) (car ans) s)))
        (else s)))))
```

The code in this chapter is short but extremely subtle. This subtlety is due to the use of side effects, interaction of multiple functions to calculate fixed points, and introduction of the suspended stream and waiting stream datatypes. Understanding how the manipulation of waiting streams by *mplus* makes the definition of *w-check* possible is especially subtle. To fully appreciate this last point, the reader is encouraged to modify the implementation by replacing all uses of waiting streams with suspended streams, and then ascertain why many tabled programs diverge as a result.

[3] This is the only code that introduces a new waiting stream, as opposed to rebuilding or appending existing waiting streams.

Part V

Ferns

Chapter 14

Techniques IV: Ferns

In this chapter we provide a bottom-avoiding generalization of core miniKanren using *ferns* (Friedman and Wise 1981), a shareable data structure designed to avoid divergence.

The chapter is organized as follows. Section 14.1 introduces the ferns data structure and shows examples of familiar recursive functions using ferns. Section 14.2 describes the promotion algorithm (Friedman and Wise 1979) that characterizes the necessary sharing properties of ferns. Section 14.3 defines bottom-avoiding logic programming goal constructors, corresponding to core miniKanren with non-interleaving search. Chapter 15 presents a complete *shallow embedding* (Boulton et al. 1992) of the ferns data structure and related operators.

14.1 Introduction to Ferns

Ferns are constructed with *cons* and $\mathbf{cons}_\bot$, originally called **frons** (Friedman and Wise 1980), and accessed by $\mathit{car}_\bot$ and $\mathit{cdr}_\bot$, generalizations of *car* and *cdr*, respectively. Ferns built with $\mathbf{cons}_\bot$ are like streams in that the *evaluation* of elements is delayed, permitting unbounded data structures. In contrast to streams, the *ordering* of elements is also delayed: convergent values form the prefix in some unspecified order, while divergent values form the suffix.

We begin with several examples that illustrate the properties of ferns, showing their similarities to and differences from traditional lists and streams. Later, we include examples that show that a natural recursive style can be used when programming with ferns and point out the advantages ferns afford the user.

14.1.1 Two Simple Programs

Convergent elements of a fern form its prefix in some unspecified order. For example, evaluating the expression

```
(let ((s (cons⊥ 0 (cons⊥ 1 ()))))
  (display (car⊥ s)) (display (cadr⊥ s)) (display (car⊥ s)))
```

prints either 010 or 101, demonstrating that the order of values within a fern is not specified in advance but remains consistent once determined, while

```
(let ((s1 (cons⊥ (! 6) ⊥)) (s2 (cons⊥ ⊥ (cons⊥ (! 5) ⊥))))
  (cons (car⊥ s1) (car⊥ s2)))
```

returns (720 . 120), demonstrating that accessing a fern avoids divergence as much as possible. (⊥ is any expression whose evaluation diverges.) In the latter example, each fern contains only one convergent value; taking the $cdr_\perp$ of s_1 or the $cadr_\perp$ of s_2 results in divergence.

Ferns are *shareable* data structures; sharing, combined with delayed ordering of values, can result in surprising behavior. For example, consider these expressions:

```
(let ((b (cons 2 ())))
  (let ((a (cons 1 b)))
    (list (car a) (cadr a) (car b))))
```

and

```
(let ((b (cons⊥ 2 ())))
  (let ((a (cons⊥ 1 b)))
    (list (car⊥ a) (cadr⊥ a) (car⊥ b))))
```

The first expression must evaluate to (1 2 2). The second expression may also return this value—as expected, the car of *b* would then be equal to the cadr of *a*. The second expression might instead return (2 1 2) however; in this case, the car of *b* would be equal to the car of *a* rather than to its cadr. Section 14.2 discusses sharing in detail.

14.1.2 Recursion

We now present examples of the use of ferns in simple recursive functions. Consider the definition of $ints\text{-}from_\perp$[1].

```
(define ints-from⊥
  (λt (n)
    (cons⊥ n (ints-from⊥ (+ n 1)))))
```

Then ($caddr_\perp$ ($ints\text{-}from_\perp$ 0)) could return any non-negative integer, whereas a stream version would return 2.

There is a tight relationship between ferns and lists, since every cons pair is a fern. The empty fern is also represented by **()**, and (*pair?* (**cons**$_\perp$ e_1 e_2)) returns #t for all e_1 and e_2. After replacing the list constructor *cons* with the fern constructor

[1] λ_t is identical to λ, except it creates preemptible procedures. (See Appendix D.)

$\mathbf{cons}_\perp$, many recursive functions operating on lists avoid divergence. For example, $\mathit{map}_\perp$ is defined by replacing *cons* with $\mathbf{cons}_\perp$, *car* with $\mathit{car}_\perp$, and *cdr* with $\mathit{cdr}_\perp$ in the definition of *map*, and can map a function over an unbounded fern: the value of ($\mathit{caddr}_\perp$ ($\mathit{map}_\perp$ *add1* ($\mathit{ints\text{-}from}_\perp$ 0))) can be any positive integer.

Ferns work especially well with *annihilators*. True values are annihilators for $\mathit{or}_\perp$

```
(define or⊥
  (λt (s)
    (cond
      ((null? s) #f)
      ((car⊥ s) (car⊥ s))
      (else (or⊥ (cdr⊥ s))))))
```

which searches in a fern for a true convergent value and avoids divergence if it finds one: ($\mathit{or}_\perp$ ($\mathbf{list}_\perp$ $\perp$ (*odd?* 1) (! 5) $\perp$ (*odd?* 0))) returns some true value, where $\mathbf{list}_\perp$ is defined as follows.

```
(define-syntax list⊥
  (syntax-rules ()
    ((_) ())
    ((_ e e* ...) (cons⊥ e (list⊥ e* ...)))))
```

Let us define $\mathit{append}_\perp$ for ferns.

```
(define append⊥
  (λt (s1 s2)
    (cond
      ((null? s1) s2)
      (else (cons⊥ (car⊥ s1) (append⊥ (cdr⊥ s1) s2))))))
```

To observe the behavior of $\mathit{append}_\perp$, we define $\mathit{take}_\perp$ whose first argument is either #f (all results) or $n > 0$ (no more than n results).

```
(define take⊥
  (λt (n s)
    (cond
      ((null? s) ())
      (else (cons (car⊥ s)
                  (if (and n (= n 1)) () (take⊥ (and n (− n 1)) (cdr⊥ s)))))))))
```

When determining the nth value, it is necessary to avoid taking the $\mathit{cdr}_\perp$ after the nth value is determined, since it is that $\mathit{cdr}_\perp$ that might not terminate and we already have n results.

The definition of $\mathit{append}_\perp$ appears to work as expected:

($\mathit{take}_\perp$ 2 ($\mathit{append}_\perp$ ($\mathbf{list}_\perp$ 1) ($\mathbf{list}_\perp$ $\perp$ 2))) ⇒ (1 2).

Moving $\perp$ from the second argument to the first, however, reveals a problem:

$(\mathit{take}_\perp\ \mathsf{2}\ (\mathit{append}_\perp\ (\mathbf{list}_\perp\ \perp\ \mathsf{1})\ (\mathbf{list}_\perp\ \mathsf{2}))) \Rightarrow \perp.$

Even though the result of the call to *append*$_\perp$ should contain two convergent elements, taking the first two elements of that result diverges. This is because the definition of *append*$_\perp$ requires that s_1 be completely exhausted before any elements from s_2 can appear in the result. If one of the elements of s_1 is $\perp$, then no element from s_2 will ever appear. The same is true if s_1 contains an unbounded number of convergent elements: since s_1 is never null, the result will never contain elements from s_2. With the definition of **mplus**$_\perp$ in Section 14.3.1, it becomes clear that the solution to these problems is to interleave the elements from s_1 and s_2 in the resulting fern as in the next example.

Functional programs often share rather than copy data, and ferns are designed to encourage this programming style. Consider a procedure to compute the Cartesian product of two ferns:

```
(define Cartesian-product⊥
  (λt (s1 s2)
    (cond
      ((null? s1) ())
      (else (mplus⊥ (map⊥ (λt (e) (cons (car⊥ s1) e)) s2)
                    (Cartesian-product⊥ (cdr⊥ s1) s2))))))
```

$(\mathit{take}_\perp\ \mathsf{6}\ (\mathit{Cartesian\text{-}product}_\perp\ (\mathbf{list}_\perp\ \perp\ \mathsf{a}\ \mathsf{b})\ (\mathbf{list}_\perp\ \mathsf{x}\ \perp\ \mathsf{y}\ \perp\ \mathsf{z})))$

$\rightsquigarrow$ **((a . x) (a . y) (b . x) (a . z) (b . y) (b . z))**

where $\rightsquigarrow$ indicates *one* of the possible values. This definition ensures that the resulting fern shares elements with the ferns passed as arguments. Many references to a particular element may be made without repeating computations, hence the expression

$(\mathit{take}_\perp\ \mathsf{2}\ (\mathit{Cartesian\text{-}product}_\perp\ (\mathbf{list}_\perp\ (\mathbf{begin}\ (\mathit{display}\ \mathsf{\#t})\ \mathsf{5}))\ (\mathbf{list}_\perp\ \mathsf{a}\ \perp\ \mathsf{b})))$

$\rightsquigarrow$ **((5 . a) (5 . b))**

prints #t *exactly once*. (There are more examples of the use of ferns in Johnson (1983), Filman and Friedman (1984), and Jeschke (1995).)

In the next section we look at how the sharing properties of ferns are maintained alongside bottom-avoidance.

14.2 Sharing and Promotion

In this section, we provide examples and a high-level description of the *promotion algorithm* of Friedman and Wise (Friedman and Wise 1979). The values in a fern are computed and *promoted* across the fern while ensuring that the correct values are available from each subfern, $\perp$'s are avoided, and non-$\perp$ values are computed only once. Ferns have structure, and there may be references to more than one subfern of a particular fern. Consider the example expression

```
(let (($\delta$ (cons$_\perp$ (! 6) ())))
  (let (($\gamma$ (cons$_\perp$ (! 3) $\delta$)))
    (let (($\beta$ (cons$_\perp$ (! 5) $\gamma$)))
      (let (($\alpha$ (cons$_\perp$ $\perp$ $\beta$)))
        (list (take$_\perp$ 3 $\alpha$) (take$_\perp$ 3 $\beta$) (take$_\perp$ 2 $\gamma$) (take$_\perp$ 1 $\delta$))))))
```

$\leadsto$ ((6 120 720) (6 120 720) (6 720) (720))

assuming *list* evaluates its arguments left-to-right. Importantly, accessing δ cannot retrieve values in the prefix of the enclosing fern α. We now describe in detail how the result of $(\mathit{take}_\perp\ 3\ \alpha)$ is determined along with the necessary changes to the fern data structure during this process. Whenever we encounter a choice, we shall assume a choice consistent with the value returned in the example.

During the first access of α the cdrs are evaluated, as indicated by the arrows in Figure 14.1a. Figure 14.1b depicts the data structure after $(\mathit{car}_\perp\ \alpha)$ is evaluated. We assume that, of the possible values for $(\mathit{car}_\perp\ \alpha)$, namely $\perp$ (which is never chosen), (! 5), (! 3), and (! 6), the value of (! 3) is chosen and promoted. Since the value of (! 3) might be a value for $(\mathit{car}_\perp\ \beta)$ and $(\mathit{car}_\perp\ \gamma)$, we replace the cars of all three pairs with the value of (! 3), which is 6. We replace the cdrs of α and β with new frons pairs containing $\perp$ and (! 5), which were not chosen. The new frons pairs are linked together, and linked at the end to the old cdr of γ. Thus α, β, and γ become a fern with 6 in the car and a fern of the rest of their original possible values in their cdrs. As a result of the promotion, α, β, and γ become cons pairs, represented in the figures by rectangles.

Figure 14.1c depicts the data structure after $(\mathit{cadr}_\perp\ \alpha)$ is evaluated. This time, (! 5) is chosen from $\perp$, (! 5), and (! 6). Since the value of (! 5) is also a possible value for $(\mathit{cadr}_\perp\ \beta)$, we replace the cadrs of both α and β with the value of (! 5), which is 120, and replace the cddr of α with a frons pair containing the $\perp$ that was not chosen and a pointer to δ. The cddr of β points to δ; no new fern with remaining possible values is needed because the value chosen for $(\mathit{cadr}_\perp\ \beta)$ was the first value available. As before, the pairs containing values become cons pairs.

Figure 14.1d depicts the data structure after $(\mathit{caddr}_\perp\ \alpha)$ is evaluated. Of $\perp$ and (! 6), it comes as no surprise that (! 6) is chosen. Since the value of (! 6), which is 720, is also a possible value for $(\mathit{car}_\perp\ \delta)$ (and in fact the only one), we update the car of δ and the car of the cddr of α with 720. The cdr of δ remains as the empty list, and the cdr of the cddr of α becomes a new frons pair containing $\perp$. The cdr of the new frons pair is the empty list copied from the cdr of δ. The remaining values are obvious given the final state of the data structure. No further manipulation of the data structure is necessary to evaluate the three remaining calls to $\mathit{take}_\perp$.

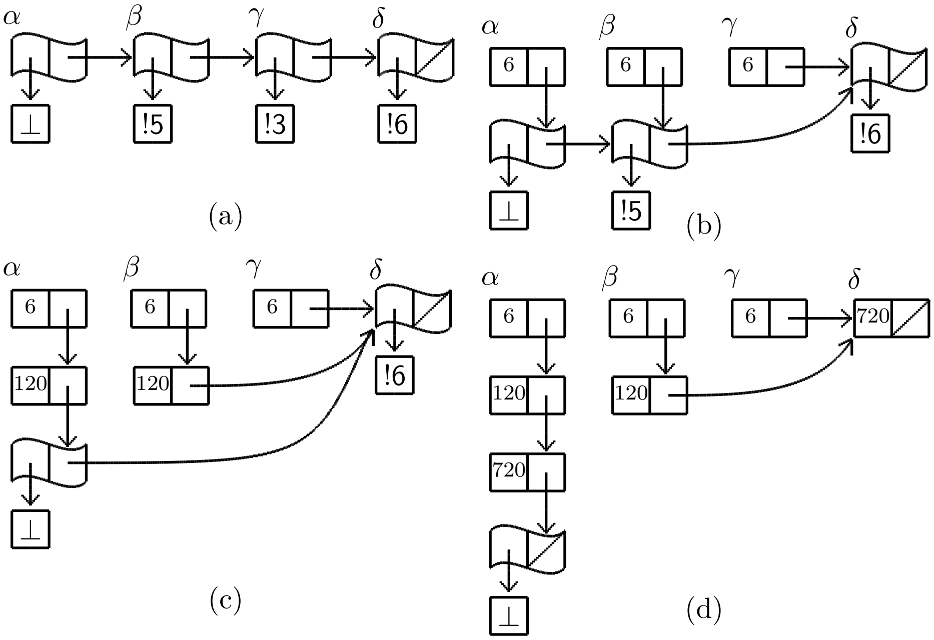

Figure 14.1: Fern α immediately after evaluation of cdrs, but before any cars have finished evaluation (a) and after the values, 6 (b), 120 (c), and 720 (d) have been promoted.

In Figure 14.1d each of the ferns α, β, γ, and δ contains some permutation of its original possible values, and $\perp$ has been pushed to the end of α. Furthermore, if there are no shared references to β, γ, and δ, the number of accessible pairs is linear in the length of the fern. If there are references to subferns, for a fern of size n, the worst case is $(n^2 + n)/2$. But, as these shared references vanish, so do the additional cons pairs.

If *list* evaluated from right-to-left instead of evaluating from left-to-right, the example expression would return `((720 6 120) (720 6 120) (720 6) (720))`. Each list would be independent of the others and the last pair of α would be a frons pair with $\perp$ in the car and the empty list in the cdr. This demonstrates that if there is sharing of these lists, the lists contain four pairs, three pairs, two pairs, and one pair, respectively. If the example expression just returned α, then only four pairs would be accessible.

The example presented in this section provides a direct view of promotion. When a fern is accessed by multiple computations, the promotion algorithm must be able to handle various issues such as multiple values becoming available for promotion at once. The code presented in Chapter 15 handles these details.

We are now ready to consider a ferns-based implementation of miniKanren.

14.3 Ferns-based miniKanren

In this section we describe a simple bottom-avoiding logic programming language, which corresponds to core miniKanren with non-interleaving search. We begin by describing and implementing operators **mplus**$_\perp$ and *bind*$_\perp$ over ferns, and go on to implement goal constructors in terms of these operators. The fern-based goal constructors are shown to be more general than the standard stream-based ones presented in Chapter 3[2].

14.3.1 mplus$_\perp$ and *bind*$_\perp$

Since we are developing goal constructors in Scheme, a call-by-value language, we make **mplus**$_\perp$ itself lazy to avoid diverging when one or more of its arguments diverge. This is accomplished by defining **mplus**$_\perp$ as a macro that wraps its two arguments in **list**$_\perp$ before passing them to *mplus-aux*$_\perp$. In addition, **mplus**$_\perp$ must interleave elements from both of its arguments so that a fern of unbounded length in the first argument will not cause the second argument to be ignored.

```
(define-syntax mplus⊥
  (syntax-rules ()
    ((_ s1 s2) (mplus-aux⊥ (list⊥ s1 s2)))))

(define mplus-aux⊥
  (λt (p)
    (cond
      ((null? (car⊥ p)) (cadr⊥ p))
      (else (cons⊥ (caar⊥ p)
                (mplus⊥ (cadr⊥ p) (cdar⊥ p)))))))

(define bind⊥
  (λt (s f)
    (cond
      ((null? s) ())
      (else (mplus⊥ (f (car⊥ s)) (bind⊥ (cdr⊥ s) f))))))
```

bind$_\perp$ avoids the same types of divergence as *map*$_\perp$ described in Section 14.1.2 but uses **mplus**$_\perp$ to merge the results of the calls to f. Thus, (*bind*$_\perp$ (*ints-from*$_\perp$ 0) *ints-from*$_\perp$) is an unbounded fern of integers; for every (nonnegative) integer n, it contains the integers starting from n and therefore every nonnegative integer n is contained $n+1$ times. The interleaving leads to duplicates in the following example:

(*take*$_\perp$ 13 (*bind*$_\perp$ (*ints-from*$_\perp$ 0) *ints-from*$_\perp$)) ⇝ (0 1 2 1 3 4 5 6 7 8 9 2 10).

The addition of *unit*$_\perp$ and *mzero*$_\perp$ rounds out the set of operators we need to implement a minimal miniKanren-like language.

[2] See Wand and Vaillancourt (2004) for a historical account of logic combinators.

```
(define unit⊥ (λ_t (s) (cons s ())))
```

```
(define mzero⊥ (λ_t () ()))
```

Using these definitions, we can run programs that require multiple unbounded ferns, such as this program inspired by Seres and Spivey (Spivey and Seres 2003) that searches for a pair a and b of divisors of 9 by enumerating the integers from 2 in a fern of possible values for a and similarly for b:

```
(car⊥ (bind⊥ (ints-from⊥ 2)
        (λ_t (a)
          (bind⊥ (ints-from⊥ 2)
            (λ_t (b)
              (if (= (* a b) 9) (unit⊥ (list a b)) (mzero⊥)))))))
⇒ (3 3).
```

Using streams instead of ferns in this example, which would be like nesting "for" loops, would result in divergence since 2 does not evenly divide 9.

14.3.2 Goal Constructors

We are now ready to define three goal constructors: $\equiv_\perp$, which unifies terms; **disj**$_\perp$, which performs disjunction over goals; and **conj**$_\perp$, which performs conjunction over goals[3]. These goal constructors are required to terminate, and they always return a goal. A *goal* is a procedure that takes a substitution and returns a fern of substitutions (rather than a stream of substitutions, as in Chapter 3).

```
(define-syntax ≡⊥
  (syntax-rules ()
    ((_ u v)
     (λ_t (s)
       (let ((s (unify u v s)))
         (if (not s) (mzero⊥) (unit⊥ s)))))))
```

```
(define-syntax disj⊥
  (syntax-rules ()
    ((_ g1 g2) (λ_t (s) (mplus⊥ (g1 s) (g2 s))))))
```

```
(define-syntax conj⊥
  (syntax-rules ()
    ((_ g1 g2) (λ_t (s) (bind⊥ (g1 s) g2)))))
```

A logic program evaluates to a goal; to obtain answers, this goal is applied to the empty substitution. The result is a fern of substitutions representing answers. We define *run*$_\perp$ in terms of *take*$_\perp$, described in Section 14.1.2, to obtain a list of answers from the fern of substitutions

[3] **disj**$_\perp$ is just a simplified version of **cond**^{e}, while **conj**$_\perp$ is just a simplified version of **exist**.

```
$(\mathbf{define}\ \mathit{run}_\perp$
  $(\lambda_t\ (n\ g)$
    $(\mathit{take}_\perp\ n\ (g\ \mathit{empty\text{-}s}))))$
```

where n is a non-negative integer (or #f) and g is a goal.

Given two logic variables x and y, here are some simple logic programs that produce the same answers using both fern-based and stream-based goal constructors.

```
$(\mathit{run}_\perp\ \texttt{\#f}\ (\equiv_\perp\ 1\ x)) \Rightarrow (\{x/1\})$
$(\mathit{run}_\perp\ 1\ (\mathbf{conj}_\perp\ (\equiv_\perp\ y\ 3)\ (\equiv_\perp\ x\ y))) \Rightarrow (\{x/3, y/3\})$
$(\mathit{run}_\perp\ 1\ (\mathbf{disj}_\perp\ (\equiv_\perp\ x\ y)\ (\equiv_\perp\ y\ 3))) \Rightarrow (\{x/y\})$
$(\mathit{run}_\perp\ 5\ (\mathbf{disj}_\perp\ (\equiv_\perp\ x\ y)\ (\equiv_\perp\ y\ 3))) \Rightarrow (\{x/y\}\ \{y/3\})$
$(\mathit{run}_\perp\ 1\ (\mathbf{conj}_\perp\ (\equiv_\perp\ x\ 5)\ (\mathbf{conj}_\perp\ (\equiv_\perp\ x\ y)\ (\equiv_\perp\ y\ 4)))) \Rightarrow ()$
$(\mathit{run}_\perp\ \texttt{\#f}\ (\mathbf{conj}_\perp\ (\equiv_\perp\ x\ 5)\ (\mathbf{disj}_\perp\ (\equiv_\perp\ x\ 5)\ (\equiv_\perp\ x\ 6)))) \Rightarrow (\{x/5\})$
```

It is not difficult, however, to find examples of logic programs that diverge when using stream-based goal constructors but converge using fern-based constructors:

```
$(\mathit{run}_\perp\ 1\ (\mathbf{disj}_\perp\ \perp\ (\equiv_\perp\ x\ 3))) \Rightarrow (\{x/3\})$
$(\mathit{run}_\perp\ 1\ (\mathbf{disj}_\perp\ (\equiv_\perp\ \perp\ x)\ (\equiv_\perp\ x\ 5))) \Rightarrow (\{x/5\})$
```

and given idempotent substitutions (Lloyd 1987), the fern-based operators can even avoid some circularity-based divergence without the occurs-check, while stream-based operators cannot:

```
$(\mathit{run}_\perp\ 1\ (\mathbf{disj}_\perp\ (\equiv_\perp\ (\mathit{list}\ x)\ x)\ (\equiv_\perp\ x\ 6))) \Rightarrow (\{x/6\})$
```

There are functions that represent relations. The relation $\mathit{always\text{-}five}_\perp$ associates 5 with its argument an unbounded number of times:

```
$(\mathbf{define}\ \mathit{always\text{-}five}_\perp$
  $(\lambda_t\ (x)$
    $(\mathbf{disj}_\perp\ (\mathit{always\text{-}five}_\perp\ x)\ (\equiv_\perp\ x\ 5))))$
```

Because both stream and fern constructors do not evaluate their arguments, we may safely evaluate the goal $(\mathit{always\text{-}five}_\perp\ x)$, obtaining an unbounded collection of answers. Using $\mathit{run}_\perp$, we can ask for a finite number of these answers. Because the ordering of streams is determined at construction time, however, the stream-based operators cannot even determine the first answer in that collection. This is because the definition of $\mathit{always\text{-}five}_\perp$ is left recursive. The fern-based operators, however, compute as many answers as desired:

```
$(\mathit{run}_\perp\ 4\ (\mathit{always\text{-}five}_\perp\ x)) \Rightarrow (\{x/5\}\ \{x/5\}\ \{x/5\}\ \{x/5\})$.
```

Chapter 15

Implementation VI: Ferns

In this chapter we present a complete, portable, R^6RS compliant (Sperber et al. 2007) implementation of ferns[1]. We begin with a description of *engines* (Haynes and Friedman 1987), which we use to handle suspended, preemptible computations. We then describe and implement frons pairs, the building blocks of ferns. Next we present $car_\perp$ and $cdr_\perp$, which work on both frons pairs and cons pairs. Taking the $car_\perp$ of a frons pair involves choosing one of the possible values in the fern and promoting the chosen value. Taking the $cdr_\perp$ of a frons pair ensures the first value in the pair is determined and returns the rest of the fern. Taking the $car_\perp$ ($cdr_\perp$) of a cons pair is the same as taking its *car* (*cdr*).

15.1 Engines

An engine is a procedure that computes a delayed value in steps[2]. To demonstrate the use of engines, consider the procedure

```
(define wait
  (λ_t (n)
    (cond
      ((zero? n) done)
      (else (wait (− n 1))))))
```

To create an engine *e* to delay a call to (*wait* 20), we write

```
(define e (engine (wait 20)))
```

To partially compute (*wait* 20), we call *e* with a number of *ticks*: (*e* 5), which returns either a pair with false in the car and a new advanced engine (one advanced 5 ticks) in the cdr or a pair with unused ticks (always true) in the car and the value of the

[1] The ferns library is available at `http://www.cs.indiana.edu/~webyrd/ferns.html`

[2] See Appendix D for our implementation of nestable engines.

computation (here done) in the cdr. In our embedding of engines, a tick is spent on each call to a procedure defined with λ_t. Consider

```
(let loop ((p (e 5)))
  (cons (car p) (if (car p) (list (cdr p)) (loop ((cdr p) 5)))))
⇒ (#f #f #f #f 4 done).
```

In this example, (*wait* 20) calls *wait* a total of 21 times (including the initial call), so on the fifth engine invocation, it terminates with 4 unused ticks.

The delayed computation in an engine may involve creating and calling more engines. When a *nested engine* (Hieb et al. 1994) consumes a tick, every frons-enclosing engine also consumes a tick. To see this, we define **choose**$_\perp$ using engines:

```
(define-syntax choose⊥
  (syntax-rules ()
    ((_ exp1 exp2) (choose-aux⊥ (engine exp1) (engine exp2)))))

(define choose-aux⊥
  (λt (e1 e2)
    (let ((p (e1 1)))
      (if (car p) (cdr p) (choose-aux⊥ e2 (cdr p))))))
```

Nested calls to **choose**$_\perp$, for example (**choose**$_\perp$ v_1 (**choose**$_\perp$ v_2 v_3)), rely on nestable engines. This implementation of **choose**$_\perp$ is fair because our embedding of nested engines is fair: every tick given to the second engine in the outer call to *choose-aux*$_\perp$ is passed on to exactly one of the engines, alternating between the engines for v_2 and v_3, in the inner call to *choose-aux*$_\perp$.

15.2 The Ferns Data Type

We represent a frons pair by a cons pair that contains at least one tagged engine (*te*). Engines are tagged with either L when locked (being advanced by another computation) or U when unlocked (runnable). We distinguish between locked and unlocked engines because the *car*$_\perp$ of a fern may be requested more than once simultaneously. Thus, to manage effects, the locks prevent the same engine from being advanced in more than one computation[3].

We define simple predicates $L_a?$, $U_a?$, $L_d?$, and $U_d?$ for testing whether one side of a frons pair contains a locked or unlocked engine.

```
(define engine-tag-compare
  (λ (get-te tag)
    (λ (q)
      (and (pair? q) (pair? (get-te q)) (eq? (car (get-te q)) tag)))))
```

[3] A lock creates a localized critical region that corresponds to the intended use of *sting-unless* (Friedman and Wise 1978).

```
(define L_a? (engine-tag-compare car L))
(define U_a? (engine-tag-compare car U))
(define L_d? (engine-tag-compare cdr L))
(define U_d? (engine-tag-compare cdr U))
```

The procedure *coax*$_d$ (*coax*$_a$) takes a frons pair with an unlocked tagged engine in the cdr (car) and locks and advances the tagged engine by *nsteps* ticks. If *coax*ing (Friedman and Wise 1979) the engine does not finish, the tagged engine is unlocked and updated with the advanced engine. If *coax*ing the engine finishes with value v, then v becomes the frons pair's cdr (car). In addition, the tagged engine will be updated with an unlocked dummy engine that returns v. We do this because the cdrs of multiple frons pairs may share a single engine, as will be explained at the end of this section. Although the cars of frons pairs never share engines, we do the same for the cars.

```
(define coaxer
  (λ (get-te set-val!)
    (λ (q)
      (let ((te (get-te q)))
        (set-car! te L)
        (let ((p (coax (cdr te))))
          (let ((b (car p)) (v (cdr p)))
            (when b (set-val! q v))
            (replace! te U (if b (engine v) v))))))))

(define coax (λ (e) (e nsteps)))
(define coax_a (coaxer car set-car!))
(define coax_d (coaxer cdr set-cdr!))

(define replace!
  (λ (p a d)
    (set-car! p a)
    (set-cdr! p d)))
```

Now we present the implementation of the fern operators.

15.3 cons$_\perp$, *car*$_\perp$, and *cdr*$_\perp$

cons$_\perp$ constructs a frons pair by placing unlocked engines of its unevaluated operands in a cons pair.

```
(define-syntax cons_⊥
  (syntax-rules ()
    ((_ a d) (cons (cons U (engine a)) (cons U (engine d))))))
```

When the *car*$_\perp$ (definition below), which is defined only for ferns, is requested, parallel evaluation of the possible values is accomplished by a round-robin race of the engines in the fern. During its turn, each engine is advanced a fixed, arbitrary number of ticks until a value is produced. The race is accomplished by two mutually recursive functions: $race_a$, which works on the possible values of the fern, and $race_d$, which moves onto the next frons pair by either following the cdr of the current frons pair or starting again at the beginning.

```
(define $car_\bot$
  ($\lambda_t$ (p)
    (letrec (($race_a$
               ($\lambda_t$ (q)
                 (cond
                   (($L_a$? q) (wait nsteps) ($race_d$ q))
                   (($U_a$? q) ($coax_a$ q) ($race_d$ q))
                   ((not (pair? q)) ($race_a$ p))
                   (else (promote p) (car p)))))
             ($race_d$
               ($\lambda_t$ (q)
                 (cond
                   (($L_d$? q) ($race_a$ p))
                   (($U_d$? q) ($coax_d$ q) ($race_a$ p))
                   (else ($race_a$ (cdr q)))))))
      ($race_a$ p))))
```

$race_a$ dispatches on the current pair or value q. When the car of q is a locked engine, $race_a$ waits for it to become unlocked by waiting *nsteps* ticks and then calling $race_d$. The call to *wait* is required to allow $race_a$ to be preempted at this point, so the owner of the lock does not starve. When the car is an unlocked engine, $race_a$ advances the unlocked engine *nsteps* ticks, then continues the race by calling $race_d$. When q is not a pair, $race_a$ simply starts the race again from the beginning. This happens when racing over a finite fern and emerges from the **else** clause of $race_d$. When the car contains a value, we call *promote* which ensures a value is promoted to the car of p, then return that value.

One subtlety of the definition of $race_a$ is that after coaxing an engine it does not check if the coaxing has led to completion. If it has, the value will be picked up the next time the race comes around, if necessary. Calling *promote* immediately would be incorrect because an engine may be preempted while advancing, at which point promotion from p may be performed by another computation with a different value for the car of p.

$race_d$ also dispatches on q, this time examining its cdr. When the cdr of q is a locked engine, $race_d$, being unable to proceed further down the fern, restarts the race by calling $race_a$ on p. When the cdr of q contains an unlocked engine, $race_d$ advances the engine *nsteps* ticks as in $race_a$, and then restarts the race. If that engine finishes with a new frons pair, the new pair will then be competing in the race and will be examined next time around. When the cdr of q is a value, usually a fern, $race_d$ continues the race by passing it to $race_a$; if a non-pair value is at the end of a fern, it will be picked up by the third clause in $race_a$.

$car_\perp$ avoids starvation by running each engine in a subfern for the same number of ticks. During a race, a subfern of the fern in question is in a fair state: for some (potentially empty) prefix of the subfern there are no engines in the cdrs, so each potential value in a fair subfern is considered equally. When this fair subfern is not the entire fern, the race devotes the same number of ticks to lengthening the fair subfern as it does to each element of that subfern. Since cdr engines often evaluate to pairs quickly, the entire fern usually becomes fair in a number of races equal to the length of the fern. When cdr engines do not finish quickly, however, the process of making the entire fern fair can take much longer, especially for long ferns. The cost of finding the value of an element occurring near the end of such a fern can be much greater than the cost for an element near the beginning.

Starting from p, *promote* (definition below) finds the first pair r whose car contains a convergent value, and propagates that value back to p. Each frons pair in this chain (excluding r) is transformed into a cons pair whose car is the convergent value. These new frons pairs are connected as a fern and the last one shares r's cdr. When *promote* is called from $race_a$, we know that q's car is a value but we don't know for certain that there is no other pair, say r, in the chain from p to q. Thus, we must search from p without preemption to find the closest value to p. This situation can arise when there are two calls to $car_\perp$ on the same fern competing:

```
(let ((α (list⊥ (! 5) (! 6))))
  (car⊥ (list⊥ ⊥ (car⊥ α) ⊥ (car⊥ α) ⊥)))
```

If a call to $race_a$ finds the value 720 and tries to promote it, but the value 120 has already been promoted, we don't want to change the car of α. Instead, the call to *promote* when 720 is found will find the 120 first and stop.

```
(define-syntax let_t
  (syntax-rules ()
    ((_ ((x e) ...) b_0 b ...) ((λ_t (x ...) b_0 b ...) e ...))))

(define promote
  (λ_t (p)
    (cond
      ((L_a? p) (wait nsteps) (promote p))
      ((U_a? p)
       (set-car! (car p) L)
       (let_t ((te (car p)))
         (let_t ((r (promote (cdr p))))
           (replace! p (car r) (cons te (cdr r)))
           (set-car! te U)
           p)))
      (else p))))
```

The $cdr_\perp$ of a fern (definition below) cannot be determined until the fern's $car_\perp$ has been determined. Once the car has been determined, there is no longer parallel competition between potential cdrs. Thus, we can use cdr_s, which takes the cdr of a stream. Then, since *p*'s car has been determined, *p* has therefore become a cons pair, so $cdr_\perp$ returns the value in *p*'s cdr. (car_s's definition follows by replacing all *d*s by *a*s. $\mathbf{cons}_s$ is the same as $\mathbf{cons}_\perp$, and the definitions of the other stream operators follow the definitions with operators $f_\perp$ replaced by f_s.)

```
(define $cdr_\perp$ ($\lambda_t$ (p) ($car_\perp$ p) ($cdr_s$ p)))

(define $cdr_s$
  ($\lambda_t$ (p)
    (cond
      (($L_d$? p) (wait nsteps) ($cdr_s$ p))
      (($U_d$? p) ($coax_d$ p) ($cdr_s$ p))
      (else (cdr p)))))
```

If the engine being advanced by cdr_s completes, cdr_s indicates that $coax_d$ should replace the tagged engine in *p* by the computed value. However, $race_d$ and $cdr_\perp$ are required not only to update the frons pair with the calculated value, but also to update the tagged engine because there might be a fern other than *p* sharing this engine. Consider the following expression where we assume *list* evaluates its arguments from left to right.

```
(let (($\beta$ ($\mathbf{cons}_\perp$ 1 ($ints\text{-}from_\perp$ 2))))
  (let (($\alpha$ ($\mathbf{cons}_\perp$ $\perp$ $\beta$)))
    (list ($car_\perp$ $\alpha$) ($cadr_\perp$ $\beta$) ($cadr_\perp$ $\alpha$))))
⇝ (1 2 2)
```

Figure 15.1 shows the data structures involved in evaluating the expression.
Figure 15.1a shows α immediately after it has been constructed, with engines delaying evaluation of $\perp$ and β. In evaluating ($car_\perp$ α), the engine for β finishes, resulting in Figure 15.1b. β can now participate in the race for ($car_\perp$ α). Suppose the value 1 found in the car of β is chosen and promoted. The result is Figure 15.1c, in which the engine delaying ($ints\text{-}from_\perp$ 2) is shared by both β and the cdr of α. ($cadr_\perp$ β) forces calculation of ($ints\text{-}from_\perp$ 2), which results in a fern, γ, whose first value (in this example) is 2. Figure 15.1d now shows why $coax_d$ updates the current pair (β) and creates a new dummy engine with the calculated value (γ): the cddr of α needs the new engine to avoid recalculation of ($ints\text{-}from_\perp$ 2). In Figure 15.1e when ($cadr_\perp$ α) is evaluated, the value 2, calculated already by ($cadr_\perp$ β), is promoted and the engine delaying ($ints\text{-}from_\perp$ 3) is shared by both α and β.

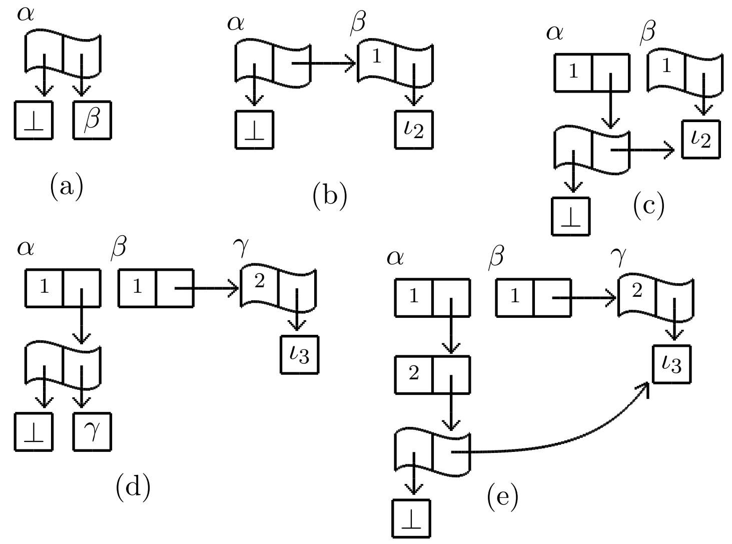

Figure 15.1: Fern α after construction (a); after β in the cdr of α has been evaluated (b); after 1 from the car of β has been promoted to the car of α, resulting in a shared tagged engine (c); after the shared engine is run, while evaluating ($cadr_\perp$ β), to produce a fern γ (d); after 2 from the car of γ has been promoted to the cadr of α (e).

Part VI

Context and Conclusions

Chapter 16

Related Work

This chapter describes some of the work by other researchers that is related to the research presented in this dissertation.

Lloyd (1987) is the standard work on the theoretical foundations of logic programming; Doets (1994) has written a more recent introduction to the theory of logic programming.

The most popular logic programming language is Prolog (Intl. Organization for Standardization 1995, 2000). Clocksin and Mellish (2003) have written one of the most popular introductions to the language. Prolog was designed by Colmerauer (Colmerauer 1985, 1990); Colmerauer and Roussel (1996) describe the early history of Prolog.

Most modern implementations of Prolog are based on the Warren Abstract Machine (WAM) (Warren 1983); Aït-Kaci (1991) presents a tutorial reconstruction of the WAM. Van Roy (1994) describes in detail the first decade of sequential Prolog implementation techniques after the invention of the WAM.

Apt has advocated using Prolog for declarative programming (1993); unfortunately, Prolog's design and implementation encourages the use of cut and other non-logical features. For example, Naish (1995) argues that Prolog programming without cut is impractical.

There is a long tradition of embedding logic programming operators in Scheme (Ruf and Weise 1990; Sitaram 1993; Felleisen 1985; Abelson and Sussman 1996; Bonzon 1990; Haynes 1987). Most of this work was done during the mid-1980's to early-1990's, and most of these embeddings can be seen as attempts to combine Prolog's unification and backtracking search with Scheme's lexical scope and first-class functions. Similarly, there have been attempts to embed logic programming in other functional languages, such as Lisp (Robinson and Sibert 1982; Cattaneo and Loia 1988; Nayak 1989; Komorowski 1979; Kahn and Carlsson 1984) and Haskell (Spivey and Seres 1999; Seres and Spivey 2000; Spivey and Seres 2003; Claessen and Ljunglöf 2000; Todoran and Papaspyrou 2000). However, the extent to which these languages truly integrate functional programming and logic programming is debatable; as with miniKanren, these embeddings are not functional logic programming languages in the modern sense; they do not provide higher-order unification or higher-order pattern matching, as in λProlog (Nadathur and Miller 1988; Nadathur 2001), nor do they use narrowing or residuation.

Two modern languages that combine logic programming with functional programming are Mercury (Somogyi et al. 1995) and Curry (Hanus et al. 1995). The syntax and type systems of both languages are inspired by Haskell.

The Mercury compiler uses programmer-supplied type, mode, and determinism annotations to compile each goal into multiple functions. this results in very efficient code, which is essential to the Mercury team's objective of facilitating declarative programming "in-the-large". Unfortunately, this emphasis on the efficiency comes at the expense of relational programming—forcing, or even permitting, a programmer to explicitly specify an argument's mode as "input" or "output" is the antithesis of relational programming.

The Curry language takes a different approach, integrating functional and logic programming through the single implementation strategy of narrowing (Antoy et al. 2001); that is, lazy term rewriting, with the ability to instantiate logic variables. Curry also supports residuation, which allows a goal to suspend if its arguments are not sufficiently instantiated. For example, a goal that performs addition might suspend if its first two arguments are not ground. While residuation is a useful language feature, it inhibits relational programming since the program will diverge if the arguments never become instantiated.

miniKanren is the descendant of Kanren (Friedman and Kiselyov 2005), another embedding of logic programming in Scheme. Kanren is closer in spirit to Prolog than is miniKanren. Philosophically, Kanren was designed for efficiency rather than for relational programming. Kanren supports neither nominal logic, disequality constraints, nor tabling. Kanren allows programmers to easily extend existing relations[1].

Sokuza Kanren is a minimal embedding of logic programming in Scheme; it is essentially a stripped down version of the core miniKanren implementation from Chapter 3[2].

16.1 Purely Relational Arithmetic

Chapter 6 presents a purely relational binary arithmetic system.

We first presented arithmetic predicates over binary natural numbers (including division and logarithm) in a book (Friedman et al. 2005). That presentation had no detailed explanations, proofs, or formal analysis; this was the focus of a later paper (Kiselyov et al. 2008) that presented the arithmetic relations in Prolog rather than

[1] This can be done in miniKanren as well, through the technique of function extension. However, Kanren provides an explicit form for extending a relation.

[2] For example, Sokuza Kanren does not include a reifier.

miniKanren. A lengthier, unpublished version of this paper[3] includes appendices containing additional proofs.

Braßel, Fischer, and Huch's paper (2007) appears to be the only previous description of declarative arithmetic. It is a practical paper, based on the functional logic language Curry. It argues for declaring numbers and their operations in the language itself, rather than using external numeric data types and operations. It also uses a little-endian binary encoding of positive integers (later extended to signed integers).

Whereas our implementation of arithmetic uses a pure logic programming language, Braßel, Fischer, and Huch use a non-strict functional-logic programming language. Therefore, our implementations use wildly different strategies and are not directly comparable. Also, we implement the logarithm relation.

Braßel, Fischer, and Huch leave it to future work to prove termination of their predicates. In contrast, we have formulated and proved decidability of our predicates under interleaving search (as used in miniKanren) and depth-first search (used in Prolog).

Our approach is minimalist and pure; therefore, its methodology can be used in other logic systems—specifically, Haskell's type classes. Hallgren (2001) first implemented (unary) arithmetic in such a system, but with restricted modes. Kiselyov (2005, §6) treats decimal addition more relationally. Kiselyov and Shan (2007) first demonstrated all-mode arithmetic relations for arbitrary binary numerals, to represent numerical equality and inequality constraints in the type system. Their type-level declarative arithmetic library enables resource-aware programming in Haskell with expressive static guarantees.

16.2 αKanren

αKanren, presented in Chapters 9 and 11, is a nominal logic programming language; it was based on both miniKanren and αProlog (Cheney 2004a; Cheney and Urban 2004).

Early versions of αProlog implemented equivariant unification (Cheney 2005), which allows the permutations associated with suspensions to contain logic variables. The expense of equivariant unification (Cheney 2004b) led Urban and Cheney to replace full equivariant unification with nominal unification (Urban and Cheney 2005). Cheney's dissertation presents numerous examples of nominal logic programming in αProlog (Cheney 2004a).

MLSOS (Lakin and Pitts 2008) is another nominal logic language, designed for easily expressing the rules and side-conditions of Structured Operational Semantics (Plotkin 2004). MLSOS uses nominal unification, and introduces *name constraints*, which are essentially disequality constraints restricted to noms (or to suspensions that will become noms).

[3] `http://okmij.org/ftp/Prolog/Arithm/arithm.pdf`

Nominal logic was introduced by Pitts (2003). Nominal functional languages include FreshML (Shinwell et al. 2003), Fresh O'Caml (Shinwell 2006), and Cαml (Pottier 2006).

The first nominal unification algorithm was presented and proved correct by Urban et al. (2004); the algorithm was described using idempotent substitutions.

A naive implementation of the Urban et al. algorithm has exponential time complexity; however, by representing nominal terms as graphs, and by lazily pushing in swaps, it is possible to implement a polynomial-time version of nominal unification (Calvès and Fernández 2008; Calvès and Fernández 2007).

More recently, Dowek et al. (2009) presented a variant of nominal unification using "permissive" nominal terms, which do not require explicit freshness constraints. To our knowledge, there are no programming languages that currently support permissive nominal terms.

16.3 αlean*TAP*

The αlean*TAP* relational theorem prover presented in Chapter 10 is based on lean*TAP*, a lean tableau-based prover for first-order logic due to Beckert and Posegga (1995).

Through his integration of lean*TAP* with the Isabelle theorem prover, Paulson (1999) shows that it is possible to modify lean*TAP* to produce a list of Isabelle tactics representing a proof. This approach could be reversed to produce a proof translator from Isabelle proofs to αlean*TAP* proofs, allowing αlean*TAP* to become interactive as discussed in section 10.2.2.

The lean*TAP* Frequently Asked Questions (Beckert and Posegga) states that lean*TAP* might be made declarative through the elimination of Prolog's cuts but does not address the problem of `copy_term/2` or specify how the cuts might be eliminated. Other provers written in Prolog include those of Manthey and Bry (1988) and Stickel (1988), but each uses some impure feature and is thus not declarative.

Christiansen (1998) uses constraint logic programming and metavariables (similar to nominal logic's names) to build a declarative interpreter based on Kowalski's non-declarative `demonstrate` predicate (Kowalski 1979). This approach is similar to ours, but the Prolog-like language is not complicated by the presence of binders.

Higher-order abstract syntax (HOAS), presented in Pfenning and Elliot (1988), can be used instead of nominal logic to perform substitution on quantified formulas. Felty and Miller (1988) were among the first to develop a theorem prover using HOAS to represent formulas; Pfenning and Schurmann (1999) also use a HOAS encoding for formulas.

Kiselyov uses a HOAS encoding for universally quantified formulas in his original translation of lean*TAP* into miniKanren (Friedman and Kiselyov 2005). Since miniKanren does not implement higher-order unification, the prover cannot generate theorems.

Lisitsa's λlean*TAP* (2003) is a prover written in λProlog that addresses the problem of `copy_term/2` using HOAS, and is perhaps closest to our own work. Like αlean*TAP*, λlean*TAP* replaces universally quantified variables with logic variables using substitution. However, λlean*TAP* is not declarative, since it contains cuts. Even if we use our techniques to remove the cuts from λlean*TAP*, the prover does not generate theorems, since λProlog uses a depth-first search strategy. Generating theorems requires the addition of a tagging scheme and iterative deepening on *every clause* of the program. Even with these additions, however, λlean*TAP* often generates theorems that do not have the proper HOAS encoding, since that encoding is not specified in the prover.

16.4 Tabling

Tabling is essentially an efficient way to find fixed points. Tabling can be used to implement model checkers, abstract interpreters, deductive databases, and other useful programs that must calculate fixed points (Guo and Gupta 2009; Warren 1992).

Many Prolog implementations support some form of tabling. XSB Prolog (Sagonas et al. 1994), which uses SLG Resolution (Chen and Warren 1996) and the SLG-WAM abstract machine (Sagonas and Swift 1998), remains the standard testbed for advanced tabling implementation. Our implementation was originally inspired by the Dynamic Reordering of Alternatives (DRA) approach to tabling (Guo and Gupta 2009, 2001).

16.5 Ferns

Chapter 14 describes ferns, a shareable, bottom-avoiding data structure invented by Friedman and Wise (1981). Chapter 15 presents our shallow embedding of ferns in Scheme.

Previous implementations of ferns have been for a call-by-need language. The work of Friedman and Wise (1979, 1980, 1981) presumes a deep embedding whereas our approach is a shallow embedding. The function *coax* is taken from their conceptualization (Friedman and Wise 1979):

> COAX is a function which takes a suspension as an argument and returns a field as a value; that field may have its *exists* bit *true* and its pointer referring to its *exist*ent value, or it may have its *exists* bit false and its pointer referring to another suspension.

Thus, engines are a user-level, first-class manifestation of suspensions where *true* above corresponds to the unused ticks. Johnson's master's thesis (1977) under Friedman's direction presents a deep embedding in Pascal for a lazy ferns language.

Subsequently, Johnson and his doctoral student Jeschke implemented a series of native C symbolic multiprocessing systems based on the Friedman and Wise model. This series culminated with the parallel implementation Jeschke describes in his dissertation (Jeschke 1995). In their *Daisy* language, ferns are the means of expressing explicit concurrency (Johnson 1983).

Chapter 17

Future Work

In this chapter we propose future work related to miniKanren, and to relational programming in general.

This chapter is organized as follows. In section 17.1 we discuss how our work on miniKanren might be formalized. Section 17.2 presents possible improvements to the existing miniKanren implementation, while section 17.3 suggests how the miniKanren language might be extended. Section 17.4 considers future work on relational idioms, while section 17.5 proposes future applications of miniKanren. Finally, in section 17.6 we propose tools that might ease the burden on relational programmers.

17.1 Formalization

From a formalization standpoint, the most important future work is to create a formal semantics for miniKanren. Perhaps the simplest approach would be to start from the operational semantics of the nominal logic programming language MLSOS, as described in Lakin and Pitts (2008). Of course, miniKanren's semantics would become more complex if the language extensions proposed in section 17.3 were added. Indeed, it is the interaction between different language features (nominal unification and constraint logic programming, for example) that will make extending miniKanren challenging.

The core miniKanren implementation presented in Chapter 3 uses a stream-based interleaving search strategy. The use of **inc**s (thunks) to force interleaving makes it difficult to exactly characterize the search behavior, and therefore the order in which miniKanren produces answers. It would be both interesting and useful to mathematically describe this interleaving behavior (see section 17.2).

In Chapter 10 we replaced `lean`*TAP*'s use of Prolog's `copy_term/2` with a purely declarative combination of tagging and nominal unification; this technique was key to making α`lean`*TAP* purely relational. Unfortunately, this approach can only be used when the programmer knows the structure of the terms to be copied. It would be useful to formalize this technique, to better understand its applicability and limitations.

The relational arithmetic system presented in Chapter 6 uses bounds on term sizes to provide strong termination guarantees for arithmetic relations[1]. A systematic approach to deriving such bounds on term sizes would be very helpful for relational programmers. Of course, Gödel and Turing showed that it is impossible to guarantee termination for all goals we might wish to write, so in general we will not be able to achieve finite failure through bounds, or any other technique[2]. However, even when such bounds exist, it may be difficult to express them in miniKanren. Indeed, poorly expressed bounds may themselves cause divergence—for example, by attempting to eagerly determine the length of an uninstantiated (and therefore unbounded) list[3]. A systematic approach to expressing bounds already derived by the programmer would be most useful.

Section 11.4 presents a Scheme implementation of a nominal unifier that uses triangular substitutions. This algorithm should be formalized and proved correct, similar to the presentation of (idempotent) nominal unification in Urban et al. (2004).

Herman and Wand (2008) use nominal logic to describe an idealized version of Scheme's **syntax-rules** hygienic macro system. It would be interesting to extend this work to the full **syntax-rules** system, perhaps by implementing the macro system as an αKanren relation.

A more speculative area of future work is the connection between the various causes of divergence described in Chapter 5. As discussed in the conclusion of this dissertation, there may be a deep connection between these causes of divergence, and between the techniques for avoiding them. Since divergence is an effect, monads (Moggi 1991) or arrows (Hughes 1998) may provide the best framework for exploring these ideas.

17.2 Implementation

The core miniKanren implementation presented in Chapter 3 uses streams to implement backtracking search[4]. As described in Wand and Vaillancourt (2004), our use of streams could be modelled using explicit success and failure continuations. When extending the miniKanren language, it is sometimes more convenient to use this two-continuation model of backtracking—for example, the first implementation of tabling for miniKanren used continuations rather than streams.

[1] At least, for single arithmetic relations whose arguments do not share unassociated logic variables.

[2] For example, the strong termination guarantees for our arithmetic system do not hold for conjunction of addition and multiplication goals.

[3] See Chapter 5 for more on the difficulty of expressing bounds on term sizes.

[4] Although one could argue that the stream-based implementation performs backtracking search without actually backtracking.

The streams implementation of miniKanren makes liberal use of **inc**s (thunks) to force interleaving in the search. Unfortunately, it is difficult to exactly replicate this interleaving search behavior in the two-continuation model. As a result, continuation-based implementations of miniKanren may produce answers in a different order than stream-based implementations, which makes it difficult to test, benchmark, or otherwise compare different implementations. It therefore would be extremely convenient to have a continuation-based implementation of miniKanren that exactly mirrors the search behavior of the streams-based implementation from Chapter 3. This may require a formal characterization of the stream-based search strategy, as discussed in section 17.1.

We currently use association lists to represent substitutions; we may wish to consider other purely functional representations of substitutions that would make variable lookup less expensive. For example, Abdulaziz Ghuloum previously implemented a trie-based representation of substitutions that performs at least as well as the fastest walk-based algorithm presented in Chapter 4. Using a trie-based representation of substitutions may mean giving up on the clever method of implementing disequality constraints described in Chapter 8.

Relational programming is inherently parallelizable. In fact, we have already implemented two parallel versions of miniKanren: one written in Scheme and one in Erlang (Armstrong 2003). However, neither parallel implementation runs as quickly as the sequential implementation of miniKanren presented in Chapter 3. One difficulty in making a parallel implementation run efficiently is that miniKanren suffers from an "embarrassment of parallelism". For example, a recursive goal might contain a **cond**^{e} whose first clause contains a single unification. The overhead of sending this single unification to a new core or processor may be more expensive than just performing the unification. Ciao Prolog solves this problem by performing a "granularity analysis" to determine which parts of a program perform enough computation to offset the overhead of parallelization (Debray et al. 1990; Lopez et al. 1996).

Our purely functional implementation of miniKanren also implies a different set of design choices than would be made when parallelizing a Prolog implementation based on the Warren Abstract Machine. In particular, our stream-based search implementation, combined with our functional representation of substitutions[5], means that disjunction is truly parallel: failure of one disjunct does not require communication with other disjuncts.

Reification of nominal terms is another area for future work. The core-miniKanren reifier presented in Chapter 3 enforces several important invariants: swapping adjacent calls to $\equiv$, swapping arguments within a single call to $\equiv$, or reordering nested **exist** clauses[6] cannot affect reified answers. We would like αKanren to ensure simi-

[5] Gupta and Jayaraman (1993) have explored the tradeoffs of different environment representations in the context of parallel logic programming.

[6] Assuming this is done without inadvertently shadowing variables, or leaving previously bound

lar invariants; however, reification in αKanren is more complicated, since each term containing a $\bowtie$ now represents an infinite equivalence class of α-equivalent terms. Additionally, we do not have a canonical representation for permutations associated with suspensions. Finally, reification must also handle freshness constraints.

miniKanren uses a complete interleaving search strategy, which ensures disjunction (**cond**^{e}) is commutative—swapping the order of **cond**^{e} clauses can affect the order in which answers are returned, but cannot affect whether a goal diverges. In contrast, miniKanren's conjunction operators (**exist** and **fresh**) are not commutative—swapping conjuncts can cause a goal that previously failed finitely to now diverge. It is easy to see that commutative conjunction can be implemented: just run in parallel every possible ordering of conjuncts. Unfortunately, this simplistic approach is far too expensive to be used in practice. However, it may be possible to more efficiently implement commutative conjunction by interleaving the evaluation of conjuncts, and allowing each conjunct to partially extend the substitution. This would allow conjuncts to communicate with each other by extending the substitution, thereby allowing the conjunction to "fail fast", and avoiding the duplication of work inherent in the naive approach described above. It is not clear whether this approach is efficient enough to be used throughout an entire program; the programmer may need to restrict use of commutative conjunction to conjunctions containing multiple recursive goals.

Alternatively, it may be possible to *simulate* commutative conjunction using a combination of continuations, interleaving search, and tabling. This approach would only be a simulation of true commutative conjunction because tabling is defeated if an argument changes with each recursive call.

The core miniKanren implementation presented in Chapter 3 is an embedding in Scheme, using a combination of procedures and hygienic macros. Although this embedding allows us to easily benefit from the optimizations provided by a host Scheme implementation, we lose the ability to analyze or transform entire miniKanren programs. A miniKanren compiler would allow us to perform more sophisticated program analyses. Finally, a miniKanren interpreter[7] or abstract machine would be useful from both an implementation and formalization standpoint.

17.3 Language Extensions

αKanren's support for nominal logic programming could be extended in several ways. Perhaps the simplest extension would be to add MLSOS's name inequality constraint (Lakin and Pitts 2008), which is essentially a disequality constraint limited to noms (and to suspensions that will become noms). A more ambitious extension would be to add full disequality constraints to αKanren. One might

variables unbound.

[7] In the long tradition of writing meta-circular Scheme interpreters, a meta-circular miniKanren interpreter would be especially satisfying.

also implement equivariant unification (Cheney 2005), which extends nominal unification with the ability to include logic variables in permutations; however, the expense of equivariant unification (Cheney 2004b) limits its appeal[8]. Dowek et al. (2009) recently presented a variant of nominal unification using “permissive” nominal terms, which do not require explicit freshness constraints; permissive nominal terms might simplify reification of αKanren answers.

Our tabling implementation does not currently work with disequality constraints or freshness constraints. It would be very useful to extend tabling to work with these constraints. Alternatively, it may be possible to add tabling to αKanren by using permissive nominal terms, which do not require freshness constraints.

Gupta et al. (2007) have implemented a coinductive logic programming language that can express infinite streams using coinductive definitions of goals. The heart of their system is an implementation of tabling, in which unification rather than reification is used to determine whether a call is a variant of an already tabled call. It should be straightforward to add coinductive logic programming to miniKanren, since we have already implemented tabling. Also, it would be interesting to investigate if other notions of variant calls make sense—for example, what if we used subsumption instead of reification or unification? Would we get a different type of logic programming? Finally, the streams that can be created using the system of Gupta et al. must have a regular structure—for example, their system cannot represent a stream of all the prime numbers. How might more sophisticated streams be expressed?

One alternative to requiring the occurs check for sound unification is to allow infinite terms, as in Prolog II. This would require changing the reifier to print circular terms. We would also want our core language forms, such as disequality constraints, to handle infinite terms[9].

An extremely useful extension to miniKanren would be the addition of constraint logic programming, or CLP (Jaffar and Maher 1994)[10]. The notation ‘CLP(X)’ refers to constraint logic programming over some domain ‘X’; common domains include the integers (CLP(Z)), rational numbers (CLP(Q)), real numbers (CLP(R)), and finite domains (CLP(FD)). Most useful for existing applications of miniKanren would be CLP(FD) and CLP(Z), which would allow us to declaratively express arithmetic in a more efficient manner than the arithmetic system of Chapter 6[11].

[8] Although Urban and Cheney (2005) show that it is often possible to avoid full equivariant unification in real programs.

[9] SWI Prolog (Wielemaker 2003) includes many predicates that work on infinite terms, and might serve as an inspiration.

[10] Actually, miniKanren and αKanren already support several types of constraints: unification ($\equiv$) and dis-unification ($\neq$) constraints, and the freshness constraints of nominal logic. However, there are many other types of constraints we might want to add.

[11] The declarative arithmetic system of Chapter 6 has several advantages over the constraint approach, however. As opposed to CLP(FD), our system works on numbers of arbitrary size. Our system is also implemented entirely at the user-level language, without any constraints other than unification, while adding CLP(FD) or CLP(Z) requires significant changes to the underlying

miniKanren, like Scheme, is dynamically-typed. Siek and Taha (2006) show how *gradual typing* can be used to add a sophisticated type system to a dynamically typed language, without giving up the flexibility of dynamic typing[12]. It would be interesting to apply this typing scheme to miniKanren, since supporting logic variables and constraints may require extending the notions of gradual typing.

Relational goals often append two lists; if the first list is an uninstantiated logic variable, this results in infinitely many answers, which can easily lead to divergence. It may be possible to create an *append* constraint that represents the delayed appending of two lists, and avoids enumerating infinitely many appended lists.

Another line of future work would be to implement non-standard logics for relational programming, such as temporal logic, linear logic, and modal logic. Of course, supporting any of these logics would require significant changes to miniKanren, and would require careful consideration of how various language extensions would interact with the new logic.

Modern functional logic programming languages like Curry are based on narrowing (Antoy et al. 2001), which combines term rewriting with the ability to instantiate logic variables. It would be interesting to implement a language based on *nominal* narrowing—that is, narrowing based on nominal rewriting (Fernández and Gabbay 2007). This would allow a single implementation to express nominal functional programming (as in FreshML (Shinwell et al. 2003) or Cαml (Pottier 2006)), nominal logic programming (as in αProlog (Cheney and Urban 2004), MLSOS (Lakin and Pitts 2008), or αKanren), hygienic macros (as in Scheme[13]), and nominal term rewriting (as in Maude (Clavel et al. 2003), Stratego (Visser 2001), or PLT Redex (Matthews et al. 2004), but with the addition of nominal logic).

Like MLSOS and αProlog, αKanren is well suited for expressing the rules and side-conditions of Structural Operational Semantics (SOS) (Plotkin 2004). It would be informative to explore which SOS rules or side-conditions *cannot* be easily expressed in αKanren; such an exercise would likely result in new constraints and other language extensions. Similarly, it would be informative to investigate which Scheme, Prolog, and Curry programs we cannot satisfactorily express in a purely relational manner.

Perhaps the greatest challenge in extending miniKanren is to combine all of these language features in a meaningful way. Ciao Prolog attempts to control interactions between language features through a module system (Gras and Hermenegildo 1999). The addition of libraries to the *R^6RS* Scheme standard (Sperber et al. 2007) should allow us to do the same. However, a more sophisticated approach based on monads and monad transformers may better control the interaction between language

implementation, and may interact in undesirable ways with other language features.

[12] There has also been recent work on adding something like gradual typing to Prolog (see (Schrijvers et al. 2008b), although it is unclear whether these researchers are aware of the Scheme community's work on gradual typing and soft typing (Cartwright and Fagan 1991).

[13] Herman and Wand (2008) describe a simplified version of Scheme's **syntax-rules** macro system using nominal logic.

features.

17.4 Idioms

Okasaki (1999) has investigated the use of purely functional data structures, many of which are comparable in efficiency to imperative data structures[14]. Even more so than in functional programming, data representation is essential to relational programming. Therefore, it would be interesting and useful to investigate the use of *purely relational* data structures—that is, data structures and data representations that are especially well-suited for relational programming. Some of these data structures might take advantage of relational language features such as nominal unification or constraints.

Also, as mentioned in section 17.1, it would be useful to formalize our combination of tagging and nominal unification to emulate Prolog's `copy_term/2` in a purely declarative manner.

17.5 Applications

It should be relatively easy to extend the arithmetic system of Chapter 6 to handle rational numbers. Probably the most difficult part of this exercise would be maintaining fractions in simplified form.

An interesting extension to the type inferencer in section 9.3 would be to support polymorphic-**let** (Pierce 2002). At a minimum, this would require a declarative way to perform a combination of substitution and term copying. Of course, the implementation of αlean*TAP* in Chapter 10 also uses these techniques. However, there may be enough differences between αlean*TAP* and the type inferencer to make applying these techniques difficult or impossible. If so, a new type of constraint may be called for.

As described in Chapter 10, the αlean*TAP* theorem prover allows a user to guide the proof search by partially instantiating the prover's proof-tree argument. It should be possible to extend αlean*TAP*, making it act as a rudimentary interactive proof assistant. This would further demonstrate the flexibility of relational programming; more importantly, creating such a tool might require new techniques that would be useful for writing relational programs in general.

17.6 Tools

As mentioned in section 17.1, integrating bounds on term size into an existing relation can be difficult. A tool that could take a relation, along with a specification

[14] Indeed, uses of purely functional data structures can be even more efficient than uses of imperative data structures, due to sharing.

of bounds on the argument sizes, and synthesize a new relation that incorporates those bounds would be extremely helpful.

A tool to automatically translate Scheme programs to miniKanren would also be handy. Ideally, this tool would generate purely relational miniKanren code adhering to the non-overlapping principle (see section 7.3). This may be possible, at least for many simple Scheme functions, if the programmer were to help the tool by specifying how to represent terms, along with an appropriate tagging scheme. However, deriving miniKanren relations from Scheme functions is not the real difficultly—rather, ensuring finite failure for a wide variety of arguments is what makes relational programming so difficult.

A Prolog-to-Scheme translator would also be useful. Translating pure Prolog programs into miniKanren should be very easy, especially since the λ^e pattern-matching macro is similar to Prolog's pattern matching syntax.

Chapter 18

Conclusions

This dissertation presents the following high-level contributions:

1. A collection of idioms, techniques, and language constructs for relational programming, including examples of their use, and a discussion of each technique and when it should or should not be used.

2. Various implementations of core miniKanren and its variants, which utilize the full power of Scheme, are concise and easily extensible, allow sharing of substitutions, and provide backtracking "for free".

3. A variety of programs demonstrating the power of relational programming.

4. A clear philosophical framework for the practicing relational programmer.

More specifically, this dissertation presents:

1. A novel constraint-free binary arithmetic system with strong termination guarantees.

2. A novel technique for eliminating uses of `copy_term/2`, using nominal logic and tagging.

3. A novel and extremely flexible lean tableau theorem prover that acts as a proof generator, theorem generator, and even a simple proof assistant.

4. The first implementation of nominal unification using triangular substitutions, which is much faster than a naive implementation that follows the formal specification by using idempotent substitutions.

5. An elegant, streams-based implementation of tabling, demonstrating the advantage of embedding miniKanren in a language with higher-order functions.

6. A novel *walk*-based algorithm for variable lookup in triangular substitutions, which is amenable to a variety of optimizations.

7. A novel approach to expression-level divergence avoidance using ferns, including the first shallow embedding of ferns.

The result of these contributions is a set of tools and techniques for relational programming, and example applications informing the use of these techniques.

As stated in the introduction, the thesis of this dissertation is that miniKanren supports a variety of relational idioms and techniques, making it feasible and useful to write interesting programs as relations. The technique and implementation chapters should establish that miniKanren supports a variety of relational idioms and techniques. The application chapters should establish that it is feasible and useful to write interesting programs as relations in miniKanren, using these idioms and techniques.

A common theme throughout this dissertation is divergence, and how to avoid it. Indeed, an alternative title for this dissertation could be, “Relational Programming in miniKanren: Taming $\perp$.”[1] As we saw in Chapter 5, there are many causes of divergent behavior, and different techniques are required to tame each type of divergence. Some of these techniques merely require programmer ingenuity, such as the data representation and bounds on term size used in the arithmetic system of Chapter 6. Other techniques, such as disequality constraints and tabling, require implementation-level support.

Gödel and Turing showed that it is impossible to guarantee termination for every goal we might wish to write. However, this does not mean that we should give up the fight. Rather, it means that we must be willing to thoughtfully employ a variety of techniques when writing our relations—as a result, we can write surprisingly sophisticated programs that exhibit finite failure, such as our declarative arithmetic system. It also means we must be creative, and willing to invent new declarative techniques when necessary—perhaps a new type of constraint or a clever use of nominal logic, for example[2].

Of course, no one is forcing us to program relationally. After trying to wrangle a few recalcitrant relations into termination, we may be tempted to abandon the

[1] With apologies to Olin Shivers.

[2] We can draw inspiration and encouragement from work that has been done on NP-complete and NP-hard problems. Knowing that a problem is NP hard is not the end of the story, but rather the beginning. Special cases of the general problem may be computationally tractable, while probabilistic or approximation algorithms may prove useful in the general case. (A good example is probabilistic primality testing, used in cryptography for decades. Although Agrawal et al. (2002) recently showed that primality testing can be performed deterministically in polynomial time, the potentially fallible probabilistic approach is still used is practice, since it is more efficient.) A researcher in this area must be willing to master and apply a variety of techniques to construct tractable variants of these problems. Similarly, a relational programmer must be willing to master and apply a variety of techniques in order to construct a relation that fails finitely. This often involves trying to find approximations of logical negation (such as various types of constraints).

relational paradigm, and use miniKanren's impure features like **cond**^{a} and **project**. We might then view miniKanren as merely a "cleaner", lexically scoped version of Prolog, with S-expression syntax and higher-order functions. However tempting this may be, we lose more than the flexibility of programs once we abandon the relational approach: we lose the need to construct creative solutions to difficult yet easily describable problems, such as the *rember*^{o} problem in Chapter 7.

The difficulties of relational programming should be embraced, not avoided. The history of Haskell has demonstrated that a commitment to purity, and the severe design constraints this commitment implies, leads to a fertile and exciting design space. From this perspective, the relationship between miniKanren and Prolog is analogous to the relationship between Haskell and Scheme. Prolog and Scheme allow, and even encourage, a pure style of programming, but do not require it; in a pinch, the programmer can always use the "escape hatch" of an impure operator, be it cut, **set!**, or a host of other convenient abominations, to leave the land of purity. miniKanren and Haskell explore what is possible when the escape hatch is welded shut. Haskell programmers have learned, and are still learning, to avoid explicit effects by using an ever-expanding collection of monads; miniKanren programmers are learning to avoid divergence by using an ever-expanding collection of declarative techniques, many of which express limited forms of negation in a bottom-avoiding manner. Haskell and miniKanren show that, sometimes, painting yourself into a corner can be liberating[3].

A final, very speculative observation: it may be possible to push the analogy between monads and techniques for bottom avoidance further. Before Moggi's work on monads (Moggi 1991), the relationship between different types of effects was

[3] President John F. Kennedy expressed this idea best, in his remarks at the dedication of the Aerospace Medical Health Center, the day before he was assassinated.

> We have a long way to go. Many weeks and months and years of long, tedious work lie ahead. There will be setbacks and frustrations and disappointments. There will be, as there always are,...temptations to do something else that is perhaps easier. But this research here must go on. This space effort must go on. ... That much we can say with confidence and conviction.
>
> Frank O'Connor, the Irish writer, tells in one of his books how, as a boy, he and his friends would make their way across the countryside, and when they came to an orchard wall that seemed too high and too doubtful to try and too difficult to permit their voyage to continue, they took off their hats and tossed them over the wall—and then they had no choice but to follow them.
>
> This Nation has tossed its cap over the wall of space, and we have no choice but to follow it. Whatever the difficulties, they will be overcome. Whatever the hazards, they must be guarded against. With the...help and support of all Americans, we will climb this wall with safety and with speed—and we shall then explore the wonders on the other side.

Remarks at the Dedication of the Aerospace Medical Health Center
President John F. Kennedy
San Antonio, Texas
November 21, 1963

not understood—signaling an error, printing a message, and changing a variable's value in memory seemed like very different operations. Moggi showed how these apparently unrelated effects could be encapsulated using monads, providing a common framework for a wide variety of effects. Could it be that the various types of divergence described in Chapter 5 are also related, in a deep and fundamental way? Indeed, divergence itself is an effect. From the monadic viewpoint, divergence is equivalent to an error, while from the relational programming viewpoint, divergence is equivalent to failure; is there a deeper connection?

Appendix A

Familiar Helpers

The auxiliaries below are used in the implementation of αKanren in Chapter 11.

```
(define get
  (λ (x s)
    (cond
      ((assq x s) ⇒ cdr)
      (else x))))

(define assp
  (λ (p s)
    (cond
      ((null? s) #f)
      ((p (car (car s))) (car s))
      (else (assp p (cdr s))))))

(define filter
  (λ (p s)
    (cond
      ((null? s) ())
      ((p (car s)) (cons (car s) (filter p (cdr s))))
      (else (filter p (cdr s))))))

(define remove-duplicates
  (λ (s)
    (cond
      ((null? s) ())
      ((memq (car s) (cdr s)) (remove-duplicates (cdr s)))
      (else (cons (car s) (remove-duplicates (cdr s)))))))
```

Appendix B

pmatch

In this appendix we describe **pmatch**, a simple pattern matcher written by Oleg Kiselyov. Let us first consider a simple example of **pmatch**.

```
(define h
  (λ (x y)
    (pmatch (x . y)
      ((a . b) (guard (number? a) (number? b)) (+ a b))
      ((_ . c) (guard (number? c)) (* c c))
      (else (* x x)))))
(list (h 1 2) (h w 5) (h 6 w)) ⇒ (3 25 36)
```

In this example, a dotted pair is matched against three different kinds of patterns.

In the first pattern, the value of *x* is lexically bound to *a* and the value of *y* is lexically bound to *b*. Before the pattern match succeeds, however, an optional guard is run within the scope of *a* and *b*. The guard succeeds only if *x* and *y* are numbers; if so, then the sum of *x* and *y* is returned.

The second pattern matches against a pair, provided that the optional guard succeeds. If so, the value of *y* is lexically bound to *c*, and the square of *y* is returned.

If the second pattern fails to match against **(***x* **.** *y***)**, because *y* is not a number, then the third and final clause is tried. An **else** pattern matches against *any* value, and never includes a guard. In this case, the clause returns the square of *x*, which must be a number in order to avoid an error at runtime.

Below is the definition of **pmatch**, which is implemented using continuation-passing-style macros (Hilsdale and Friedman 2000).

```
(define-syntax pmatch
  (syntax-rules (else guard)
    ((__ (op arg ...) cs ...)
     (let ((v (op arg ...)))
       (pmatch v cs ...)))
    ((__ v) (if #f #f))
    ((__ v (else e0 e ...)) (begin e0 e ...))
    ((__ v (pat (guard g ...) e0 e ...) cs ...)
     (let ((fk (λ () (pmatch v cs ...))))
       (ppat v pat
         (if (and g ...) (begin e0 e ...) (fk))
         (fk))))
    ((__ v (pat e0 e ...) cs ...)
     (let ((fk (λ () (pmatch v cs ...))))
       (ppat v pat (begin e0 e ...) (fk))))))
```

```
(define-syntax ppat
  (syntax-rules (__ quote unquote)
    ((__ v __ kt kf) kt)
    ((__ v () kt kf) (if (null? v) kt kf))
    ((__ v (quote lit) kt kf)
     (if (equal? v (quote lit)) kt kf))
    ((__ v (unquote var) kt kf) (let ((var v)) kt))
    ((__ v (x . y) kt kf)
     (if (pair? v)
       (let ((vx (car v)) (vy (cdr v)))
         (ppat vx x (ppat vy y kt kf) kf))
       kf))
    ((__ v lit kt kf) (if (equal? v (quote lit)) kt kf))))
```

The first clause ensures that the expression whose value is to be **pmatch**ed against is evaluated only once. The second clause returns an unspecified value if no other clause matches.

The remaining clauses represent the three types of patterns supported by **pmatch**. The first is the trivial **else** clause, which matches against any datum, and which behaves identically to an **else** clause in a **cond** expression. The other two clauses are identical, except that the first one includes a guard containing one or more expressions—if the datum matches against the pattern, the guard expressions are evaluated in left-to-right order. If a guard expression evaluates to #f, then it is as if the datum had failed to match against the pattern: the remaining guard expressions are ignored, and the next clause is tried. The expression (*fk*) is evaluated if the pattern it is associated with fails to match, or if the pattern matches but the guard fails.

ppat does the actual pattern matching over constants and pairs. The wildcard pattern **__** matches against *any* value[1]; the second pattern matches against

[1] The **pmatch** presented in (Byrd and Friedman 2007) uses a single underscore (_) as the wildcard pattern. Here we use a double underscore (__) for compatibility with R^6RS.

the empty list; the third pattern matches against a quoted value; and the fourth pattern matches against *any* value, and binds that value to a lexical variable with the specified identifier name. The fifth pattern matches against a pair, tears it apart, and recursively matches the *car* of the value against the *car* of the pattern. If that succeeds, the *cdr* of the value is recursively matched against the *cdr* of the pattern. (We use **let** to avoid building long *car/cdr* chains.) The last pattern matches against constants, including symbols.

Here is the definition of *h* after expansion.

```
(define h
  (λ (x y)
    (let ((v (x . y)))
      (let ((fk (λ ()
                (let ((fk (λ () (* x x))))
                  (if (pair? v)
                    (let ((vx (car v)) (vy (cdr v)))
                      (let ((c vy))
                        (if (number? c) (* c c) (fk))))
                    (fk))))))
        (if (pair? v)
          (let ((vx (car v)) (vy (cdr v)))
            (let ((a vx))
              (let ((b vy))
                (if (and (number? a) (number? b))
                  (+ a b)
                  (fk)))))
          (fk))))))
```

There are four kinds of improvements that should be resolved by the compiler. First, *vx* is not used in the top definition of *fk*, so it should not get a binding. Second, the binding to *a* and *b* should be parallel **let** bindings. Third, where *c* is bound, could have been where *vy* is bound, and where *a* and *b* are bound, could have been where *vx* and *vy* are bound, respectively. Fourth, thunk creation is unnecessary where no guard is present.

The **mv-let** macro used in Chapter 11 can be defined using **pmatch**.

```
(define-syntax mv-let
  (syntax-rules ()
    ((_ ((x ...) e) b0 b ...) (pmatch e ((x ...) b0 b ...)))))
(mv-let ((x y z) (list 1 2 3)) (+ x y z)) ⇒ 6
```

Appendix C

match^e and λ^e

In this appendix we describe **match**^{e} and λ^e, pattern-matching macros for writing concise miniKanren programs. These macros were designed by Will Byrd and implemented by Ramana Kumar with the help of Dan Friedman.

To illustrate the use of **match**^{e} and λ^e we will rewrite the explicit definition of *append*^{o}, which uses the core miniKanren operators $\equiv$, **cond**^{e}, and **exist**.

```
(define append^o
  (λ (l s out)
    (cond^e
      ((≡ () l) (≡ s out))
      ((exist (a d res)
         (≡ (a . d) l)
         (≡ (a . res) out)
         (append^o d s res))))))
```

We can shorten the *append*^{o} definition using **match**^{e}. **match**^{e} resembles **pmatch** (Appendix B) syntactically, but uses unification rather than uni-directional pattern matching. **match**^{e} expands into a **cond**^{e}; each **match**^{e} clause becomes a **cond**^{e} clause[1]. As with **pmatch** the first expression in each clause is an implicitly quasiquoted pattern. Unquoted identifiers in a pattern are introduced as unassociated logic variables whose scope is limited to the pattern and goals in that clause.

Here is *append*^{o} defined with **match**^{e}.

```
(define append^o
  (λ (l s out)
    (match^e (l s out)
      ((() s s))
      (((a . d) s (a . res)) (append^o d s res)))))
```

The pattern in the first clause attempts to unify the first argument of *append*^{o} with the empty list, while also unifying *append*^{o}'s second and third arguments. The

[1] The **match**^{a} and **match**^{u} forms are identical to **match**^{e}, except they expand into uses of **cond**^{a} and **cond**^{u}, respectively.

same unquoted identifier can appear more than once in a **match**^{e} pattern; this is not allowed in **pmatch**.

We can make *append*^{o} even shorter by using λ^e. λ^e just expands into a λ wrapped around a **match**^{e}—the **match**^{e} matches against the λ's argument list[2].

```
(define append^o
  (λ^e (l s out)
    ((() s s))
    (((a . d) s (a . res)) (append^o d s res))))
```

The double-underscore symbol __ represents a pattern wildcard that matches any value without binding it to a variable. For example, the pattern in *pair*^{o}

```
(define pair^o
  (λ^e (x)
    (((__ . __)))))
```

matches any pair, regardless of the values of its car and cdr.

λ^e and **match**^{e} also support nominal logic (see Chapter 9). Just as unquoted identifiers in a pattern are introduced as unassociated logic variables, using unquote splicing in a pattern introduces a fresh nom whose scope is limited to the pattern and goals in that clause. For example, the goal constructor

```
(define foo
  (λ (t)
    (fresh (a b)
      (exist (x y)
        (cond^e
          ((≡ (⋈ a (⋈ b (x b))) t))
          ((≡ (⋈ a (⋈ b (y b))) t))
          ((≡ (⋈ a (⋈ b (b y))) t))
          ((≡ (⋈ a (⋈ b (b y))) t)))))))
```

can be re-written as

```
(define foo
  (λ^e (t)
    ((tie @a (tie @b (x @b))))
    ((tie @a (tie @b (y @b))))
    ((tie @a (tie @b (@b y))))
    ((tie @a (tie @b (@b y))))))
```

where tie is the tag returned by the ⋈ constructor[3].

Here is the definition of λ^e, and its impure variants λ^a and λ^u.

```
(define-syntax λ^e
  (syntax-rules ()
```

[2] The λ^a and λ^u forms are identical to λ^e, except they expand into uses of **match**^{a} and **match**^{u}, respectively.

[3] Unfortunately, this explicit pattern matching breaks the abstraction of the ⋈ constructor.

```
    ((_ (x ...) c c^* ...)
     (λ (x ...) (match^e (quasiquote (unquote x) ...) (c c^* ...) ()))))))

(define-syntax λ^a
  (syntax-rules ()
    ((_ (x ...) c c^* ...)
     (λ (x ...) (match^a (quasiquote (unquote x) ...) (c c^* ...) ()))))))

(define-syntax λ^u
  (syntax-rules ()
    ((_ (x ...) c c^* ...)
     (λ (x ...) (match^u (quasiquote (unquote x) ...) (c c^* ...) ()))))))
```

Here is the definition of **match**^{e}, and its impure variants **match**^{a} and **match**^{u}.

```
(define-syntax exist^*
  (syntax-rules ()
    ((_ (x ...) g_0 g ...)
     (λ_G (a)
       (inc
         (let^* ((x (var x)) ...)
           (bind^* (g_0 a) g ...)))))))

(define-syntax fresh^*
  (syntax-rules ()
    ((_ (x ...) g_0 g ...)
     (λ_G (a)
       (inc
         (let^* ((x (nom x)) ...)
           (bind^* (g_0 a) g ...)))))))

(define-syntax match^e
  (syntax-rules ()
    ((_ (f x ...) g^* . cs)
     (let ((v (f x ...))) (match^e v g^* . cs)))
    ((_ v g^* . cs) (mpat cond^e v (g^* . cs) ()))))

(define-syntax match^a
  (syntax-rules ()
    ((_ (f x ...) g^* . cs)
     (let ((v (f x ...))) (match^a v g^* . cs)))
    ((_ v g^* . cs) (mpat cond^a v (g^* . cs) ()))))
```

```
(define-syntax match^u
  (syntax-rules ()
    ((_ (f x ...) g* . cs)
     (let ((v (f x ...))) (match^u v g* . cs)))
    ((_ v g* . cs) (mpat cond^u v (g* . cs) ()))))

(define-syntax mpat
  (syntax-rules (_ quote unquote unquote-splicing expand cons)
    ((_ co v () (l ...)) (co l ...))
    ((_ co v (pat) xs as ((g ...) . cs) (l ...))
     (mpat co v cs (l ... ((fresh* as (exist* xs (≡ pat v) g ...))))))
    ((_ co v ((_ g0 g ...) . cs) (l ...))
     (mpat co v cs (l ... ((exist () g0 g ...)))))
    ((_ co v (((unquote y) g0 g ...) . cs) (l ...))
     (mpat co v cs (l ... ((exist (y) (≡ y v) g0 g ...)))))
    ((_ co v (((unquote-splicing b) g0 g ...) . cs) (l ...))
     (mpat co v cs (l ... ((fresh (b) g0 g ...)))))
    ((_ co v ((pat g ...) . cs) ls)
     (mpat co v (pat expand) () () ((g ...) . cs) ls))
    ((_ co v (_ expand . k) (x ...) as cs ls)
     (mpat co v ((unquote y) . k) (y x ...) as cs ls))
    ((_ co v ((unquote y) expand . k) (x ...) as cs ls)
     (mpat co v ((unquote y) . k) (y x ...) as cs ls))
    ((_ co v ((unquote-splicing b) expand . k) xs (a ...) cs ls)
     (mpat co v ((unquote b) . k) xs (b a ...) cs ls))
    ((_ co v ((quote c) expand . k) xs as cs ls)
     (mpat co v (c . k) xs as cs ls))
    ((_ co v ((a . d) expand . k) xs as cs ls)
     (mpat co v (d expand a expand cons . k) xs as cs ls))
    ((_ co v (d a expand cons . k) xs as cs ls)
     (mpat co v (a expand d cons . k) xs as cs ls))
    ((_ co v (a d cons . k) xs as cs ls)
     (mpat co v ((a . d) . k) xs as cs ls))
    ((_ co v (c expand . k) xs as cs ls)
     (mpat co v (c . k) xs as cs ls))))
```

Appendix D

Nestable Engines

Our implementation of ferns in Chapter 15 requires nestable engines (Dybvig and Hieb 1989; Hieb et al. 1994), which we present here with minimal comment. The implementation uses a global variable, *state*, which holds two values: the number of ticks available to the currently running engine or #f representing infinity; and a continuation. *make-engine* makes an engine out of a thunk. **engine** is a macro that makes an engine from an expression. λ_t is like λ except that it passes its body as a thunk to *expend-tick-to-call*, which ensures a tick is spent before the body is evaluated and passes the suspended body to the continuation if no ticks are available. Programs that use this embedding of nestable engines (and by extension our embedding of **cons**$_\perp$) should not use *call/cc*, because the uses of *call/cc* in the nestable engines implementation may interact with other uses in ways that are difficult for the programmer to predict.

```
(define-syntax engine
  (syntax-rules ()
    ((_ e) (make-engine (λ () e)))))

(define-syntax λt
  (syntax-rules ()
    ((_ formals b0 b ...) (λ formals (expend-tick-to-call (λ () b0 b ...))))))

(define state (cons #f 0))

(define expend-tick-to-call
  (λ (thunk)
    ((call/cc
       (λ (k)
         (let th  ()
           (cond
             ((not (car state)) (k thunk))
             ((zero? (car state)) ((cdr state) th))
             (else (set-car! state (− (car state) 1)) (k thunk)))))))))

(define make-engine
(λ (thunk)
 (λ (ticks)
  (let* ((gift (if (car state) (min (car state) ticks) ticks))
         (saved-state (cons (and (car state) (− (car state) gift)) (cdr state)))
         (caught (call/cc
                   (λ (k)
                    (replace! state gift k)
                    (let ((result (thunk)))
                     ((cdr state) (cons (car state) result)))))))
   (replace! state (car saved-state) (cdr saved-state))
   (let ((owed (− ticks gift)))
    (cond
     ((pair? caught)
      (and (car state) (set-car! state (+ (car state) (car caught))))
      (cons (+ (car caught) owed) (cdr caught)))
     (else (let ((e (make-engine caught)))
             (if (zero? owed) (cons #f e)
              (let ((th (λ () (e owed))))
               ((call/cc (λ (k̂) ((cdr state) (λ () (k̂ th))))))))))))))
```

Appendix E

Parser for Nominal Type Inferencer

This parser is used by the nominal type inferencer is section 9.3.

```
(define parse (λ (exp) (parse-aux exp ())))

(define parse-aux
  (λ (exp env)
    (pmatch exp
      (x (guard (symbol? x))
       (let ((v (cdr (assq x env))))
         (var v)))
      (n (guard (number? n)) (intc n))
      (b (guard (boolean? b)) (boolc b))
      ((zero? e) (let ((e (parse-aux e env))) (zero? e)))
      ((sub1 e) (let ((e (parse-aux e env))) (sub1 e)))
      ((fix e) (let ((e (parse-aux e env))) (fix e)))
      ((* e1 e2) (let ((e1 (parse-aux e1 env)) (e2 (parse-aux e2 env))) (* e1 e2)))
      ((if e1 e2 e3)
       (let ((e1 (parse-aux e1 env)) (e2 (parse-aux e2 env)) (e3 (parse-aux e3 env)))
         (if e1 e2 e3)))
      ((λ (x) e)
       (let* ((a (nom x)) (e (⋈ a (parse-aux e (cons (cons x a) env)))))
         (lam e)))
      ((e1 e2)
       (let ((e1 (parse-aux e1 env)) (e2 (parse-aux e2 env)))
         (app e1 e2))))))
```

Bibliography

H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, second edition, 1996.

M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Ann. of Math.*, 2:781–793, 2002.

H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

S. Antoy, M. Hanus, B. Massey, and F. Steiner. An implementation of narrowing strategies. In *PPDP '01: Proc. of the 3rd Intl. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, Florence, Italy, Sep. 5–7, 2001*, pages 207–217. ACM, 2001.

K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *ILPS '93: Proc. of the 1993 Intl. Symp. on Logic Programming, Vancouver, Oct. 26–29, 1993*, pages 12–35, Cambridge, MA, 1993. MIT Press.

K. R. Apt and E. Marchiori. Reasoning about Prolog programs: From modes through types to assertions. *Formal Aspects of Comp.*, 6(6A):743–764, 1994.

K. R. Apt and A. Pellegrini. Why the occur-check is not a problem. In M. Bruynooghe and M. Wirsing, editors, *PLILP '92: Proc. of the 4th Intl. Symp. on Programming Language Implementation and Logic Programming, Leuven, Belgium, Aug. 26–28, 1992*, pages 69–86, London, 1992. Springer.

J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, December 2003.

F. Baader and W. Snyder. Unification theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, Amsterdam and Cambridge, MA, 2001.

H. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.

B. Beckert and J. Posegga. The leanTAP-FAQ: Frequently asked questions about lean*TAP*. `http://www.uni-koblenz.de/~beckert/pub/LeanTAP_FAQ.pdf`.

B. Beckert and J. Posegga. leanTAP: Lean tableau-based deduction. *J. Auto. Reason.*, 15(3):339–358, 1995.

R. Becket and Z. Somogyi. DCGs + memoing = packrat parsing but is it worth it? In P. Hudak and D. S. Warren, editors, *PADL '08: Practical Aspects of Declarative Languages, 10th Intl. Symp., San Francisco, January 7–8, 2008*, volume 4902 of *LNCS*, pages 182–196. Springer, 2008.

P. E. Bonzon. A metacircular evaluator for a logical extension of Scheme. *Lisp Symb. Comput.*, 3(2):113–134, 1990.

R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design: Proc. of the IFIP TC10/WG 10.2 Intl. Conf., Nijmegen, The Netherlands, Jun. 22–24, 1992*, pages 129–156. North-Holland, 1992.

B. Braßel, S. Fischer, and F. Huch. Declaring numbers. In R. Echahed, editor, *WFLP '07: Proc. of 16th Intl. Workshop on Functional and (Constraint) Logic Programming, Paris, June 25, 2007*, pages 23–36, 2007.

W. E. Byrd and D. P. Friedman. αKanren: A fresh name in nominal logic programming. In D. Dubé, editor, *Proc. of the 2007 Workshop on Scheme and Functional Programming, Freiburg, Germany, Sep. 30, 2007*, Université Laval Technical Report DIUL-RT-0701, pages 79–90 (*see also* `http://www.cs.indiana.edu/~webyrd` *for improvements*), 2007.

C. Calvès and M. Fernández. Implementing nominal unification. *Electr. Notes Theor. Comput. Sci.*, 176(1):25–37, 2007.

C. Calvès and M. Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008.

R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proc. of the ACM SIGPLAN 1991 Conf. on Programming Language Design and Implementation, Toronto, Jun. 26–28, 1991*, pages 278–292, New York, 1991. ACM.

G. Cattaneo and V. Loia. A Common-LISP implementation of an extended Prolog system. *SIGPLAN Notices*, 23(4):87–102, 1988.

W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

J. Cheney. *Nominal Logic Programming.* PhD thesis, Cornell University, Aug. 2004a.

J. Cheney. The complexity of equivariant unification. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *ICALP '04: Proc. of the 31st Intl. Colloq. on Automata, Languages and Programming, Turku, Finland, Jul. 12–16, 2004*, volume 3142 of *LNCS*, pages 332–344. Springer, 2004b.

J. Cheney. Equivariant unification. In J. Giesl, editor, *RTA '05: Proc. of the 16th Intl. Conf. on Rewriting Techniques and Applications, Nara, Japan, Apr. 19–21, 2005*, volume 3467 of *LNCS*, pages 74–89. Springer, 2005.

J. Cheney and C. Urban. αProlog: A logic programming language with names, binding and α-equivalence. In B. Demoen and V. Lifschitz, editors, *ICLP '04: Proc. of the 20th Intl. Conf. on Logic Programming, Saint-Malo, France, Sep. 6–10, 2004*, volume 3132 of *LNCS*, pages 269–283, Saint-Malo, France, Sept. 6–10, 2004. Springer.

J. Cheney and C. Urban. Nominal logic programming. *ACM Trans. Program. Lang. and Syst.*, 30(5):1–47, 2008.

H. Christiansen. Automated reasoning with a constraint-based metainterpreter. *J. Log. Program.*, 37(1-3):213–254, 1998.

K. Claessen and P. Ljunglöf. Typed logical variables in Haskell. In *Proc. of the Haskell Workshop*. ACM SIGPLAN, 2000.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *RTA '03: Proc. of the 14th Intl. Conf. on Rewriting Techniques and Applications, Valencia, Spain, Jun. 9–11, 2003*, number 2706 in LNCS, pages 76–87. Springer, June 2003.

W. F. Clocksin. *Clause and Effect: Prolog Programming for the Working Programmer*. Springer, Secaucus, NJ, 1997.

W. F. Clocksin and C. S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer, 2003.

A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, London, 1982.

A. Colmerauer. Equations and inequations on finite and infinite trees. In *FGCS '84: Proc. of the Intl. Conf. on Fifth Generation Computer Systems, Tokyo, Nov. 6–9, 1984*, pages 85–99, Tokyo, 1984.

A. Colmerauer. Prolog in 10 figures. *Commun. ACM*, 28(12):1296–1310, 1985.

A. Colmerauer. An introduction to Prolog III. *Commun. ACM*, 33(7):69–90, 1990.

A. Colmerauer and P. Roussel. The birth of Prolog. In *History of programming languages—II*, pages 331–367. ACM, New York, 1996.

H. Comon. Disunification: a survey. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 322–359, Cambridge, MA, 1991. MIT Press.

H. Comon and P. Lescanne. Equational problems and disunification. *J. Symb. Comput.*, 7(3-4):371–425, 1989.

S. K. Debray, N.-W. Lin, and M. V. Hermenegildo. Task granularity analysis in logic programs. In *PLDI '90: Proc. of the ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation, White Plains, New York, Jun. 20–22, 1990*, pages 174–188, New York, 1990. ACM.

E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, 1997.

K. Doets. *From Logic to Logic Programming.* MIT Press, Cambridge, MA, 1994.

G. Dowek, M. Gabbay, and D. Mulligan. Permissive nominal terms and their unification. In M. Gavanelli and F. Riguzzi, editors, *CILC '09: 24-esimo Convegno Italiano di Logica Computazionale, Ferrara, Italy, Jun. 2009*, Ferrara, Italy, June 2009.

R. K. Dybvig and R. Hieb. Engines from continuations. *Comput. Lang*, 14(2): 109–123, 1989.

M. Felleisen. Transliterating Prolog into Scheme. Technical report, Indiana University Computer Science Department, Oct. 1985. Indiana University Computer Science Department Technical report No. 182.

A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In E. Lusk and R. Overbeek, editors, *CADE '88: Proc. of the 9th Intl. Conf. on Automated Deduction, Argonne, IL, May 23–26, 1988*, pages 61–80. Springer, 1988.

M. Fernández and M. J. Gabbay. Nominal rewriting. *Inf. Comput.*, 205(6):917–965, 2007.

R. E. Filman and D. P. Friedman. *Coordinated Computing: Tools and Techniques for Distributed Software.* McGraw-Hill, 1984.

M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1996.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI '93: Proc. of the ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, Albuquerque, Jun. 23–25, 1993*, pages 237–247, New York, 1993. ACM.

B. Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *ICFP '02: Proc. of the Seventh ACM SIGPLAN Intl. Conf. on Functional Programming, Pittsburgh, Oct. 4–6, 2002*, pages 36–47, New York, 2002. ACM.

E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

D. P. Friedman and O. Kiselyov. A declarative applicative logic programming system. `http://kanren.sourceforge.net`, 2005.

D. P. Friedman and M. Wand. *Essentials of Programming Languages.* MIT Press, third edition, 2008.

D. P. Friedman and D. S. Wise. Sting-unless: a conditional, interlock-free store instruction. In *16th Annual Allerton Conf. on Communication, Control, and Computing, University of Illinois (Urbana-Champaign)*, pages 578–584. University of Illinois, Urbana-Champaign, 1978.

D. P. Friedman and D. S. Wise. An approach to fair applicative multiprogramming. In G. Kahn, editor, *Proc. of the Intl. Symp. on Semantics of Concurrent Computation, Evian, France, Jul. 2–4, 1979*, volume 70 of *LNCS*, pages 203–225. Springer, July 1979.

D. P. Friedman and D. S. Wise. An indeterminate constructor for applicative programming. In *POPL '80: Conf. Record of the 7th Annual ACM Symp. on Principles of Programming Languages, Las Vegas, Jan. 1980*, pages 245–250, New York, Jan. 1980. ACM Press.

D. P. Friedman and D. S. Wise. Fancy ferns require little care. In S. Holmström, B. Nordström, and Å. Wikström, editors, *Symp. on Functional Languages and Computer Architecture, Göteborg, Sweden, 1981*, pages 124–156, Göteborg, Sweden, June 1981. Laboratory for Programming Methodology, University of Göteborg and Chalmers University of Technology.

D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer.* MIT Press, Cambridge, MA, 2005.

D. C. Gras and M. V. Hermenegildo. The Ciao module system: A new module system for Prolog. *Electr. Notes Theor. Comput. Sci.*, 30(3), 1999.

H.-F. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In P. Codognet, editor, *ICLP '01: Proc. of the 17th Intl. Conf. on Logic Programming, Paphos, Cyprus, Nov. 26–Dec. 1, 2001*, volume 2237 of *LNCS*, pages 181–196, London, 2001. Springer.

H.-F. Guo and G. Gupta. Dynamic reordering of alternatives for definite logic programs. *Comput. Lang. Syst. Struct.*, 35(3):252–265, 2009.

G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM Trans. on Program. Lang. and Syst.*, 15(4):659–680, September 1993.

G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In V. Dahl and I. Niemelä, editors, *ICLP '07: Proc. of the 23rd Intl. Conf. on Logic Programming, Porto, Portugal, Sep. 8–13, 2007*, volume 4670 of *LNCS*, pages 27–44. Springer, 2007.

T. Hallgren. Fun with functional dependencies. `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html`, 2001.

M. Hanus. Analysis of Residuating Logic Programs. *J. Log. Program.*, 24(3):219–245, Sept. 1995.

M. Hanus. *Report on Curry (ver.0.8.2).* Inst. für Informatik, Christian-Albrechts-Universität, Germany, 2006.

M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *ILPS '95: Proc. Workshop on Visions for the Future of Logic Programming, Portland*, pages 95–107, 1995.

C. T. Haynes. Logic continuations. *J. Log. Program.*, 4(2):157–176, 1987.

C. T. Haynes and D. P. Friedman. Abstracting timed preemption with engines. *J. Comp. Lang.*, 12(2):109–121, 1987.

F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, University of Melbourne, 1996.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, third edition, 2002.

D. Herman and M. Wand. A theory of hygienic macros. In *ESOP '08: Proc. of the 17th European Symp. on Programming, Budapest, Mar. 29–Apr. 6, 2008*, volume 4960 of *LNCS*. Springer, Mar. 2008.

M. V. Hermenegildo and F. Rossi. Strict and non-strict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *J. Log. Program.*, 22(1):1–45, 1995.

R. Hieb, K. Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp and Symb. Comp.*, 7(1):83–110, 1994.

E. Hilsdale and D. P. Friedman. Writing macros in continuation-passing style. In *Scheme and Functional Programming 2000, Montréal, Sep. 17, 2000*, Sept. 5, 2000.

J. Hughes. Generalising monads to arrows. *Science of Comp. Program.*, 37:67–111, 1998.

Intl. Organization for Standardization. *ISO/IEC 13211-1:1995: Information Technology — Programming Languages — Prolog — Part 1: General Core.* 1995.

Intl. Organization for Standardization. *ISO/IEC 13211-2:2000: Information Technology – Programming Languages – Prolog – Part 2: Modules.* 2000.

J. Jaffar and M. J. Maher. Constraint logic programming: A Survey. *J. Log. Program.*, 19/20:503–581, 1994.

E. R. Jeschke. *An Architecture for Parallel Symbolic Processing Based on Suspending Construction.* PhD thesis, Indiana University Computer Science Department, May 1995. Technical Report No. 445, 152 pages.

S. D. Johnson. An interpretive model for a language based on suspended construction. Master's thesis, Indiana University Computer Science Department, 1977. Indiana University Computer Science Department Technical Report No. 68.

S. D. Johnson. Circuits and systems: Implementing communication with streams. *IMACS Trans. on Sci. Comp., Vol. II*, pages 311–319, 1983.

K. M. Kahn and M. Carlsson. How to implement Prolog on a LISP machine. In *Implementations of Prolog*, Ellis Horwood Series in Artificial Intelligence, pages 117–134. Ellis Horwood/Halsted Press/Wiley, 1984.

O. Kiselyov. Number-parameterized types. *The Monad.Reader*, 5, 2005. URL `http://www.haskell.org/tmrwiki/NumberParamTypes`.

O. Kiselyov and C. chieh Shan. Lightweight static resources: Sexy types for embedded and systems programming. In M. T. Morazán, editor, *Draft Proc. of the 8th Symp. on Trends in Functional Programming, , New York City, 2–4 Apr. 2007.* Seton Hall University, 2007. URL `http://cs.shu.edu/tfp2007/draftProcDocument.pdf`. TR-SHU-CS-2007-04-1.

O. Kiselyov, C. chieh Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers. In O. Danvy and B. C. Pierce, editors, *ICFP '05: Proc. of the 10th ACM SIGPLAN Intl. Conf. on Functional Programming, Tallinn, Estonia, Sep. 26–28, 2005*, pages 192–203, Sept. 2005.

O. Kiselyov, W. E. Byrd, D. P. Friedman, and C. chieh Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). In J. Garrigue and M. V. Hermenegildo, editors, *FLOPS '08: Proc. of the 9th Intl. Symp. on Functional and Logic Programming, Ise, Japan, Apr. 14–16, 2008*, volume 4989 of *LNCS*, pages 64–80. Springer, 2008.

S. C. Kleene. *Introduction to Metamathematics.* Bibl. Mathematica. North-Holland, Amsterdam, 1952.

H. J. Komorowski. QLOG interactive environment—the experience from embedding a generalized Prolog in Interlisp. Technical report, Linkoping University, 1979. LiTH-MAT-R-79-19.

R. A. Kowalski. *Logic for Problem Solving.* Prentice Hall PTR, Upper Saddle River, NJ, 1979.

M. R. Lakin and A. M. Pitts. A metalanguage for structural operational semantics. In M. T. Morazán, editor, *Trends in Functional Programming*, volume 8, pages 19–35. Intellect/The University of Chicago Press, 2008.

A. Lisitsa. λleanTAP: lean deduction in λProlog. Technical report, ULCS-03-017, University of Liverpool, Department of Computer Science, 2003.

J. W. Lloyd. *Foundations of Logic Programming.* Springer, New York, second extended edition, 1987.

P. Lopez, M. V. Hermenegildo, and S. K. Debray. A methodology for granularity-based control of parallelism in logic programs. *J. Symb. Comp.*, 21(4):715–734, 1996.

R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In E. Lusk and R. Overbeek, editors, *CADE '88: Proc. of the 9th Intl. Conf. on Automated Deduction, Argonne, IL, May 23–26, 1988*, pages 415–434, Argonne, IL, 1988. Springer.

Y. V. Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, Cambridge, MA, 1993.

J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In V. van Oostrom, editor, *RTA '04: Proc. of the 15th Intl. Conf. on Rewriting Techniques and Applications, Aachen, Germany, Jun. 3–5, 2004*, volume 3091 of *LNCS*, pages 301–311. Springer, 2004.

D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

G. Nadathur. The metalanguage λProlog and its implementation. In H. Kuchen and K. Ueda, editors, *FLOPS '01: Proc. of the 5th Intl. Symp. on Functional and Logic Programming, Waseda University, Tokyo, Mar. 7–9, 2001*, volume 2024 of *LNCS*, pages 1–20, London, 2001. Springer.

G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Proc. of the Fifth Intl. Conf. and Symp. on Logic Programming, Seattle, Aug. 1988*, pages 810–827, Aug. 1988.

L. Naish. Pruning in logic programming. Technical Report 95/16, Department of Computer Science, University of Melbourne, Melbourne, June 1995.

H. R. Nayak. *Concurrent LogLISP*. PhD thesis, Syracuse University, Syracuse, 1989.

J. P. Near, W. E. Byrd, and D. P. Friedman. αlean*TAP*: A declarative theorem prover for first-order classical logic. In M. G. de la Banda and E. Pontelli, editors, *ICLP '08: Proc. of the 24th Intl. Conf. on Logic Programming, Udine, Italy, Dec. 9–13, 2008*, volume 5366 of *LNCS*, pages 238–252. Springer, 2008. ISBN 978-3-540-89981-5.

C. Okasaki. Purely functional random-access lists. In *FPCA '95: Proc. of the 7th Intl. Conf. on Functional Programming Languages and Computer Architecture, La Jolla, CA, Jun. 25–28, 1995*, pages 86–95, New York, 1995. ACM.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

L. C. Paulson. A generic tableau prover and its integration with Isabelle. *J. of Univ. Comp. Sci.*, 5(3):73–87, 1999.

F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *J. Auto. Reason.*, 2(2):191–216, 1986.

F. Pereira and D. Warren. Definite clause grammars for language analysis. In *Readings in Natural Language Processing*, pages 101–124. Morgan Kaufmann, San Francisco, 1986.

F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88: Proc. of the ACM SIGPLAN 1988 Conf. on Programming Language Design and Implementation, Atlanta, Jun. 22–24, 1988*, pages 199–208, New York, 1988. ACM.

F. Pfenning and C. Schurmann. System description: Twelf—a meta-logical framework for deductive systems. In H. Ganzinger, editor, *CADE '99: Proc. of the 16th Intl. Conf. on Automated Deduction, Trento, Italy, Jul. 7–10, 1999*, pages 202–206, Trento, Italy, 1999. Springer.

B. C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, 2002.

A. M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

G. D. Plotkin. A structural approach to operational semantics. *J. Logic and Algebraic Program.*, 60–61:17–139, 2004.

F. Pottier. *Cαml Reference Manual.* INRIA, 2006-12-14 edition, Dec. 2006.

J. A. Robinson and E. E. Sibert. LOGLISP: An alternative to Prolog. In J. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 399–419. Ellis Horwood Ltd., Chicester, England, 1982.

E. Ruf and D. Weise. LogScheme: Integrating logic programming into Scheme. *Lisp Symb. Comput.*, 3(3):245–288, 1990.

K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Program. Lang. and Syst.*, 20(3):586–634, 1998.

K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *SIGMOD '94: Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data, Minneapolis, May 24–27, 1994*, pages 442–453, New York, 1994. ACM.

T. Schrijvers, B. Demoen, and D. S. Warren. TCHR: a framework for tabled CLP. *Theory and Practice of Logic Program.*, 8(04):491–526, 2008a.

T. Schrijvers, V. Santos Costa, J. Wielemaker, and B. Demoen. Towards typed Prolog. In *ICLP '08: Proc. of the 24th Intl. Conf. on Logic Programming, Udine, Italy, Dec. 9–13, 2008*, pages 693–697, Berlin, 2008b. Springer.

S. Seres and M. Spivey. Functional reading of logic programs. *J. of Univ. Comp. Sci.*, 6(4):433–446, 2000.

M. R. Shinwell. Fresh O'Caml: Nominal abstract syntax for the masses. *Electr. Notes Theor. Comput. Sci.*, 148(2):53–77, 2006.

M. R. Shinwell, A. M. Pitts, and M. Gabbay. FreshML: programming with binders made simple. In C. Runciman and O. Shivers, editors, *ICFP '03: Proc. of the 8th ACM SIGPLAN Intl. Conf. on Functional Programming, Uppsala, Sweden, Aug. 25–29, 2003*, pages 263–274. ACM Press, 2003.

J. G. Siek and W. Taha. Gradual typing for functional languages. In R. Findler, editor, *Proc. of the 2006 Scheme and Functional Programming Workshop, Portland, Sep. 17, 2006*, University of Chicago Technical Report TR-2006-06, pages 81–92, 2006.

D. Sitaram. Programming in Schelog.
`http://www.ccs.neu.edu/home/dorai/schelog/schelog.html`, 1993.

Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an efficient purely declarative logic programming language. In *Proc. of the Australian Computer Science Conf., Glenelg, Australia*, pages 499–512, 1995.

M. Sperber, R. K. Dybvig, M. Flatt, and van Straaten, A. (eds.). Revised^6 report on the algorithmic language Scheme, Sept. 2007. URL `http://www.r6rs.org/`.

J. M. Spivey and S. Seres. Embedding Prolog in Haskell. In E. Meijer, editor, *Proc. of the 1999 Haskell Workshop*, Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, 1999.

J. M. Spivey and S. Seres. Combinators for logic programming. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 177–200. Palgrave, 2003.

G. L. Steele Jr. *COMMON LISP: The Language.* Digital Press, second edition, 1990.

M. Stickel. A Prolog technology theorem prover. In E. Lusk and R. Overbeek, editors, *CADE '88: Proc. of the 9th Intl. Conf. on Automated Deduction, Argonne, IL, May 23–26, 1988*, pages 752–753. Springer, 1988.

J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1979.

G. Sutcliffe and C. Suttner. The TPTP Problem Library. *J. Auto. Reason.*, 21(2): 135–277, 1998.

E. Todoran and N. S. Papaspyrou. Continuations for parallel logic programming. In *PPDP '00: Proc. of the 2nd ACM SIGPLAN Intl. Conf. on Principles and Practice of Declarative Programming, Montréal, Sep. 20–22, 2000*, pages 257–267, New York, 2000. ACM.

C. Urban and J. Cheney. Avoiding equivariance in Alpha-Prolog. In P. Urzyczyn, editor, *TLCA '05: 7th Intl. Conf. on Typed Lambda Calculi and Applications, Nara, Japan, Apr. 21–23, 2005*, volume 3461 of *LNCS*, pages 401–416. Springer, 2005.

C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.

P. Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *J. Log. Program.*, 19/20:385–441, 1994.

E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *RTA '01: Proc. of the 12th Intl. Conf. on Rewriting Techniques and Applications, Utrecht, The Netherlands, May 22–24, 2001*, volume 2051 of *LNCS*, pages 357–362, London, 2001. Springer.

P. Wadler. The essence of functional programming. In *POPL '92: Conf. Record of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Albuquerque, Jan. 1992*, pages 1–14, Albuquerque, Jan., 1992. ACM Press.

M. Wand and D. Vaillancourt. Relating models of backtracking. In *ICFP '04: Proc. of the 9th ACM SIGPLAN Intl. Conf. on Functional Programming, Snow Bird, UT, Sep. 19–22, 2004*, pages 54–65, New York, 2004. ACM.

D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, AI Center, SRI Intl., Menlo Park, CA, Oct. 1983.

D. S. Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, 1992.

J. Wielemaker. An overview of the SWI-Prolog programming environment. In F. Mesnard and A. Serebenik, editors, *WLPE '03: Proc. of the 13th Intl. Workshop on Logic Programming Environments, Tata Institute of Fundamental Research, Mumbai, India, Dec. 8, 2003*, pages 1–16, Heverlee, Belgium, Dec. 2003. Katholieke Universiteit Leuven.