CONSTRAINT MICROKANREN IN THE CLP SCHEME

Jason Hemann

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 School of Informatics, Computing, and Engineering
Indiana University
January 2020

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

_____

Daniel P. Friedman, Ph.D.

_____

Amr Sabry, Ph.D.

_____

Sam Tobin-Hochstadt, Ph.D.

_____

Lawrence Moss, Ph.D.

December 20, 2019

To Mom and Dad.

Acknowledgements

I want to thank all my housemates and friends from 1017 over the years for their care and support. I'm so glad to have you all, and to have you all over the world. Who would have thought that an old house in Bloomington could beget so many great memories. While I'm thinking of it, thanks to Lisa Kamen and Bryan Rental for helping to keep a roof over our head for so many years.

Me encantan mis salseros y salseras. I know what happens in the rueda stays in the rueda, so let me just say there's nothing better for taking a break from right-brain activity.

Thanks to Kosta Papanicolau for his early inspiration in math, critical thinking, and research, and to Profs. Mary Flagg and Michael Larsen for subsequent inspiration and training that helped prepare me for this work. Learning, eh?—who knew? I want to thank also my astounding undergraduate mentors including Profs. Mark Lewis, Berna Massingill, Paul Myers for their early and continued support, a gentle push when I needed it, and for their willingness to answer my questions. Likewise, I want to thank the people of my summer REU in Oakland University including Profs. Fatma Mili, Debasis Debnath, and Mohahmed Zody—and a special shout-out to T-313—for an educational, inspiring, and interesting summer.

Thank you Lynne Mikolon and Laura Reed. Without you I don't know what I would have done. But whatever it was, odds are it would have taken four times as long or that I would still be trying to figure out the coffee machine. Thanks also to Regina and Patty and Sherry and of the IU SICE staff who helped me jump through all manner of hoops when, left to my own devices, I would have instead garroted myself with red tape. Likewise thanks

to the Wells librarians. Their inter-library loan jiu-jitsu saved me countless hours and made the otherwise impossible possible. Thank you also to the IU Graduate School staff for their help in preparing this manuscript.

I'm grateful to the entire 311 and 304 staff: Zach, Robert, Cam, Tim, Suzanne, Brittany, Kyle, Ken, Andy, Andre, Ed, Kristyn, Erik, Josh, Carl, Russell, Ryan, Taylor, K, Mozzy, Mark, Anna, Lalo, Weixi, Jack, Coleman, Adam, David, Hao, Alyssa, and Lewis. Without you all I'd still be in the middle of grading. I could not have hoped for a better set of colleagues and friends. I need to thank all of my 311 and 521 students, for allowing me to guinea pig much of this material to/at them. Your comments, suggestions, corrections and improvements helped make this dissertation what it is. And I need to also thank Adam Foltzer, my predecessor and 311 AI, for his early help and getting me sucked into all of this.

Thank you Fiora Pizzo and Kaitlin Kertesz for keeping me on track, and thanks also to CITL and my writing posses and group leaders—especially Tara. I need to also thank Haley, Rin, Praveen, Kaitlin Guidarelli, Emily Larson, and all of my other accountabilibuddies and ersatz writing groups for sitting and working with me. To Ben Lerner, thanks for both sharing your space and time, and thanks for keeping me on track. I want to shout out Kyle and Vince for their support and for providing much needed R & R back home. We'll have to find a holiday to celebrate this—Dis-mas?

Many thanks also to my coauthors and collaborators, including Matt Might, Daniel Brady, Will Byrd, Oleg Kiselyov, and Erik Holk. I want to thank Chung-chieh Shan, and again thank Will Byrd and Oleg Kiselyov, for early discussions of constraints in miniKanren. I want to thank Ryan Culpepper, Leif Anderson, and Alex Knauth for their improvements to the framework macros, Josh Cox for his initial work on miniKanren constraints, and Andre Kuhlenschmidt and Michael Ballantyne for their semantics help specifically and general help generally. I also want to thank all of my anonymous reviewers for all their suggestions

and improvements. I need to thank Will Byrd, for paving my way practically, technically, academically, and socially. I would not have been to half as many conferences and talks without his initial prodding. I would not have known what this research could be. His larger-than-life presence and visions inspire a wider view of the current project and then some. I need to thank also the IU PL wonks for allowing me to dry-run so much of this dissertation by them. And I want to thank also the Northeastern University PL group and my Rose-Hulman posse for their advice and suggestions. Thanks to Dr. Spencer and his mantra for helping to coax me through the trickier bits, and thanks also to my good friend, caffeine.

Special thanks to Lindsey Kuiper, for being so in my corner when I've needed it. Lindsey, you've been more supportive and inspirational than I think you know. Thank you Memo Dalkilic, for lending an outside ear and a comforting shoulder. Thanks also to Charles Pope, for his continued assistance in helping me staff courses, Funda Ergun for her well-timed prodding, and both for being an ear when I needed one. Matthias Felleisen, thank you so, so much for not just earlier foundational work, but also for the mentoring, older-brothering, inspirational talks, practical advice, proofreading, the best whip-cracking this side of Gen. Patton, and so much more. I'm so lucky to be under your wing.

I of course want to thank my wonderful, loving family. Momma, thanks for all the fruit and coffee and cookies that were the fuel that made this possible. Dad, thanks for the frequent talks and even more frequent flier miles. Thank you Tyler and Jana, your wonderful spouses Aly and Ross, and little Ady, Andy, and Brett. And Larry for such wonderful talks. And Jill, not least of all for the never-ending snacks. Having more time to spend with everyone in the whole, extended family is another great reason to have this finished.

I give my heartfelt thanks to my entire committee for their suggestions, support, and shoulders. I could not have accomplished this without your guidance. Amr and Larry, I hope I can pay forward the support and mentoring that you've given me. Sam, I don't know how I'll manage life's great mysteries without barging into your office to ask about them.

Thank you to the Friedmans. To work with Dan is to get to know the whole Friedman clan. Thank you so much for so graciously welcoming me into your home and your lives. Dan and Mary, I'll forever treasure your friendship. To my advisor and mentor Dan specifically, what can I possibly say here? Thank you for being a wonderful mentor, partner, raconteur, hacking buddy, the better part of the "Friedman-and-Hemann" standup duo, and my dear, dear friend. Dan introduced me to the joys of Scheming before we even met, and he has not let up since then. You've taught me so much about computing, navigating the academic world, and life in general. One would think I'd be used to it by now, but I am still surprised anew by the depth of his wisdom. It may take a while to realize it, but in the large Dan usually ends up being correct. My office is lined with books and stacks upon stacks of papers as I write this, and I have my irons in several fires; it's the image of a "little Dan". I continue to put to use all the wisdom you've imparted.

Financial Support. My dissertation has been financed by a teaching assistantship from the Indiana University Department of Computer Science and faculty positions at the Rose-Hulman Institute of Technology and Northeastern University. I am grateful for all this kindness and support.

Abstract

Programmers in related constraint-logic languages should have language semantics that span different implementations and enable reasoning generally about the shared parts of languages' behaviors while reflecting their differences. A wide class of miniKanren languages are syntactic extensions over a small kernel logic programming language with interrelated semantics parameterized by their constraint systems. This thesis characterizes succinctly a set of miniKanren CLP languages parameterized by their constraints, for pure, relational programming by instantiating, for each, portions of the constraint domain. This set of languages carry related components of their declarative and operational semantics that are independent of a particular host language or their particular constraint sets. This characterization bolsters the development of useful tools and aids in solving important tasks with pure relational programming.

Prerequisites and Mathematics

We presume the reader has formal logic background sufficient to complete an introductory logic course, and is familiar with the subject matter from a first course in programming languages for graduate or advanced undergraduate students such as is taught at Indiana University, including the programming language Racket, a dialect of Scheme. Some familiarity with miniKanren, Prolog, or constraint logic programming would be helpful.

In several key places the terminology of the miniKanren community diverges from that of Prolog or the larger logic programming community. Beyond just a divergence, there are naming collisions in which a single word means different but related notions to the different groups. Throughout our work we will use terminology and notation standard in the broader logic programming community. When there are relevant differences and especially at possible points of confusion, we will explain and compare the differences.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1 Aims & Motivation

The declarative programming approach is helping meet a growing demand for programming. The broad trajectory of computer hardware performance over the past 50 years is a dramatic increase in overall computing capacity, of increasingly powerful computers, and decreasing cost for comparable performance [22]. In many domains, this change has rendered approaches that were deemed heretofore infeasible or hampered by unacceptably poor performance now tractable or acceptable. However, increasingly sophisticated and often difficult-to-program computing platforms necessitate a better way to harness this raw power.

While the term is not entirely rigorously defined and there is no universally agreed-upon definition [82], the promise of declarative programming languages is that they should let the user say *what* to compute, instead of *how* to perform the computation. Broadly speaking both functional and logic programming languages have been described as declarative, and we intend this term to be used in opposition to the low-level languages that Perlis [172] describes:

> 8. A programming language is low level when its programs require attention to the irrelevant.

So called declarative programming languages are attractive in part for their promise to make programming easier. This, in turn would help us better address the programming workload, and better distribute it among all parties. Declarative programming is not a panacea for the raft of problems stemming from complexity. But declarative programming—and associated languages, tooling, and frameworks—are arrows in the quiver, or arms in the armory, to meet this challenge.

Declarative programming has seen uptake in industry. Even as venerable an industrial language as C++ now has λ expressions and its designers are considering first-class control structures [105, 125]. Excel's spreadsheet formulas are one of the most common uses of functional programming [90]. This is chiefly the domain of end-user programmers. Recent editions of the tool have made practical use of program synthesis techniques [76]. Moreover, embedded declarative languages are frequently hidden in plain sight in common approaches to design [186]. These are but a few examples of a continuing broader trend.

Another recent trend in programming toward the declarative has been the uptake of domain-specific languages (DSLs): languages custom-built to address particular problems or particular problem areas. DSLs tend to be declarative in that they let the programmer code directly in the intended domain. This is an alternative to the more traditional option of selecting a general-purpose programming language (GPL), and its associated tooling and technology, "off the shelf". The domain-specific language designer fashions a custom "right tool for the job" and perhaps distributes it for others. As a result the group of programming language designers and implementers has also grown in tandem with the trend in DSLs.

As a part of this uptake in declarative programming, logic languages are making a resurgence. We present a short history of logic languages in Section 1.1 and in Chapter 2 provide a more formal treatment of logic languages' background. For now, though, it suffices to say that one of the more recent entries in the long history of logic languages is the miniKanren family. This family is also both the subject of and substrate in which we make the contributions of this dissertation. miniKanren is an up-and-coming language family [206] that has been used effectively in education [65], as well as academic research and industrial applications. Some form of miniKanren is available in more than 50 host languages, typically implemented as an internal DSL. miniKanren could be well poised to help address problems in the areas mentioned above.

Consequently, we should want a formal specification and characterization of a miniKanren language and describe its model of constraints. In the rest of this chapter, we historically and analytically explore logic languages, constraint-logic languages, and domain-specific languages. Our presentation is informal, but assumes the reader has background in formal

logic and the background from a first course in programming languages for graduate or advanced undergraduate students. In the penultimate section we formulate and characterize the problem statement, and in the final section we outline the remainder of this dissertation.

1.1. A Brief Description and History of Logic Languages

The purpose of this brief history is to contextualize the sub-domain in which we place our results, and consequently to help contextualize our work itself. This non-technical, historical introduction to many aspects of this dissertation's topics is an alternative to the more formal background in Chapter 2. Our history of logic programming briefly introduces predecessors to Prolog, before moving on to the birth of Prolog itself. We carry this sketch through to logic programming's semi-quiescence and renaissance.

Logic programming (LP) is a programming paradigm that casts predicate logic, or some limited fragments thereof, as a computational formalism. This style of programming can allow a programmer to reason in terms of a program's logical meaning. In the traditional, older, paradigmatic view of programming languages [130], LP is often listed as one of the major computing paradigms, alongside imperative programming and functional programming. Kowalski et al. [128] summarize the essence of traditional logic programming thusly:

> Ordinary LP solves problems by representing problem-solving procedures by means of clauses of the form
>
> $$H \leftarrow L_1 \wedge \ldots \wedge L_m$$
>
> with $m \geq 0$, H an atom and each L_i a literal. Variables H and L_i are implicitly universally quantified with scope of the entire clause. H is called the *head* and $L_1 \wedge \ldots \wedge L_m$ is called the *body* of the clause. Clauses of this form are used backwards to unfold atoms in goals (existentially quantified conjunctions of literals).

LP emerged as a byproduct of related work in automated theorem proving. Theorem proving mechanisms of the 1950s and 1960s were designed for human reasoning patterns (e.g. natural deduction [171]) and were not especially well-suited as computational formalisms. Robinson's [177] 1965 development of the resolution principle was a key breakthrough. In a sense the history of automated reasoning systems mirrors the history of computer arithmetic systems: just as early computers used, stored, and computed in the natural-for-humans base 10, initial approaches to automated reasoning were rendering human-reasoning systems digitally. From this point of view the analogue to binary for computer reasoning systems is Robinson's *resolution rule*. The principle generally states that, for sets of literals Γ_1 and Γ_2 whose variables are disjoint, and literals L_1 and L_2:

$$\frac{\Gamma_1 \cup \{L_1\} \qquad \Gamma_2 \cup \{L_2\}}{(\Gamma_1 \cup \Gamma_2)\phi}\,\phi$$

where ϕ is the *most general unifier* (MGU) of L_1 and $\overline{L_2}$. In the course of developing resolution Robinson rediscovered *unification*, a kind of two-way pattern matching used to implement resolution. We will return to the concept of an MGU and unification in Section 2.3. We say *rediscovered* unification, because Robinson independently developed an approach previously known, at least for special cases, to Post [see 210], Herbrand [89], and Prawitz [173]. Robinson certainly introduced the *name* "unification" for this process and singled it out for study, and it was through Robinson that unification became widely recognized as a powerful tool for automated reasoning. Unification is broadly applicable to a number of fields and areas of study, including theorem proving, term-rewriting systems, machine learning, natural-language processing, and logic programming among many others [122]. This broad applicability is also evident from the various generalizations that enable unification in more expressive algebras. These include *E-unification* [109] (solving equalities of sets of terms modulo a typically more powerful equational theory), and nominal unification [209], among many others. Unification is so important and broadly applicable because we can implement it efficiently in so many of these cases. Both Martelli and Montanari [153] and Paterson

and Wegman [170] are responsible for first-order unification algorithms over term algebras that have proven linear time bounds, and subsequent improvements have introduced more practically efficient versions (see Siekmann [198, §3.1.1]).

While Robinson's original resolution rule was a vast leap forward for automated theorem proving, guiding the proof search was still difficult. Subsequent theoretical advancements showed how to better employ particular instances of the general resolution principle in ways that simplify theorem provers' implementations and guide search to reduce the search space while maintaining completeness. These later refinements led to a version termed SLD-resolution [176] that provides a reasonably efficient complete search technique for a fragment of first-order logic large enough to practically program in. Taken together, these pieces enabled logic programming. The key insight of the logic languages' original designers was that the programmer could code directly in an expressive, executable subset of formal logic. LP languages aim to unite the language in which the programmer or project manager specifies the behaviors of the program (traditionally not executable), and the language in which the programmer actually writes code. The shift from automatic theorem proving and LP is partly attitudinal, a question of viewpoint. One of the things that logic can be used to express are computable functions and procedures. One of the things proof procedures can do is perform deductions that execute programs.

LP languages differ from λ-calculus based formalisms, quoth Kowalski [127], in that they are derived "from the normative study of human logic, rather than from investigations into the mathematical logic of functions." LP languages broadly distinguish themselves from earlier Planner-style systems by their operational behavior. Planner-style languages are "bottom-up" in that they use assertions to generate new assertions [127], whereas logic programming languages are goal-directed reasoning systems that, from old goals, produce new goals; these are called "top down". One of the first (although not actually the first [53, 54]) and certainly the most widely known of these logic languages is Prolog [72]. "Prolog" is at least in part a portmanteau of the French words *programation* and *logique*.

Prolog was first born as a library or tool for natural-language processing but grew into a language [42]. Early Prologs all implemented unification with an occurs-check, but this was soon removed for efficiency considerations [38]. Colmerauer [40] showed the resulting unification procedure could be reasonably interpreted as unification on infinite trees, and this work of Colmerauer's exhibits one early strain of research connecting logic programming and general constraint solving. The theoretical improvements were matched by technological and engineering developments and insights. Between both, Prolog quickly spread out from the Marseille group to Edinburgh and elsewhere through conferences, workshops, summer schools and publications, and an early "sneaker net" that distributed implementations from group to group. People expressed intense interest in the paradigm, and experimented with it in various ways and with different motivations. Through the 1970s researchers continued to revisit and extend logic based language design and to push the boundaries.

The announcement of the new Japanese Fifth Generation Computing System Project punctuated the end of the 1970s and beginning of the 1980s. Announced in 1981, its main goal was to revolutionize computing by developing massively parallel machines tailored toward AI. However, logic programming emerged as the favored language paradigm for implementing software, and soon became another of the FGCP's central efforts. They hoped to make simultaneous leaps forward in programming in parallel logic languages, as well as major advances in hardware. In combination, these goals proved overly ambitious. Alternate emerging technologies (e.g. object-oriented languages, x86 hardware) proved to out-compete their efforts in these directions.

While the FGCP is not synonymous with logic programming's progress in the 1980s, it is emblematic. During this decade concerns with Prolog's control's inefficiency led to a proliferation of designs for new or enhanced control features. This led to a splintering of logic programming into various specific application-tailored dialects. In addition, there were limitations of a single, uniform proof procedure, making it tough to be efficient to execute for medium-large domains, and conversion to clausal form and resolution proof hides some of the underlying structure of the problem and the proof of the solution. Disappointment

with the progress of logic programming and artificial intelligence during this period and the successes of other subfields and technologies led to an "AI winter" that cooled overall interest in the technology.

1.1.1. Resurgence of logic programming and beyond. With fundamental research, there is often a time lapse between the research and its application to more practical or commercial efforts. There is evidence to suggest that the LP boom of the 1980s was partly just ahead of its time. Other classes of logic languages that exist have emerged some from Prolog's shadow. Datalog, for instance, is one notable result. Datalog is a decidable language in which the terms to be computed do not allow function symbols. The development of answer set programming and tabling also served to make logic programming more declarative. As another data point, in just 2016 the Picat programming language, a multi-paradigm constraint solving and planning language won first place and a cash prize at a major New York-based media lab summit. Media summit awards rarely go to programming languages. The miniKanren family of embedded logic programming languages is also a member of this next generation of logic languages and is reaching more common use.

There is of course much more to say about the history and the current state of the art in logic programming. In addition to some of the personal and historical accounts referenced in this section, the interested reader could consult volumes of the *Handbook of Logic in Artificial Intelligence and Logic Programming* series for the vast amount of information they contain as well as their exhaustive bibliographies, and consider Hewitt's [91] "Middle History of Logic Programming Resolution, Planner, Prolog, and the Japanese Fifth Generation Project" for an alternative view connecting Prolog to the earlier Planner. For the development of and the subsequent rebirth of datalog, see Huang et al. [102]. The interested reader should also consult Balbin and Lecot's [12] older logic programming bibliography. While the field has advanced since this bibliography's publication in 1985, this author knows no other publication that so well categorizes such an exhaustive listing of important early results, most of which are still relevant.

1.2. Constraint and Constraint-Logic Programming

In this section we revisit a middle era in the history of logic programming when constraint-logic programming emerged. The constraint-logic programming paradigm combines logic programming with *constraint programming*. Constraint solving serves as a declarative paradigm in its own right [204], astride functional and logic programming. In constraint programming, computation proceeds by satisfying a collection of constraints that describe the contours of the final answer.

Programming with constraints was investigated at least as early 1964 [205], well before the emergence of widespread interest in constraint logic programming. Researchers combined constraint programming with aspects of other programming paradigms. Even these mixed paradigms are themselves broad areas of programming language research. Many of these mixtures benefit from connections with yet other areas of work in artificial intelligence, language design, and operations research, while at the same time inspiring new approaches to addressing problems in those same application areas. As we noted before, scholars [106] use the term *constraint-logic programming* (CLP) for the combination of constraint- and logic programming; it is to this combination that our research primarily speaks.

1.2.1. The Constraint-Logic Programming Paradigm. Constraint logic programming more than merely kludges constraint solving into logic programming. CLP should be properly understood as itself an independent general declarative paradigm. As Maher [148] writes,

> Indeed, in some sense both constraint programming and logic programming can be considered part of an umbrella relational programming paradigm.

By the mid 1980s, several groups were working to extend logic programming languages with constraints. These include research emerging from the logic programming community itself such as the groups in Marseille and Edinburgh, but also included operations research-based work such as that undertaken at the ECRC. Through this lens traditional logic

programming emerges a special case [110]. Much of the following discussion is adapted from Maher [147, 148], Cohen [39], Kriwaczek [131], and Wallace [219]—any of which the interested reader should consult.

Constraint logic programming [40, 106], in the sense we use in this dissertation, extends the bare-bones sense of logic programming from Kowalski of Section 1.1 by including in CLP languages a collection of special primitive relations called "constraints", and a means by which to solve them. The programmer does not define these constraints by clauses for the implementation to evaluate through backwards reasoning by unfolding, as is the case with predicates in traditional LP. Instead, the CLP language implementer builds these constraints into the language's implementation itself. A language implementation solves constraints by one or more dedicated *constraint solvers*. Lassez [135], in "From LP to LP: Programming with Constraints", describes the chief requirements thusly (emphasis in the original):

> Let us review the three main points, for a given domain:
>
> **A set of constraints is viewed as an implicit representation of the set of all constraints that it entails.**
>
> **There is a query system such that an answer to a query $Q(x, y)$ is a relationship that is satisfied if and only if the query is entailed by the system.**
>
> And most importantly:
>
> **There exists a SINGLE algorithm to answer all queries (an oracle).**

We distinguish CLP in the sense of Colmerauer, Jaffar and Lassez and this dissertation, from a separate, related, contemporaneously-developed approach. This second style comes from research into constraint propagation that emerged out of operations research and work in artificial intelligence [88, 219]; research in the CHIP language at the ECRC in the 1980s typifies this style of CLP. This second approach, instead of adding new, additional features, extends the behaviors of existing LP language features. These two CLP paradigms are orthogonal, and compatible [140]. When we subsequently discuss CLP, we will refer to constraint logic programming in the first sense.

There are many reasons that a language designer might provide some facility as a constraint, rather than leaving the logic programmer to implement it, say, as a library of a standard LP language. Firstly, it might be that the pure logic programming predicate implementation of whatever facility the constraint provides would be insufficiently expressive or too inefficient to benefit end users (see, e.g. Jaffar and Maher [110, §1.2]). Secondly, when designing program idioms to represent negative information, programmers may find it more natural to represent this information negatively [107, 137]. Thirdly, constraints provide many of the benefits of structure-sharing. The programmer may deem some number of a term's instances "similar enough" that he would like to compress these many terms into one overarching representation limited to those instances fulfilling some constraint.[1] Furthermore—and perhaps more importantly—the addition of constraints can add expressivity in surprising ways. Constraints can represent information implicitly, as opposed to explicitly and exhaustively representing that same information [117, 134]. When almost every (all but a finite few) possibility solves some problem, equations alone cannot finitely express infinitely many solutions. However, auxiliary constraints can permit a single finite construction to represent the solution set. For instance, when there is no way to positively, finitely enumerate infinitely many possible values for a term, a disequality constraint can instead explicitly represent some finite quantity of *disallowed* terms. An equation between two terms and a disequality between variables of each term can finitely schematize what otherwise require an infinite number of plain equalities. Constraints frequently provide more expressive conditions than just syntactic equality or disequality over first-order terms.

Much of the work that has led to interest in CLP as a paradigm emerged from research in logic programming, specifically Prolog. The developers of logic programming languages have explored constraints beyond equality since at least the earliest versions of Prolog. An early Prolog implementation from the Marseille group in 1972 contains the `dif/2` predicate that implements syntactic disequality constraints [42]. Curiously, the disequality constraints of this early implementation are well-behaved as regards non-ground terms, and these

[1] With our system we will find that we cannot express finite constraints or constraints with finite domains. However, this is not a limitation of constraint systems generally.

Prolog systems could emit these disequalities as part of the final answer—capabilities often forgone in future implementations [38]. Colmerauer and the Marseille group reintroduced a less capable form of disequality constraints over a term language of rational trees in Prolog II [41], and implementers continued to experimentally blend a variety of constraints with logic programming languages [41].

A variety of Prolog extensions have been built that have added constraints enhancing the expressiveness of the language. CLP($\mathscr{R}$) [113] added inequality constraints (i.e. $\leq$ and $\geq$) over the real numbers. Other typical examples are set membership and subset relationships and Boolean satisfiability [110]. These are but a few among a variety of extensions that one could rightly group together as CLP languages. This diversity of systems demonstrates that there is a wide design space in which to construct a CLP language. Choices such as whether to allow constraints that modify a program's control flow, whether constraints can be dynamically generated, a particular choice or choices of constraint domain, and the nature of the precise constraint solving algorithm create a whole raft of possibilities.

In this proliferation, it came to be that many CLP languages included specialized dedicated solvers and additional control mechanisms (e.g. arc consistency to restrict domains and check domain restrictions, freezing and thawing constraints to delay or force their evaluation). These constraints were typically implemented as a fixed group of primitive operators in a fixed language and intended to operate over a precise, well-known constraint domain. Usually programmers could not mix the constraints from different domains, even if designers implemented multiple collections of these constraints inside the same language.

For a time there was some concern that these various extensions sacrifice the unique semantic properties of logic programming [106, 143]. Individual extensions of LP languages by constraints often came equipped with their own unique semantics. It wasn't clear that the important properties of LP languages would carry over with these extensions. It was certainly cumbersome to reason at this level about the behavior of each individual extension or blend of extensions when constructing a new CLP system.

1.3. The CLP Scheme

The *CLP Scheme*[2] emerged from research to characterize collectively some different extensions of Prolog-like languages that each supply additional constraints. Jaffar and Lassez's CLP Scheme [106] separates the semantics of constraint solving from the semantics of the search [111, 213]. In separating the particulars of a languages' constraint system from its control, the Scheme enables reasoning generically about constraint systems. In a way one can view the CLP Scheme as expanding Kowalski's thesis ("Algorithm = Logic + Control") to the context of constraint-logic programming.[3]

Logic programming languages are special in part for the existence of equivalent operational, logical, and functional (i.e. denotational, fix-point) semantics. Somewhat surprising to researchers at the time, constraint logic programming languages retain this property. Moreover, since CLP programs compute over some particular domain of computation, such programs also carry "algebraic" semantics: a semantics which we can define directly on the algebraic structure of that domain.

In earlier research, Jaffar et al. [109] showed with their "Logic Programming Scheme" that those key semantic results hold for more general notions of equality than the typical syntactic equality of Prolog. This provided a firmly-grounded theoretical foundation for programming in logic languages with equational theories of the sort that, for instance, make programming with arithmetic expressions more natural. With the CLP Scheme, Jaffar & Lassez show the semantic results of ordinary logic programming generalize still further to hold for a wide class of general constraint logic languages as well. Critically, Jaffar et al. [111] write,

[2]Though it has "Scheme" in the name, the CLP scheme has nothing to do with the Scheme programming language. This similarity is just a fortuitous happenstance.

[3]In "Programming with Constraints: An Introduction", Marriott and Stuckey [152, p. 151] include the requirement for a constraint simplifier. We do not follow them in requiring this component.

> The key insight of the CLP Scheme is that for these languages the operational semantics, declarative semantics, and the relationship between them can be parameterized by a choice of constraints, solver, and an algebraic and logical semantics for the constraints.

Constraint logic programming replaces unification over terms (itself a kind of constraint solving) by some other constraint solving in a solution-compact domain with a satisfaction-complete theory subject to certain additional requirements. We defer a detailed explanation to Chapter 2; we include this to say that it is a "scheme" in that a schematized language CLP($\mathscr{X}$) describes a family of languages that share certain common properties [147]. The CLP Scheme parameterizes a CLP language's operational, declarative, and algebraic semantics—as well as the relationships between them—by the computational domain.

A language designer instantiates $\mathscr{X}$ with an appropriate choice of signature, mathematical structure, a class of closed formulae that form the constraints, a first-order theory, and a constraint solver. Together these define the constraint domain. We give the set of constraints an operational meaning via the solver, a logical meaning via the constraint theory, and an algebraic meaning via the model that is the constraints' intended interpretation (we discuss this in detail in Chapter 2). Thus when provided a particular constraint domain, the scheme defines an entire particular constraint logic programming language of that family for an end-user to write programs and also a mechanism for evaluating programs and queries written in that language. As constraint logic programming generalizes traditional logic programming, one can view traditional LP as constraint logic programming with equations in the domain of finite trees. CLP also generalizes the characterization of an answer. In an LP language, an answer is a substitution; in CLP, an answer is a constraint. As we have discussed, constraints can sometimes represent intensionally in one answer the information of many—in some cases infinitely many—substitutions.

Beyond just checking for the consistency of a logical statement (equivalently, beyond just executing the program), fully featured CLP systems almost always include tests for: *implication*, or the entailment of some constraint by another; *projection* of constraints, that is, presenting answers restricted to constraints over the variables of interest; and *determinacy detection*, recognizing when we can replace a set of constraints with equalities [110].

1.4. Domain-specific Programming Languages

In this section, we describe the history of DSLs. "Domain specific" languages are often contrasted with "general purpose" programming languages (GPLs). The notions of "domain-specific" and "general-purpose", as they're generally used, are not rigorously defined. Necessarily, certain language design choices will bias a language toward one class of tasks and consequently away from some others. This is true even for languages envisioned with the broadest of use-cases. In this sense, it's difficult to view any language as truly "general purpose". Nor is a language's status as "domain-specific" or "general" necessarily fixed. Each of COBOL, Fortran, and Lisp—three of the most venerable general-purpose programming languages—arguably began life as a domain-specific language. One can still detect hints of their earlier life in their names, which are abbreviations for, respectively, "common business-oriented language", "Formula Translation", and "LISt Processor". Furthermore, as computing platforms and the spectrum of computing tasks has grown and diversified, what heretofore seemed to be the general task of computation (e.g. numerical calculations) has come to be just one of many specific domains. Generally though, domain-specific languages are languages that, as the name suggests, target narrowly-defined problems or problem areas. The promise of DSLs [16] is that we can more quickly and correctly map a solution to code in a language specifically tailored to the problem than we could in a more general-purpose language. In a GPL, it may require a lot of programming to construct a set of bespoke program fragments that adequately express the concepts for programming in a particular domain. Or, to quote Perlis once again,

> 26. There will always be things we wish to say in our programs that in all known languages can only be said poorly.

As such it often makes sense to craft a special-purpose language for the job. In a sense the apogee of this approach is language-oriented programming [224], of which the Racket language, for instance, represents one viewpoint [57]. In a nutshell, this approach means solving a particular problem by designing a language specifically for that problem.

Designing and using DSLs—building as needed "the right tool for the job"—is now a more common trend. As these languages are custom-built and sometimes even single-use, we will often collapse the distinction between the language and its implementation, and frequently the lone implementation defines the language.

There are a smattering of reasonable ways to classify DSLs. One such distinction is whether the language is a *standalone* or an *embedded* DSL. We can view a standalone DSL as a language built using the typical approach of a general purpose language, with its own syntax and semantics, using the usual techniques for constructing a programming language. These are also known as *external* DSLs [62, 63]. For these, the language under construction just coincidentally targets a particular purpose. Developing and maintaining infrastructure for such languages can be costly and time-consuming, as these will often require full tool-chains for the programmer and may require re-implementing in the DSL at least some features common to many existing languages (e.g. conditional structures).

Languages of the latter sort under this first distinction, *embedded DSLs*, are languages whose programs are source code in some existing programming language, known as the *embedding language* or *host*. By contrast with the former, these are also known as *internal* DSLs. Retaining the host language's surface syntax for the DSL's programs lets some of the host language tooling and infrastructure bleed through to the DSL. For instance, if a putative program in an internal DSL is invalid syntax in the host language, host language tooling can indicate this failure. Other niceties like debugging tools and IDE integration can also carry over, at least in part. The cost this carrying over this tooling is the perhaps onerous restrictions of the host language's surface syntax. If that price is acceptable though, the language designer avoids much wheel-reinvention by "piggybacking" on the host language. We can further subdivide these embedded domain specific languages.

In a *deep embedding*, programs of the DSL are abstract syntax trees built as data in the host language. Here, the implementer defines language constructs of the DSL as host-language data constructors. In a deep embedding, there is some host-language value representation for each program in the embedded language. This representation must be faithful in the sense that, for each action we provide on embedded-language programs (e.g. `eval`) we can write some host-language code that operates on such a value and performs the desired operation. We implement execution, optimizations, or other such operations over embedded language programs as host traversals over the terms of that specified datatype. In a *shallow* embedding, the DSL language implementer directly defines the language constructs of the embedded language by translation to their semantics in the host language.[4] Gibbons and Wu [73] discuss connections between the deep and shallow styles of DSL implementation and cleanly and concisely explains the conceptual background. This makes their paper a reasonable entry point for the interested reader.

We could compare DSLs along several other distinct axes as well. However, will not linger here describing the trade-offs between these approaches in DSL design. We describe some major decisions in DSL language design primarily to help situate miniKanren languages, (typically implemented as shallowly-embedded domain-specific programming languages) and our work generally, in the relevant context.

1.5. Situating miniKanrens in Context

miniKanren is a family of related languages with an overlapping set of operators and a common design philosophy. The seminal implementation, also named "miniKanren", was first presented in *The Reasoned Schemer*, and since then there has been a profusion of miniKanren languages. These have included both additional constraints and control operators.

[4]Gibbons and Wu date the terms "deep" and "shallow" to Boulton et al.'s [19] work embedding hardware description languages.

miniKanrens distinguish themselves from earlier logic languages by their interleaving depth-first search, their growing variety of primitive constraints beyond first-order syntactic equality, and their community's emphasis on pure relational programming. Different aims and emphases led to the languages' different design decisions, and consequently lead us to revisit traditional trade-offs. The completeness of this acceptably-efficient search, for instance, makes miniKanrens in some respects "more declarative". miniKanren programmers rely on these unique properties to create theorem provers that double as proof assistants, type checkers that double as type inhabiters, and interpreters that perform interesting program synthesis tasks such as generating quines [25, 26, 27, 164]. miniKanren languages are increasingly important and seeing some real use in industry [23, 75, 189, 206]. Canonically, miniKanrens are internal, (shallowly)-embedded DSLs that permit rapid prototyping and design of constraints and CLP systems. Shallow embeddings provide an easily-modified interface. There are now a large number of implementations; as they are simple, concise, customizable, and highly portable language implementations, miniKanren is now even a substrate through which researchers investigate other logic programming questions [25]. To summarize, the widely-embedded constraint-logic programming language miniKanren, which guarantees a fair search and provides intuitive explanation for functional programmers, could play some part in the declarative programming's future.

1.6. The Terrain

This section surveys some of the topics we subsequently consider in detail to give a rough feel for these languages by example. These examples demonstrate exemplary constraints and programs in CLP languages over exemplary constraint domains and have enough complexity to convey the interesting facets. This more intuitively and directly motivates this dissertation's work, and we necessarily omit some of the technical details here.

The example of Listing 1.1 demonstrates a recursive miniKanren relation and the use of an equality constraint, `==`. This use is the miniKanren equivalent of a Prolog query with `append/3`. Consider the constraint infrastructure as separate from the search and control architecture. Imagine wanting to design an enhanced constraint domain with constraints

beyond equality. Further still, imagine having already defined constraints over a domain $\mathscr{X}$, and then wanting to add more. The constraint writer must consider the envisioned extension to the existing constraint system—and the constraint writer may have difficulty foreseeing if they all "play nicely".

```
> (run 3 (q)
    (fresh (l s)
       (== `(,l ,s) q)
       (append l s '(t u v w x))))
```

Listing 1.1. An example invocation of the `append` relation

Example Listing 1.2 exhibits a query involving constraints from one such more complex domain. Intuitively, the query asks for a `q` such that, with regard to three other auxiliary variables, `a`, `b`, and `c`, `q` is a list not containing `c` that equals a pair of `a` and `b`, when `b` is not itself a pair. Then, finally, we assert that `c` is the empty list. It may not be immediately clear to the reader if such a `q` exists, and if so, how to find it. Nor is it obvious how many different such `q` there will be or the relationships between those values.

```
> (run 1 (q)
    (fresh (a b c)
      (listo q)
      (absento c q)
      (not-pairo b)
      (== q (cons a b))
      (absento b a)
      (nullo c)))
```

Listing 1.2. An example constraint from an exemplary constraint domain.

Aside from first-order syntactic equality constraints and all the examples of Listing 1.2, what more "kinds" of constraints should a constraint writer be able to add? We want to generalize from the fixed, specific constraint classes of these examples and the kinds of things that we miniKanreners already do to a useful midpoint still short of the full, general CLP-Scheme constraint domains. For instance, we will exclude finite domain constraints. Solving constraints of such domains can be computationally expensive. Assuming we can

find such a midpoint, we should hope to capture this class with a good description beyond just "the class of constraints of our system". An independent characterization of this class of constraints leads to more examples and a better understanding of these constraint domains. Given the definition specifying such a constraint domain $\mathscr{X}$, we will automatically generate a shallowly embedded implementation of a CLP($\mathscr{X}$) language. We will want to know why the class of constraints that we capture permits such implementations.

At present, the source code of a miniKanren language's implementation is often unkind to the intrigued but puzzled novice reader. Implementations' size and complexity grows by orders of magnitude with the ad-hoc addition of just a handful of new constraints. Moreover, these additions vastly complicate constraint solving. Host-language macros that provide surface syntax have obscured the details of its search. These many language's implementations have provided no other semantics than their source code. This is increasingly untenable.

What is it that these implementations are in common implementing?

The community wants for a more formal specification of at least some of the common behaviors of some of these languages' implementations. These will enable comparisons of design decisions at an abstract level rather than code-level. This will help explain differences among these miniKanrens, and between them and other logic languages. Equipped with this background, we can now proceed to our problem statement.

There is a need to make declarative programming more widely available, and specifically relational, or pure logic, programming available and accessible to more people and within their own current favorite language. This dissertation aims to advance that goal. We will use the tools of formal semantics to impose order on the miniKanren language family's organic growth by situating these languages in a design space by their term languages, and their constraint sets, within the CLP Scheme. The main task of this dissertation is to show that:

> a wide class of miniKanren languages are syntactic extensions over a small kernel logic programming language with interrelated semantics parameterized by their constraint systems, and this characterization bolsters the development of useful tools and aids in solving important tasks with pure relational programming.

We argue this thesis by separately demonstrating various pieces:

- *a small kernel logic programming language*: We exhibit *microKanren*, a small (constraint) logic language amenable to direct embedding in any eager functional host.
- *miniKanren languages are syntactic extensions*: We then demonstrate via host-language macros a reduction from miniKanren programs with first-order equality over a sufficiently expressive term language to microKanren programs.
- *parameterized by their constraint systems*: We situate microKanren constraints within the CLP Scheme that provides logical, algebraic, and operational semantics for constraint systems.
- *interrelated semantics*: We lift these constraint systems' semantics into the traditional semantics for logic programming languages.
- *bolsters the development of useful tools and aids in solving important tasks*: We exhibit example applications enabled by the above results, including novel miniKanren constraints and their applications.

Some of the content of this dissertation has been published previously. Our embedding, and the development of our search strategy, impure extensions, and recovery of pure miniKanren have been described in Hemann et al. [86] and Hemann and Friedman [83], and we described our classes of Herbrand constraints in Hemann and Friedman [85] and Hemann and Friedman [84]. Some of the programming techniques and examples we describe were previously discussed in Hemann, Swords, and Moss [87], and we previously described some of the tooling and teaching methods we suggest in Brady et al. [20].

1.7. Dissertation Outline

We develop the remainder of this dissertation in roughly three parts. Chapter 2 gives the basic notional background and introduces generally the technical aspects of logic and constraint-logic programming. These include some fundamentals and foundations of logic and constraint logic programming. In Chapter 2 we also introduce the CLP Scheme, within which we formulate our results.

Chapter 3 contains the main results of our approach. We give operational, logical, and algebraic semantics for microKanren constraint systems. In Section 3.2 we construct particular constraint systems using the results from the previous chapter, and also exhibit counter-examples and possible pitfalls. Sections 3.5 to 3.9 describe the syntax of the core microKanren languages, and Sections 3.10 to 3.12 describe the syntax of the miniKanren languages and the implementation of their embeddings.

Chapter 4 contains a collection of novel examples and uses of the constraint logic programming languages for which our framework generates embedded implementations. These examples include novel applications in program synthesis that our constraints framework facilitates and new miniKanren-specific relational programming techniques. Chapter 5 describes some of the ample related work surrounding this dissertation. Chapter 6 summarizes our results, discusses a number of remaining open problems related to the work of this dissertation, suggests other directions for future research, and concludes.

Chapter 2 Prolegomena, Programming, & Prolog

This chapter explains some preliminaries essential for characterizing our results. We first define terms and then describe first-order languages with constraints. We next describe the underpinnings of constraint systems and the CLP Scheme. We incidentally define constraint systems, constraint domains, and constraint solvers in the process. We then describe our class of constraints, and then the behavior of our constraint systems; we remark on important collections of expressions as they arise.

We assume the reader is familiar with fundamental results in first order logic and logic programming, and we will not recapitulate that background here. Instead, we refer the reader to Enderton [55] or Mendelson [155] for general background, and to Lloyd [144], Doets [51], or Downward [52] for introductions tailored to logic programming applications. Many of the concepts we use here are foundational and common among elementary logic texts; others are particular to specialized use within our subarea. We will occasionally comment on some of the latter. Authors use varying notational conventions and have some slight differences in how they develop certain technical terminology. This diversity of treatments provides a range of notational choices. We mainly adopt our conventions from Jaffar et al. [111], Jouannaud and Kirchner [114], and Lassez et al. [137] for this chapter and the remainder this dissertation.

2.1. Preliminaries

To elide some important but tangential aspects from formal computability, let us say simply that a set $Y \subseteq X$ is *recursive* if it has a total computable characteristic function $\mathbf{1}_Y$ on X. Under some fixed encoding scheme, if we index a partition P of X by a set I so that each $i \in I$ codes for the characteristic function of $\mathbf{1}_{P_i}$ on X, we say that I *exhibits* a partition

on X. We will use $\#\cdot$ or $\lceil\cdot\rceil$ to talk about the program that codes for the given function. We will find of principle interest those sets I that exhibit a finite computable partition on an underlying set X. As another notational convention, we will sometimes write $|X| < \omega$ to say that X is finite, and $|X| = \omega$ to say that X is countably infinite. For a given countably infinite set X, let $[X]^{\omega}$ ($[X]^{<\omega}$) denote the set of all subsets of X of cardinality ω (less than ω). P *finitely partitions* X if P is a partition of X and $|P| \leq \omega$. We sometimes use *blocks* to refer to the sets in a partition. If a set X is drawn from some fixed universe U, then the term "relative complement" is used to describe the set $\overline{X}$. This is also typeset as X^c.

2.2. Terms and Term Algebras

We define an algebra by a pair of disjoint sets $(\mathscr{V}, \mathscr{F})$ with *carrier* $\mathscr{V}$ and a set of finitary ranked *operators* $\mathscr{F}$—that is, a set equipped with a total function ar from $\mathscr{F}$ to $\mathbb{N}$. We say $f \in \mathscr{F}$ has an *arity* of $ar(f)$.

For a given algebra $(\mathscr{V}, \mathscr{F})$ with $\mathscr{F}$ and $\mathscr{V}$ denumerable, disjoint, and each with decidable membership, and ar appropriately defined, we write $\mathscr{T}(\mathscr{F}, \mathscr{V})$ for the first-order *terms of* $\mathscr{F}$ *over* $\mathscr{V}$ By viewing, the construction of a term from given terms in $\mathscr{T}(\mathscr{F}, \mathscr{V})$ and an operation symbol, as itself an operation, we can view the set $\mathscr{T}(\mathscr{F}, \mathscr{V})$ as an algebra. This algebra has carrier $\mathscr{T}(\mathscr{F}, \mathscr{V})$, and we call this the "free term algebra" on $\mathscr{V}$ of $\mathscr{T}(\mathscr{F}, \mathscr{V})$. In logic programming, this algebra is sometimes instead referred to as a pre-interpretation J (e.g. Lloyd [144]), and we will better understand this characterization in Section 2.4 on page 25.

Given such a term algebra, we call $\mathscr{V}$ the set of *variables* and $\mathscr{F}$ the set of *function symbols*. As a notational convention, we denote variables with v and w, function symbols with f and g, and terms with the letters t and u—each possibly subscripted. We will use $\overrightarrow{t}^n$ for the sequence of terms $t_1 \ldots t_n$, or simply $\overrightarrow{t}$ when the reader can infer n from context or when the precise arity is unimportant. We will not concern ourselves with malformed, insufficiently saturated, or over-saturated terms; unless stated otherwise, the reader should assume we construct all terms correctly and with appropriate arities. We use the following conventions to describe function symbols. We call *constant*, or *nullary* those

function symbols of arity 0. We use the term *unary* for function symbols of arity 1, and reserve *polyadic* to mean function symbols of a fixed arity at least two. We describe those function symbols with a fixed, positive arity (unary and polyadic functions collectively) as *posary*. We will extend these conventions to functions in the obvious manner.

For any choice of a countably infinite set $\mathscr{V}$ disjoint from $\mathscr{F}$, the operators generate an isomorphic term algebra. Thus the particular variable set we choose does not matter. Within the context of a known, fixed set $\mathscr{V}$, we can identify a term algebra with its signature, and call the elements of $\mathscr{T}(\mathscr{F}, \mathscr{V})$ simply $\mathscr{F}$-terms. When the particular set $\mathscr{F}$ is also unimportant or the reader can infer $\mathscr{F}$ and $\mathscr{V}$ from context, we refer to the set $\mathscr{T}(\mathscr{F}, \mathscr{V})$ as $\mathscr{T}$ and its elements simply as *terms*. The term algebra is uniquely generated by the set of variables $\mathscr{V}$. For a given term algebra, its *term algebra signature* is the set of the algebra's function symbols, together with the arity function for that set. We also describe the set of terms over an algebraic signature. We will call the set of all ground terms over operators $\mathscr{F}$, denoted $\mathscr{T}(\mathscr{F}, \emptyset)$, the *Herbrand universe* for $\mathscr{F}$.[1]

This dissertation concerns exclusively infinite term algebras, and we raise this distinction because some of our results hold only in infinite Herbrand universes. Our signatures will be at most countably infinite. A "finitary signature" simply means that all the function and relation symbols are of finite arity. For a signature to generate an infinite term algebra it suffices either to have infinitely-many nullary constructors, or instead to have at least one nullary constructor and at least one unary constructor. We follow Kunen's [133] "Signed Data Dependencies in Logic Programs" in requiring an infinite "universal language" in which all programs and queries are executed, and we define the term language against which we write programs in Section 3.2.1. Since there is no scope for confusion, we will use "term algebra" to mean a term algebra with infinitely many constants generating an infinite Herbrand universe, and we will henceforth take "term" to mean an element of such an algebra.

[1] Davis [47, p. 10] suggests that "Herbrand universe" should perhaps instead be the "Skolem universe", as Herbrand first published his work two years after Skolem.

2.3. Substitutions, Equations, and Unification

A *valuation*, or state (over some domain), is a mapping from a set of variables to a set of domain elements. Given a state σ, we can denote its restriction to a set of variables $\mathscr{X}$ by $\sigma|_X$. For a finite $\mathscr{X}$, this restriction is a *substitution*. A substitution into $\mathscr{T}$ maps each variable to a term. By expanding substitutions' domains to terms in the obvious way, we can also describe a $\mathscr{T}$-substitution σ as an endomorphism on the term algebra $\mathscr{T}$. Conceptually, a substitution is a mapping from variables to domain elements that is almost everywhere the identity mapping, so we can finitely represent a substitution by its non-identity *bindings*. We will use σ or θ (often omitting the domain restriction) to represent a generic substitution, again possibly subscripted or primed, and we also use ρ and μ for substitutions in particular classes. We use $\sigma \leq \sigma'$ for the *instantiation preorder* on substitutions.

Unification is the general mechanism for determining, in some abstract algebra, "can we find an object z that fits two given descriptions x and y?"[2] Generally we want not just to determine if a solution exists, but also to construct one. Syntactic unification is a process for constructing an assignment for a set of terms' variables that will make those terms syntactically identical. We use syntactic unification to solve finite sets of first-order term equations modulo a theory of syntactic equality, and we will henceforth use simply "equations" as a shorthand, since there is no scope for ambiguity.

2.4. Interpretation

We ascribe meaning to syntax by mapping into a *structure*. A structure S is a triple of a set S, called the *domain* of the structure (or *universe*), a signature of the structure, and an *interpretation* function I mapping each non-logical symbol in the language to that symbol's meaning in the structure.

[2]Lassez et al. [137], for instance, investigate unification and the solution of equations in degenerate cases, such as the case of a fixed, finite number of constants, or variables, or function symbols. Our term languages exclude these atypical cases, and so we can omit further discussion of them.

We assign meanings to the signature's function symbols with a pre-interpretation J that maps each n-ary symbol to an n-ary function on the domain. The Herbrand pre-interpretation assigns each symbol f to the free syntactic constructor for f. An interpretation I is a pre-interpretation J (a mapping from the term algebra into functions on the domain S) *extended by* an assignment to each n-ary relation symbol p in $\mathscr{L}$. For uniformity we denote the mapping f_J by f_I. An Herbrand interpretation is an interpretation I based on the Herbrand pre-interpretation. We say *the*, because J fixes the interpretations of the function symbols. In an Herbrand interpretation, we identify the interpretation of p_I with a subset of n-tuples of the domain. We call a term interpretation *substitution-closed* if $A \in I$ implies that for *inst*(A), the set of instantiations of A, *inst*$(A) \subseteq I$.

2.5. Elementary Logic

We separate a first-order language $\mathscr{L}$'s *non-logical syntax* from the *logical syntax*. We inductively build formulae over programmed atoms and primitive constraints using the standard propositional connectives, and we also fix the non-logical syntax to the standard logical symbols of first-order logic.[3] We treat quantifiers in the standard fashion, and we will take $Qx_1, \ldots, x_n.\phi$, $Q\overrightarrow{x}.\phi$, or simply $Q.\phi$ as shorthand for $Qx_1.\ldots.Qx_n.\phi$. We will usually omit mention of the particular logic language $\mathscr{L}$. We use metavariables ϕ and ψ to connote arbitrary formulae, again possibly subscripted. We construct a *language signature* Σ by extending a constraint domain signature $\Sigma_{\mathscr{C}}$ with a programmed relation signature Π, provided the symbols of Π are disjoint from those of $\Sigma_{\mathscr{C}}$ and that the arity function is also properly extended.

Because we fix the logical syntax of $\mathscr{L}$ for all of the languages we will define, a non-logical signature Σ uniquely specifies a first-order language $\mathscr{L}$ consisting of all well-formed formulae built from $\Sigma \cup \mathscr{L} \cup \mathscr{V}$. For simplicity, we henceforth assume that each of the sets of symbols so far described are all pairwise disjoint. With respect to a term algebra $\mathscr{T}$, we

[3]We will sometimes feel encumbered by the standard first-order language syntax; we will introduce to alternate notations, e.g. that better suggest computation, when we deem suitable.

will also let $\mathscr{L}\mathscr{T}$ denote the first order language with equality based on the same set of symbols on which $\mathscr{T}$ is written. We let $\mathscr{L}\mathscr{T}[f;p]$ denote the extension of $\mathscr{L}\mathscr{T}$ that includes the new function symbols of f and predicate symbols of p.

2.5.1. Special Formulae and Sentences. A formula in *prenex form* has the shape $Q_1x_1 \ldots\ Q_nx_n.\phi$, for a quantifier-free formula ϕ, with variables $x_1 \ldots\ x_n$ distinct and each $Q_i \in \{\forall, \exists\}$. In this case, we call $Q_1x_1 \ldots\ Q_nx_n$ the *prefix* of the formula, and ϕ the *matrix*. We say a sentence of the form $\exists x_1, \ldots, x_n.(\phi_1 \wedge \ldots \wedge \phi_k)$, with ϕ_i all positive literals, is a *primitive positive sentence*. The definitions [101] of Horn clause, Horn sentence, Horn theory, etc, are by now standard in the literature, e.g. Hodges [97], and we will not belabor them here. We will abbreviate a clause of the form $\forall x_1, \ldots, x_n.l_1 \vee \ldots \vee l_m$ by its matrix $l_1 \vee \ldots \vee l_m$, where $l_1 \ldots l_m$ are literals with free variables $\{x_1, \ldots, x_n\}$. Following Shepherdson [194, p. 363], we logically regard a query as a positive, existentially-closed sentence. Queries are often written in the computational syntax $? - L_1 \ldots L_n$. The negation of a query is a *goal*.

2.5.2. (Constraint) Logic Programs. Traditionally, a *(constraint)-logic program in* $\mathscr{L}$ is any finite set of definite $\mathscr{L}$-clauses P. Following the usual naming convention, we say a definite constraint logic program is a program with no negative programmed literals in any clause—the difference between the two is the addition of constraints in the bodies of clauses. We say P *defines* a programmed atom p if $p(\overrightarrow{t}^n)$ is the head of a definite clause in P, and that P *uses* an atom p if $\neg p(\overrightarrow{t}^n)$ is a disjunct in the body of some clause of P. We say the *definition of* $p(\overrightarrow{t}^n)$ *in* P, written $def_P(p(\overrightarrow{t}))$, is the set of clauses in P with head $p(\overrightarrow{u})$ for terms $\overrightarrow{u}$. Clark defines a logic program's completion in order to explicitly group clauses together this way.[4] This concept is similar to Deransart and Małuszyński's [48, p 103] $IF(p, P)$, representing the syntactic translation of *def*(p). For each p of arity n defined in P, Deransart and Małuszyński [48] construct $IF(p, P)$ with respect to variables $\overrightarrow{x}^n$ where for all x_i, $x_i \notin \bigcup_{c \in\ def_P(p(\overrightarrow{t}))} vars(\text{matrix}(c))$. The construction proceeds as follows. First, for

[4]We restrict well formed programs to those where every atom used in the program must be either a primitive constraint, or a programmed relation defined in the program.

each clause $c \in def_P(p(t_1, \ldots, t_n))$, define $E(c)$ as $\forall \overrightarrow{y} \neg(x_1 \equiv^? t_1) \vee \ldots \vee \neg(x_n \equiv^? t_n) \vee b$, where b is the body and $\overrightarrow{y}$ are the variables of the original clause c. We say $IF(p, P)$ is $\widetilde{\forall}(p(x_1, \ldots, x_n) \vee E(c_1) \vee \ldots \vee E(c_n))$, and that $IF(P)$ is the set of sentences $IF(p, P)$[5] for all p defined in P. The completion of a definite program is written $IFF(P)$ and just means the syntactic result of replacing $\leftarrow$ with $\leftrightarrow$ in $IF(P)$. Deransart and Małuszyński are concerned exclusively with normal logic programs; we call a *completed constraint-logic program* a completed program from clauses that permit constraints.

Fitting [59] and Naish and Søndergaard [159, 163] assume, as we do, that the program contains a single clause per predicate from the outset. Quoth Naish and Søndergaard,

> The `:-` in a single-clause definition thus tells us about both the truth and falsehood of instances of the head. Exactly how `:-` is best viewed has been the topic of much debate. . ..

We coin *relational constraint-logic program* as the name for how we write miniKanren programs—as universally closed bi-implications between a formula in the closure of atoms and constraints under conjunction, disjunction, and existential quantification, and an atom defining the relation, whose arguments include all and only free variables of the former. Every completed constraint-logic program is classically propositionally equivalent to a relational constraint-logic program.[6]

[5] This is different from Hogger [99, p 191], who uses $IF(P, R)$ to describe the set of queries that "fail infinitely" for a given program under a given computation rule.

[6] We might instead concern ourselves also with clauses' order and multiplicity in each predicate, and treat the program as equivalent to a *list* of clauses. Notions beyond each clause's mere existence matter in the general study of constructing miniKanren programs (e.g. Lu et al. [146]). However, we can solely consider clauses' existence or absence when defining relations for constraints, and for the limited purpose of this dissertation we can and will ignore these additional concerns entirely. We leave a precise miniKanren mechanism for the ordering and multiplicity of answers, and a formal characterization of the fairness of its search future work, (see Section 6.2 on page 136).

2.6. The Constraint-Logic Programming Scheme

We have not yet described constraints. These external CLP($\mathscr{C}$) languages must agree with $\mathscr{C}$ on the set $\mathscr{V}$ of variables, and must have the same pre-intepretation for $\mathscr{F}$. Sets of constraints differ from CLP language to CLP language. The designer for each language decides which formulae are constraints. A key benefit of CLP languages is that they share the same strong, tight connection between their logical, operational, and fix-point semantics as do standard LP languages.[7]. It would be tedious, however, to prove these interrelationships for each new language produced by instantiating the constraint domain. Instead, we should want to parameterize the proofs of these properties so that instantiating by a constraint domain of a certain form effectively instantiates the proofs of those interrelationships. By parameterizing out particulars of the constraint set, the CLP Scheme provides a way to reason generically about the languages' constraint systems. We should need at most to prove certain properties about the particular domain's constituent components. Jaffar and Lassez showed that if the domain components satisfy three properties (correspondence, satisfaction completeness of the theory, and solution compactness of the model) then almost all the fundamental theorems of LP can be extended to CLP using either the theory or the model. Much of the following recapitulates definitions from Jaffar and Lassez [106] and Jaffar et al.'s [111] "The Semantics of Constraint Logic Programs".

2.6.1. Constraint Domains. A *constraint domain* $\mathscr{C}$ is a 5-tuple of elements; we describe each element in turn as well as their required interrelationships.

- A *constraint domain signature* $\Sigma_{\mathscr{C}}$ contains the alphabets of the function symbols $\mathscr{F}$ and atomic constraint relation symbols $\mathscr{P}$ (with the symbols of $\mathscr{P}$ are disjoint from $\mathscr{F}$) together with function providing the arities of the elements from both sets.

[7] Even though typical logic programming syntax suppresses the equalities, Standard Prolog is itself an instance of the CLP scheme, and permits constraint logic programming in CLP(Tree) (see Marriott and Stuckey [152]).

- The *constraints* $\mathscr{L}_{\mathscr{C}}$ are some designated subset of L-formulae built over the set of primitive constraints $\mathscr{C}_{\mathscr{C}}$. These are the constraints over which we use constraint-logic programming in this particular CLP language. The CLP Scheme requires that $\mathscr{P}$ contain a nullary primitive constraint relation symbol `succeed` interpreted as an always-satisfied constraint, a nullary primitive constraint relation symbol `fail` interpreted as a never-satisfied constraint, and the binary constraint relation symbol `==` interpreted as equality. A CLP language's constraints must include all primitive constraints and will often include some formulae built with propositional connectives and quantifiers. However, for some choices $\mathscr{L}_{\mathscr{C}}$ is just equivalent to all subsets of $\mathscr{C}_{\mathscr{C}}$, those atomic constraints over terms. The CLP Scheme requires closure of $\mathscr{L}_{\mathscr{C}}$ under variable renamings ρ, conjunction, and existential quantification.
- A *computation domain* $\mathscr{D}_{\mathscr{C}}$ consists of the actual universe of values and an interpretation of the constraint predicate symbols $\mathscr{P}$ based on a pre-interpretation for the symbols of $\mathscr{F}$. The domain of computation—the carrier and the interpretation, is the intended model that gives the constraints an algebraic semantics.
- The *constraint theory* $\mathscr{T}_{\mathscr{C}}$ is the Σ-theory $\mathscr{T}_{\mathscr{C}}$ that describes the logical semantics of the constraints—$\mathscr{T}_{\mathscr{C}}$ axiomatizes some properties of $\mathscr{D}_{\mathscr{C}}$—so, that is, the theory has to describe the domain. This is where, for instance, `==` $\in \Sigma_{\mathscr{C}}$ gets interpreted as identity in $\mathscr{D}_{\mathscr{C}}$. At a minimum $\mathscr{T}_{\mathscr{C}}$ contains Clark's equality theory for this constraint domain.
- The *solver* $solv_{\mathscr{C}}$ checks the satisfiability of constraints of $\mathscr{C}$. Maher [148] points out that "solver" is generally a misnomer; such languages only actually *solve* the constraint in a limited sense. In particular, we do not define constraints' solved forms, nor do we reduce constraints to such a solved form.

  The CLP Scheme requires the solver not take variable names into account; for all renamings ρ, $solv_{\mathscr{C}}(c) = solv_{\mathscr{C}}(\rho(c))$. A *complete solver* is a total function from sets of admissible constraints to `{true,false}` that answers a *constraint*

satisfaction decision problem over the structure that is the domain of computation—that is, given an input constraint, check if the domain satisfies that constraint. In a corresponding domain, a complete solver is also *satisfaction complete*—that is, for every constraint c, the theory either entails that c is satisfiable, or entails that it is not satisfiable.

The theory $\mathscr{T}_{\mathscr{C}}$, solver $solv_{\mathscr{C}}$, and domain $\mathscr{D}_{\mathscr{C}}$, must *correspond*. First, they must be defined for the same language. Secondly, the $\Sigma_{\mathscr{C}}$-theory must model the domain. Finally, for any constraint c in the language of constraints, if the solver answers `false` then the theory entails the negation of its existential closure, and if the solver answers `true`, then the theory entails its existential closure. This last requirement says the solver must be no more powerful than the theory. When a solver is *exactly* as powerful as the theory, the solver is called *theory complete*: $solv_{\mathscr{C}}(c) = \mathit{false}$ iff $\mathscr{T}_{\mathscr{C}} \models \neg\bar{\exists}c$, and $solv_{\mathscr{C}}(c) = \mathit{true}$ iff $\mathscr{T}_{\mathscr{C}} \models \bar{\exists}c$.

Solution compactness is a requirement on the domain so that the negation of each constraint be represented by a possibly infinite set of constraints. That is, $\mathscr{D}_{\mathscr{C}} \models \bar{\forall}(\neg c \leftrightarrow \bigvee C)$, where C is some set of constraints in $\mathscr{L}_{\mathscr{C}}$. When evaluating a canonical logic program in with a complete solver, a solution compact domain guarantees that the finite failure set of the program agrees with the greatest fix point of the (CLP equivalent of the) van Emden-Kowalski immediate consequence function T. Jaffar and Lassez [106, Fig. 2] concisely describes the relationships between CLP programs' operational, logical, and algebraic semantics with respect to successful queries, and the additional requirements for these semantics to agree on finitely failing queries. We will not generally concern ourselves here with queries' finite failure.

2.7. miniKanren Constraint Domains

This section describes the schematized class of our miniKanren constraint languages. Our constraint microKanren framework in fact borrows a great deal conceptually from the CLP Scheme. We mirror Jaffar and Lassez by defining collections of CLP languages $CLP(\mathscr{C})$, with fixed logical syntax, in reference to a constraint domain $\mathscr{C}$. We parameterize the definition of families of constraint miniKanren languages by a constraint domain, and

we parameterize the expressions' meanings' over an ascription of meaning to the primitive constraints. We give such an ascription and show how these languages satisfy the CLP Scheme's requirements. We also separate predicates into the *user-defined* predicates and *built-in* constraints, and our languages employ built-in constraint solvers for the latter. These built-in predicates correspond to the constraints of CLP; the set always includes equality, for instance. These built-in predicates have fixed definitions that the CLP programmer cannot change, modify, or extend.

The differences, however, between constraint programming in Prolog-like CLP languages and these miniKanrens with constraints go beyond differences in concrete or abstract syntaxes. Unlike most logic programming languages, miniKanren languages do not provide general negation over atoms. Instead, a specified set of provided constraints, meeting certain criteria, form the class of specifically-permitted negated atoms. A more general CLP language like CLP($\mathscr{R}$) must separately provide the solver, theory and domain (the theory of real closed fields is a theory for the domain $\mathbb{R}$ and the a CLP($\mathscr{R}$) solver uses the simplex algorithm and Gauss-Jordan elimination [110]). We instead describe all three components at once, since our constraints are essentially negated logic programming predicates and these three parts of a constraint domain line up with independent but interrelated ways to define a logic [154]. Our class of constraint domains' intended models are algebras of finite trees, subjected to certain relational restrictions. The user provides an executable axiomatization of the theory of the domain, and from that we extract an implementation of a specialized solver. For all miniKanren constraint languages, a *constraint* is an existentially-closed conjunction of primitive admissible constraints from Π. Beyond the CLP scheme's requirements signature on the signature and interpretation (`succeed`, `fail`, `==`), miniKanren constraint systems also demand a symbol `=/=` to interpret as syntactic disequality, a negated form of `==`.

2.7.1. miniKanren Constraint Signatures. Our miniKanren constraints should have the following properties:

- Constraints must be decided by a complete solver.

- Constraints must be applicable over the entirety of the term language.
- Constraints must be "all intermixable"—always applicable in combination.
- Constraints have to hold modulo some background equality theory $\mathscr{T}_{\mathscr{C}}$ of first-order syntactic equality.
- We need the constraint domains to be "cumulative"—adding new forms of primitive constraints to the language "works".

Each constraint domain that we construct is a term algebra, possibly extended by a few function symbols with fixed non-term interpretations, and our constraints are the existential closure of conjunctions of a designated set of primitive constraints.

2.8. Negative Constraints

This section explores an important concept for our constraint systems: the independence of negated constraints [136, 139]. We choose to view the constraints' definitions and their interactions as fixed parts of a distinct CLP language. Under this view, the primitive constraints are a set of special primitives, and constraints are the closure of this set under conjunction and existential quantification. This is, however, just one point of view.

We can instead view constraint microKanren programs as programs written in a single, flat LP language but written in two different phases and in which it is only valid to use these "constraint" things in negative literals. Since we will not have any recursion through the negative portions of our negative clauses, what we have are equivalent to stratified logic programs. In this view these are not just stratified programs, but **staged** program definitions.

The independence of negative constraints is a commonly recurring property in logic programming [138] and language implementers have made important use of this property in constructing solvers (e.g. [40, 139]). This property in some sense generalizes the *strong compactness* over equations (see Lassez [135, §6] and Lassez et al. [137, pg 80]) to constraints more generally. The general independence of negative constraints describes a property of some set $p \in P$ we deem the "atomic positive constraints", and some "negatable constraints" $q \in Q$ whose negations we denote q', under some consequence relation $\vDash$. We take the

sequent $\{p_1, \ldots, p_n\} \vDash \{q_1, \ldots, q_m\}$ to mean that the finite conjunction $p_1 \wedge \ldots \wedge p_n$ implies the finite disjunction $q_1 \wedge \ldots \wedge q_m$. The (un-)negation of a negatable constraint permits it to be moved from one side of the sequent to the other. Maher [Definition 10 148, p 316] describes the negative constraints as independent if $\{p_1, \ldots, p_n\} \vDash \{q_1, \ldots, q_m\}$ implies $\{p_1, \ldots, p_n\} \vDash \{q_i\}$ for some $0 \leq i \leq m$. As a consequence of the previous two facts, we can know a set $\{p_1, \ldots, p_n, q'_1, \ldots, q'_m\}$ is consistent provided each set $\{p_1, \ldots, p_n, q'_i\}$ is consistent, once again for $0 \leq i \leq m$. Full independence is not a common property—equality constraints for instance, are not independent. Maher's [147] "A Logic Programming View of CLP", § 4 describes this property and connects it to earlier generalizations.

General miniKanren constraints are *not* themselves independent, even modulo the primitive equality constraints. Each constraint "bucket"—an homogeneous set in a family of sets of negative atomic constraints—is independent. There are only finitely many buckets, so after solving the equalities and applying the substitution, checking the satisfiability of a constraint only requires checking groups of at most n constraints at a time. Each set of the indexed family is n-constraint bucket-wise independent in the presence of the "full equational implication". That means that, viz. all the equations and implied equations, we can test the consistency of admissible primitive constraints by testing each possible (up-to)-n tuple independently. Because we need check at most only n primitive constraint atoms at a time, we can call these nearly independent, or *n-independent* constraints.

Internally, we write each constraint relation by a list of definite Horn clauses; the constraints we express are negated atoms defined with these predicates. Our n-independence results in part from the properties of Horn clause theories. Strict Horn clause theories are necessarily consistent; if constraint definitions came as strict Horn clauses, we would know that the resulting theory is consistent. Further, by a well-known fact (see e.g. Hodges [96, § 5]) consistent Horn clause theories always have an initial model. This is a frequently sought-after benefit [166]:

> Initial algebra semantics [GTW 78] is, probably, the most popular method for giving semantics to algebraic specifications. Several reasons justify this popularity, among them the methodological appeal of the "closed world assumption" [GoMe 83], the simplicity and power of the technical constructions used and the power of the associated methods and tools [HuOp 80].

These properties will prove important for our solvers.

Each homogeneous bucket of negative constraints needs to have the independence property. Our constraint domains' theories are close to, but not necessarily, strict Horn clause theories. By Makowsky [151, Thm. 5.9], every theory that admits an initial model is equivalent to a $\forall\exists$-Horn theory, and furthermore, any finite theory T that admits an initial model is equivalent to a finite $\forall\exists$-Horn theory.

Not every $\forall\exists$-Horn theory, however, admits an initial model. Furthermore, we do not present our constraint theories by $\forall\exists$ Horn theories explicitly. We do present the constraint relations themselves by strict Horn clause definitions. Instead of explicitly writing the $\forall\exists$ Horn sentences, we introduce a finite number of computable functions for the definitions of the constraint relations. These interpreted functions kind of "Skolemize" away the $\exists$ quantifiers, and this class fits precisely with the theories admitting initial models. Makowsky [151, Thm. 5.9] characterizes the theories admitting initial models as precisely the partially-functional $\forall\exists$-Horn theories. By Makowsky's [151, §6] result, elements in sets of negated atomic formulae—such as `=/=` formulae—are independent with respect to a partially-functional $\forall\exists$ Horn theory.

This suggests how we can combine the different classes of atomic negative constraints. What remain are the heterogeneous constraint set failures; these describe the conditions, modulo equality, for which the domains fail to be initial, even with the addition of interpreted functions.

Much of this chapter's general background comes from Doets [51], Downward [52], and Lloyd [143]. For our discussion of logic programming and its semantics, we consulted Apt and Van Emden [10], Lloyd [143], and van Emden and Kowalski [211]. We consulted Clark [34] as well as Lloyd's [143] *Foundations of Logic Programming* for the material on program completion. For a history of negation and LP, see also Apt and Bol [9], Kunen [132], and Naish and Søndergaard [163]. See Clark [33] for an historical development of logic programming schemes extending from Kowalski's [127] approach that culminates in the CLP Scheme. For more background and related literature, see also Hogger [99, Themes 2-4]. We suggest Kriwaczek [131], Lassez [134], and Wallace [219] for introductions to constraint logic programming. The CLP Scheme has since been generalized in different directions; these include Höhfeld and Smolka's [100] and those approaches described by Van Hentenryck [212]. Jaffar and Maher [110] survey the state of constraint logic programming in 1994, and Rossi [184] gives a later survey focused on its applications. The reader could consult Gabbay et al's recent volume *Computational Logic* in the *Handbook of the History of Logic* series [197] for a more recent work surveying some of the breadth of the field.

Chapter 3 Semantics of microKanren Constraints

In this chapter, we describe the specification and construction of miniKanren constraint domains. Constraint microKanren generates CLP languages whose constraints range over the particular domain of microKanren terms. We saw in Section 2.7 that the differences between constraint programming in Prolog-like CLP languages and miniKanrens with constraints go beyond differences in their syntaxes. Rather than providing general negation over atoms like most logic programming languages, each constraint miniKanren language instead permits a particular, specific, class of negated atoms; this class is the set of atomic constraints built into that particular language. These classes include the symbolic constraints used in miniKanren programming. The constraint language specification picks out the particular class, and the specification language ensures that every collection of allowed constraints meets certain correctness criteria.

First, we extract from a part of the specification an underlying, basic constraint domain. This domain has an intended model that defines the elements, the operations, and the relations on that structure. This intended model is an algebraic semantics, and from this intended model we also extract a corresponding theory. Furthermore, we require that the specification for this domain also provides an implementation of a solver. We defer the construction of such constraint systems to Section 3.2, and we exhibit programs that use constraint systems built from this infrastructure in Chapter 4.

3.1. Making a Domain

A specification begins with an effective encoding #. By #, we mean to say fixing a computable function, called the *encoding function*, that maps from a set of host language programs to μ-recursive functions and from a set of host language data structures into

$\mathbb{N}$, so that the translation respects the behavior of the host's execution of programs on its data. An embedded constraint domain specification describes an embedding into some programming language, but our specifications are otherwise agnostic to the *particular* embedding language.

3.1.1. #-based term-partition specification. The specification begins in earnest with $TP^{\#}$, a *#-based term-partition specification*. The #-based term-partition specification $TP^{\#}$ is a 4-tuple comprised of elements we explain in turn. When S is a set of elements from the encoding's domain, we will use $\#[S]$ as shorthand for $\{n \in \mathbb{N} \mid (\exists s \in S)[\#(s) = n]\}$, the image of S under the encoding. We will not explicitly #-encode sets themselves as objects, so this notation is unambiguous. We use χ_N for the *characteristic function of* N, for $N \subseteq \mathbb{N}$. With two sets $S \subseteq \mathscr{T}$ of elements from the encoding's domain, we use $\chi_{\#[S]} \restriction_{\#[T]}$ for the characteristic function on $\#[S]$, restricted to $\#[\mathscr{T}]$. We use $\chi_S^{-\#} : \mathscr{T}$ as shorthand for a #-program that implements that characteristic function. To emphasize the set S rather than the function χ, we may use $S_{p:\mathscr{T}}$ when $\#(p) = \chi_{\#[S]} \restriction_{\#[\mathscr{T}]}$.

We write $\mathscr{T}(\Sigma, X)$ for the Σ-terms freely generated over X, and we say X are the generators of $\mathscr{T}(\Sigma, X)$. When the precise contents of Σ (and perhaps X) are unimportant or the reader can infer them from context, we will use $\mathscr{T}$ for $\mathscr{T}(\Sigma, X)$ and $\mathscr{G}$ for $\mathscr{T}(\Sigma, \emptyset)$, referring to the latter as the set of *ground terms*. For given Σ, we define the X-parameterized partial function $pfs_X : \mathscr{T}(\Sigma, X) \nrightarrow \Sigma$ that determines the *primary function symbol* of a term in $\mathscr{T}(\Sigma, X)$. We define $pfs_X(\sigma(t_0, \ldots, t_n)) = \sigma$ for $\sigma \in \Sigma$ and t_0 through t_n in $\mathscr{T}(\Sigma, X)$.

#-based Term-Partition Specification
$TP^{\#} \quad = \quad \langle \mathsf{var?}, F^+, P_=, P_< \rangle$

where:

- $\mathsf{var?} \triangleq \chi_X^{-\#} : \mathscr{T}(\Sigma, X)$
- $F^+ \triangleq \{\langle f, n \rangle \mid f \in \mathscr{F} \subset \Sigma \wedge n \in \mathbb{N}^+\}$
- $P_= \triangleq \{p \in P \mid |pfs_X^{-1}[\Sigma_p]| = \omega\}$
- $P_< \triangleq \{p \in P \mid |pfs_X^{-1}[\Sigma_p]| < \omega\}$

and subject to the requirements:

- The family of sets of constructors $(\Sigma_p)_{p\in P}$ indexed by the set of programs $P \triangleq P_= \cup P_<$ partitions Σ.
- $|\mathscr{G}| = \omega$ (which implies $\mathscr{C} \neq \emptyset$, and if $|\mathscr{C}| < \omega$, further implies $\mathscr{F} \neq \emptyset$).

The first element of $TP^{\#}$ is a #-program that codes for a function (with its domain restricted to the #-encodings of $\mathscr{T}(\Sigma, X)$) characterizing the #-encodings of a set X. The remaining three components of this specification must all be finite sets, since we finitely enumerate their elements. The set $pfs_{\emptyset}^{-1}[\Sigma_p] = G_p$, and the set $pfs_X^{-1}[\Sigma_p] = T_p$. The family $(G_p)_{p\in P}$ partitions $\mathscr{G}$, and the family of sets $(T_p)_{p\in P}$ partitions $\mathscr{T} \setminus X$. These two partitions correspond to one another.

We call $\mathscr{F}$ the *posary* (positive arity) operators. We call the remainder $\mathscr{C} \triangleq \Sigma \setminus \mathscr{F}$ the *nullary operators*, or constants. This description doesn't fully specify a **particular** set $\mathscr{C}$. Instead, it leaves open lots and lots of possible choices for infinite sets $\mathscr{C}$ (and thus Σ). Any of those choices would be correct so long as they contain all the constants of the constraint or constraints for which we're using this domain/solver. There are certainly *largest* sets for which we can define them, the largest $\mathscr{C}$ that respects the encoding. Making a particular choice isn't necessarily more correct than remaining generic to any acceptable choice (e.g. any infinite set of Racket symbols that contains all the symbols used in the program, and so forth for strings.). But if there *is* a best choice, then the maximal sets accepted by the #-programs is that choice. From this description we construct the function *arity* as follows:

$$arity(\sigma) \begin{cases} n & \text{if } \langle \sigma, n \rangle \in F^+ \\ 0 & \sigma \in \mathscr{C} \end{cases}$$

We subscript the subset of programs $P_=$ (resp. $P_<$) as such because each member of that subset accepts a countably infinite (finite) term-partition block, that is, of cardinality equal to (less than) ω. Since the infinite set of ground terms is freely generated, and since the term-partition blocks together all terms of the same primary function symbol, any finite term-partition block must consist entirely of constants.

The elementary specification $TP^{\#}$ suggests a particular topological space on $\mathscr{G}$, $(\mathscr{G}, \tau)$. The topology τ is the collection of all co-finite sets formed from elements of $\mathscr{G}$: $(\mathscr{G}, \{A \mid A = \emptyset \vee |\mathscr{G} \setminus A| < \omega\})$. The co-finite sets, together with $\emptyset$, are the open sets of the topology (these are possible domains of variables "open" to the CLP programmer). An infinite $\mathscr{G}$ guarantees all co-finite sets are infinite.

Let # be the "Racket encoding", and we can discuss an example with Racket constructors, Racket data, and Racket programs. This encoding is usually implicit in actual embedded miniKanren implementations.

A typical miniKanren instance of a 4-tuple TP^{Racket} would be:

```
<natural?, (<cons,2>), (boolean? null?), (symbol? string? pair?)>
```

We could represent these data *in* Racket, but it's not necessary to do so just because we use the Racket encoding. This information should be language independent in that, under some other encoding with corresponding data-types (an embedding into some other programming language) this tuple should specify a term language and some primitive programs over that term language, too.

Typical for miniKanrens including this example, the first element of $TP^{\#}$ is `natural?`, used as `var?` in the implementation. We take the set of Racket natural numerals, $\lceil \mathbb{N} \rceil$ as the generators X. These are distinct from the set $\mathbb{N}$, the co-domain of the encoding # and the setting against which we define characteristic functions and later discuss computability. We must actually fix a bijection $\texttt{var} : \mathbb{N} \to X$ enumerating the set X accepted by `var?`.

The Racket programs `null?`, `boolean?`, `symbol?`, `string?`, and `pair?` code to characteristic functions.[1] Continuing this example, $\mathscr{F} = \{\texttt{cons}\}$ and $\mathscr{C} = \{\texttt{()}\} \cup \{\texttt{\#t}, \texttt{\#f}\} \cup$ Racket symbols $\cup$ Racket strings. We treat both the Racket symbols and Racket strings as atomic constants.[2]

[1] Special versions of those above programs built to error on any input not in our particular term language over $\lceil \mathbb{N} \rceil$ would perhaps instead be better implementations of the restricted characteristic functions.

[2] One should not confuse Racket's strings with the traditional automata theory definition of strings as the set Σ^* over an alphabet Σ. This will not be our convention, and we will not use Σ in this way.

3.1.2. Combinatoric Complexity Interlude. When solving a constraint for a variable with a domain restricted to some finite set, we may find through an exhaustive analysis by cases that the constraint is unsatisfiable. Exhaustive satisfiability testing across finite domains can be complex. Much constraint systems research stemming from operations research focuses on finding efficient, specialized solutions for certain classes of these problems. However, constraint problems of this sort are intractable in the general case. An overarching design goal for our constraint systems was to preclude such complex search techniques in the solver. To avoid this kind of search, we exclude constraints for which any variables' values come from a non-trivially finite domain.[3]

The predicates on the above structures come from an arbitrary partition. These predicates may or may not have the desired domain property in these structures. There are infinitely-many natural numbers, for instance, and only two booleans. Furthermore, there will be some sets that are not in and of themselves a block of a partition, but for which we will still permit treating membership as a constraint.

In the following, we construct larger, more complex constraint domains based on such a small structure as described in the preceding. In doing so, we will extend the constraint domain's signature and its other corresponding components. When designing such an extension, the constraint language designer must select certain families of sets that we guarantee avoid any combinatoric explosion during solving. We will see this in Section 3.1.6 on page 45.

3.1.3. $TP^{\#}$-based Primitive Predicate Specification. The *$TP^{\#}$-based primitive predicate specification* is a special kind of family of sets $\mathscr{S}$ over the particular set of programs P given from $TP^{\#}$. This family of sets of programs relates to an extension of τ, and will also closely relate to the elementary predicates that—along with the equality predicate—the constraint writer uses for defining relations. We use $\mathscr{S}^{\cup}$ for the union closure of $\mathscr{S}$, and we will use S^c for the P-complement of a set $S \subseteq P$. By convention, we will use S for

[3]This is an instance of the *Zero one infinity rule* `wikipedia.org/wiki/Zero_one_infinity_rule`.

an element of $\mathscr{S}$ and SS for an element of $\mathscr{S}^{\cup}$. For each S, let $\dot{S}$ be the name of a new predicate (the *primitive predicates*) that holds for exactly those terms for which a $p \in S$ accepts.

$TP^{\#}$-based Primitive Predicate Specification
$PP^{TP^{\#}} \quad = \quad \mathscr{S}$

such that:

- For each $SS \in \mathscr{S}^{\cup}$, $P_{=} \not\subseteq SS$.

The family $\mathscr{S}$ is the "generator" for the union closure $\mathscr{S}^{\cup}$. The above restriction guarantees that $\{t \in \mathscr{T} \mid \dot{S}(t)\}$ is co-infinite. Indeed, this guarantees the stronger condition that $\bigcup(\mathscr{G}_p)_{p \in SS}$ is co-infinite for all $SS \in \mathscr{S}^{\cup}$. Together with τ, the collection of sets $\{\bigcup(\mathscr{G}_p)_{p \in SS^c} \mid SS \in \mathscr{S}^{\cup}\}$ form the basis for an extension τ' of τ. In our system, the constraint writer defines a constraint via membership in the complement of some computable set of terms or tuples of terms. We will say more about these in Section 3.1.4.

To continue the Racket example of Section 3.1.1, we select the following family of sets over the programs P from Section 3.1.1. One can verify the above requirement holds for this set. For merely building the structure, the precise names are unimportant and the $\dot{S}$ suffice. For a programmer however, names matter, and so in comments we include suggestive names that we might instead provide (that we will in fact use in subsequent sections).

```
{ {boolean?, null?, pair?, string?}       ;; non-symbol?
  {boolean?, symbol?, string?}            ;; non-list-constant?
  {boolean?, null?, symbol?, string?}     ;; non-pair?
  {pair?}                                 ;; pairr?
  {boolean?, null?, pair?, symbol?}   }   ;; non-string?
```

Together with the binary term-equality predicate and the special trivially-true value `true` our system always includes, these primitive predicates—as named and defined above—underlie the definitions of constraint predicates.

3.1.4. $PP^{TP^{\#}}$-based Predicate Definitions. We specify the general predicates via a set of Horn clauses in implicational form. Rather than the more general *extended clauses*, we restrict the input to definite Horn clauses and allow only positive atoms in clauses' bodies.

The relation symbols at the heads of these clauses are new symbols. The clauses in the specification whose heads have the same relation symbol define collectively the behavior of a predicate. The set of all these clauses then define all general predicates for the system. Clauses' bodies may refer to the predicate it defines or to some other predicate (via relation symbols at the heads), and may also include atoms with over the provided primitives, as well as binary `equal?` and the atomic `true`. The resulting predicates need to be total over queries, and must possess Shepherdson's finite tree property for all atomic queries; the class of such programs is sufficiently expressive.

More precisely, define the k-set of $Pr^{PP^{TP^{\#}}}$ of clauses with respect to the previously-defined $PP^{TP^{\#}}$ as follows:

$$\frac{PP^{TP^{\#}}\text{-based Predicate Specification}}{Pr^{PP^{TP^{\#}}} \quad = \quad \{c_0, \ldots, c_k\}}$$

where:

- Each clause c_i has the form $h_i \leftarrow b_{i_1}, \ldots, b_{i_m}$.
- Each h_i is some $r_i(t^+_{i_1}, \ldots, t^+_{i_n})$.
- Each $t^+_{i_j} \in \mathscr{T}(\Sigma, Z_i)$, and Z_i the least such required generating set.
- The set $R = \bigcup_{i \in 0 \ldots k}\{r_i\}$, where R and each of $(Z_i)_{\{0 \ldots k\}}$ are finite, mutually disjoint sets distinct from any aforementioned sets.
- And finally then, each b_{i_j} is either `true`, or `(equal?` $t^+_{i_{k_1}}$ $t^+_{i_{k_2}}$`)` or $p_{i_j}(t^+_{i_{j_1}}, \ldots, t^+_{i_{j_n}})$ with each $t^+_{i_{k_j}} \in \mathscr{T}(\Sigma, Z_i)$, and $p_{i_j} \in R$ or p_{i_j} is one of the primitive predicate symbols from $PP^{TP^{\#}}$.

We continue the example from Section 3.1.3, defining a singleton predicate set and with $R = \{$`mem?`$\}$. To avoid confusion between the constants $\mathscr{C}$ and the elements of each Z_i, we choose single capital letters and "_" as the elements of each Z_i. The constraint writer needn't consider full unification, the set X, or indeed $\mathscr{T} \setminus \mathscr{G}$ when defining these relations' heads. Instead he need only write clauses to match against the ground terms of $\mathscr{G}$. We define the `mem?` relation on the domain with a recursive Racket implementation.

```
(mem? X X)           ←  true
(mem? X (cons _ Z))  ←  (mem? X Z)
(mem? X (cons Y _))  ←  (mem? X Y)
```

3.1.5. $TP^{\#}$-based Term Function Specification. Here, we here introduce some new, special function symbols with non-trivial (non-term-model) interpretations. As such here we are no longer interpreting into an initial model. However, we don't ever use functions inside the above Horn clause-based predicate definitions, nor will the constraint programmer have direct access to them. Further, when we use these auxiliary, interpreted function symbols in the following sections, we use them only around terms (or other such function calls) in negative atomic constraint definitions and in specifying rules for the solver. Such usages transport portions of this entire enterprise back *into* a term model.

Furthermore, we fix a syntax for *defining* such a function against a term language. There is a fixed grammar through which we write them, roughly speaking, against the signature of the F-algebra. We list what to do on the terms of the posary function symbols, and we use the primitive predicates to match all of the constants[4].

Each such function $\hat{f}$ we write will describe a structure homomorphism on $\mathscr{G}$ defined in $TP^{\#}$. In our system we express a structure homomorphism via some function $\hat{f}$, computable and totally defined on $\mathscr{G}$, into some analogous substructure on $\mathscr{G}$ (that is, $\hat{f}[\mathscr{G}] \subseteq \mathscr{G}$). Such a function describes the structure homomorphism. Since the mapping needs to preserve the relations defined by $TP^{\#}$-predicate structures, extend each relation exactly as far as the image of the domain under the given function. Since the function $\hat{f}$ will represent a morphism from the ground term structure $\mathscr{G}$, $\hat{f}$ will extend to a comparable total function $\hat{f}'$ on $\mathscr{T}(\Sigma, X)$ by subsequently describing the intended behavior on X.[5]

[4]If some primitive predicate p matches only terms with posary function symbols, then matching the posary function symbols one-by-one makes this primitive predicate p redundant.

[5]In point of fact, in our system $\hat{f}'$ will always behave like the identity on X. We will discuss the implementation of this behavior in Section 3.2.

$TP^{\#}$-based Term Function Specification		
$TF^{TP^{\#}}$	$=$	$\{\hat{f} \mid \hat{f}[\mathscr{G}] \subseteq \mathscr{G}\}$

We continue the example from Section 3.1.3 by defining one such function: `cdr*` returns the rightmost leaf of each term, for each term viewed as a tree. The function `cdr*` of this example maps from every element of the F-algebra to a member of its generating set.

$$cdr*(\texttt{X}) \begin{cases} \texttt{X} & \text{if } \texttt{(non-pair? X)} \\ cdr*(\texttt{Z}) & \text{if } \texttt{(pairr? X)} \text{ and } \texttt{X} = \texttt{(cons \_ Z)} \end{cases}$$

As we will see, we will *only* write a function invocation like `cdr*(` $\cdot$ `)` around a term, and only in either the body of a negative constraint definition or in the conditions of solver rules. With this collection of functions in hand, we next specify a grammatical structure for the full definitions of atomic independent negative constraints, based on the previously defined general predicates and these function definitions.

3.1.6. $(Pr^{PP^{TP^{\#}}}, TF^{TP^{\#}})$-based Negative Atomic Constraint Definitions. We specify the independent negative atomic constraints over the aforementioned set of general relations over the term algebras and the set of structural functions. Members of the constraint Kanren language family are parameterized by classes of *atomic constraints*. All the solver-internal infrastructure we have built thus far in this chapter is opaque to the constraint logic programmer. These atomic constraints (along with the two atomic goals) are the smallest program units against which a programmer can execute queries. The CLP literature refers to these as *negative constraints* those constraints defined as the negation of some atom.

Every negative atomic constraint definition defines a class of negative atomic constraints constructed with a new n-place programming-language relation symbol $\overline{r}$ and an n-tuple of the terms in $\mathscr{T}(\Sigma, X)$. Each negative atomic constraint definition defines the meaning of its family of $\overline{r}$-constraints as the negation of some relation $r \in R$ over n tuples of $\hat{f}_{i_0}(\ldots(\hat{f}_{i_k}(t_i)))$, with each $\hat{f}_{i_j}$ coming from the set defined in Section 3.1.5. These definitions

connect the internal infrastructure of the particular to the precise constraint logic programming language with which the solver is associated. We use the term $\overline{r}$-*atomic constraints* to refer to the set of negative atomic constraints with relation symbol $\overline{r}$, and we will also use this terminology for subsets thereof. We will also call any subset of such a set a *homogeneous set* of negative atomic constraints. In addition to the provided negative atomic constraint definitions, all of our constraint systems will, per force, include disequality constraints via the equivalent of `(=/= A D)` $\triangleq \neg$`(equal? A D)` for some new symbol `=/=`. The family of sets of $\overline{r}$-atomic constraints, for each new relation symbol $\overline{r}$, forms a family of negative atomic constraints indexed by the set of *constraint relation symbols* $\overline{R}$; this family partitions the entire set of negative atomic constraints.

We have been so particular about constructing the constraint domain to ensure homogeneous sets of negative atomic constraints have the independence of negated constraints property. This terminology can be somewhat confusing in our context, since we have called the constraint logic programming language's (positive literal) atomic constraints "negative atomic constraints". We named them such because they take their meaning as the negation of some positively-specified clausal property internal to the solver.

In our particular case, we take $\models$ as logical consequence. For every $\overline{r}$, every finite homogeneous set of atomic $\overline{r}$-constraints Q, every $\overline{r}$-atomic constraint has the independence property. This means that given any other set of atomic positive constraints P of our system, $P \cup \{q'_1, \ldots, q'_m\}$ is consistent iff for each $0 \leq i \leq m$, $P \not\models \{q_i\}$. These include the disequality constraints introduced by the system. Makowsky [151, §6], Colmerauer [40], and Lassez et al. [137, §6] each gave special study to the independence of disequality constraints.

$(Pr^{PP^{TP^{\#}}}, TF^{TP^{\#}})$-based Negative Atomic Constraint Definitions		
$AC^{(Pr^{PP^{TP^{\#}}}, TF^{TP^{\#}})}$	$=$	$\{(\overline{r}\ \ldots v_i \ldots) \triangleq \neg r(\ldots, \hat{f}_{i_0}(\ldots(\hat{f}_{i_k}(v_i))), \ldots), \ldots\}$

Here we exhibit a collection of atomic constraint formulae definitions in terms of negations of relations as defined earlier in this example. The reader can verify that sets of constraints from each class below, as well as disequality constraints, have the independence property when evaluated in the $\mathscr{T}(\Sigma, X)$.

The `absento` constraint found in several miniKanren implementations is the negation of a subterm relation. We implement this via a preorder given by the subterm ordering like that described by, e.g. Tulipani [208] and Venkataraman [216]. The first implementations of negated subterm constraints in miniKanren come from the author and Dan Friedman and first published by Byrd et al. [27]. Several recent miniKanren implementations contain a `listo` and/or `not-pairo` constraint, such as that of Hemann and Friedman [85].

```
(listo X)       ≜  ¬(non-list-constant? (cdr* X))
(symbolo Y)     ≜  ¬(non-symbol? Y)
(stringo Y)     ≜  ¬(non-string? Y)
(not-pairo NP)  ≜  ¬(pairr? NP)
(absento A D)   ≜  ¬(mem? A D)
```

Most sets of negative atomic constraints we encounter during the execution of a logic program are, however, heterogeneous. Even though members of the family of atomic constraints are $\overline{r}$-independent, a pair of atomic constraints $\langle \phi, \psi \rangle$ from different indexes can still imply something more than the sum of the implications of constraints ϕ and ψ separately. We call this situation "inter-family interaction". In the next section, we will introduce one consequence of inter-family interaction among heterogeneous sets of negative atomic constraints.

3.1.7. $AC^{(Pr^{PP^{TP^{\#}}}, TF^{TP^{\#}})}$**-based Constraint Interaction Definitions.** Homogeneous sets of atomic $\overline{r}$-constraints are independent. However, negative atomic constraints from heterogeneous sets can interact. The only interesting or meaningful interactions are subsets of inconsistent finite sets, for which the homogeneous sets of atomic $\overline{r}$-constraints are not separately subsets of inconsistent finite sets.

Importantly, even (finite collections of conjunctions of constraints) conjunctions of constraints from different indices cannot induce a non-trivial finite domain for any variable. By the requirement we placed on $\mathscr{S}$ as the basis for an appropriate topology extension. The complement of the sets of terms described by conjunctions of primitive predicates must remain infinite, and no matter how many (finitely-many) elements we exclude, there

remains an infinite set. So the interactions can occur significantly in only one of the two trivially-finite domain forms, and these are all and only the situations we must further consider.

Applying that substitution `s` across terms reduces equality to syntactic equality in a term algebra. The independence of negative constraints means here that every inconsistent finite set of constraints, heterogeneous or otherwise, has an inconsistent subset with at most one element of any $\overline{r}$.We ensure this approach to solving constraints works by forbidding any non-trivial finite domain constraints in the constraint language.

- constraints of a heterogeneous collection interact to cause failure (where some variable has 0 possible values)
- constraints of a heterogeneous collection interact to *define* a domain element (where some variable has exactly 1 possible value)

We know that in the special induced equalities above, since one side will always be a constant, the order of application of these rules cannot matter. Thus, there is no danger that some rule will fail to fire on a constant, or that we would need a fix-point algorithm to "catch" it.

$$\frac{AC^{\langle Pr^{PP^{TP^{\#}}}, TF^{TP^{\#}}\rangle}\text{-based Constraint Interaction Specification}}{CI^{AC^{\langle Pr^{PP^{TP^{\#}}}, TF^{TP^{\#}}\rangle}} \quad \langle \text{conditional equalities}, \text{failure rules}\rangle}$$

We format them as constrained rewrite rules a la Kirchner et al. [117], or like propagation forms of constraint-handling rules (CHR). The $\rightarrow$ should suggest CHR-like behavior, and we intend $\Rightarrow$ as assignment. From left to right, we read these rules to say: "for all of the antecedents, produce the consequent, when the conditions hold". These are, however, restricted versions of the general forms of such rules. These failure conditions we write here capture failures "at the limit" that aren't yet failed for finite approximations.

```
(stringo st),(symbolo y)  →  ⊥          |  (equal? st y)
   (listo l),(symbolo y)  →  ⊥          |  (equal? (cdr* l) y)
  (listo l),(stringo st)  →  ⊥          |  (equal? (cdr* l) st)
 (listo l),(absento p r)  →  ⊥          |  (nulll? p),(mem? (cdr* l) r)
(listo l),(not-pairo np)  →  np ⇒ '()   |  (equal? (cdr* l) np)
```

At most $|\overline{R}|$-many constraints are ever required to check at once for a failure. We never need two atomic constraints of the same set in there, because the homogeneous sets are independent.

3.1.8. Equality Constraints. Atomic equality constraints (be they written explicitly or implicit in the language's syntax) are critical for logic programming, and constraint logic programming does not differ in this respect. However, unlike our negative atomic constraints, these atomic equality constraints are *not* independent of one another. The inherited recursive structural equality on terms is alone insufficient for atomic equality constraints, because of the presence of variables in terms of $\mathscr{T}(\Sigma, X)$. Instead we solve collections of these constraints using unification, and by quotienting under the result.

If collectively the equations are mutually compatible, we can treat those equations as axioms. There will be no real universal equalities here, because the term-variables are just other constants. If collectively the equations are mutually compatible, we can treat the equivalent set in solved form as rewrite rules. By substitution, which is to say rewriting under those rules, we bring everything into a question of syntactic equality.

All of our constraint systems will, per force, include binary atomic equality constraints over terms for some new symbol `==`. We implement these constraints over $\mathscr{T}$ with first order syntactic unification.

3.1.9. $CI^{AC^{(Pr^{PP^{TP^{\#}}}, TF^{TP^{\#}})}}$-based Constraints, and Solver. Our full language of $CI^{AC^{(Pr^{PP^{TP^{\#}}}, TF^{TP^{\#}})}}$-based constraints is determined by the $\overline{R} \cup \{\texttt{==}\}$-indexed family of atomic negative constraints plus equations. A *constraint* is an $\overline{R} \cup \{\texttt{==}\}$-indexed family of homogeneous sets each from the family of sets over $(\overline{r}\text{-atomic constraints})_{\overline{r} \in \overline{R} \cup \{\texttt{==}\}}$. We will call this class of constraints $\mathscr{C}_{\mathscr{C}}$.

Provided the negative atomic constraints of each primitive constraint identifier in the signature are independent of each other, and that the negative atomic constraints (that is, excluding equality constraints) of different identifiers are n-independent of one another.

Logically, the class $\mathscr{L}_C$ of constraints for this miniKanren constraint language is the set of all existentially-closed conjunctions of atoms with either `==` as the predicate symbol or a negated atom with `==` or one of the other added predicate symbols.

We now describe a non-deterministic algorithm for solving a constraint C. First solve the class of atomic `==` constraints (in any order), and generating a substitution if possible. Then check every possibly instance in the constraint of any defining formulae (in any order), substituting through and extending the substitution, if possible. Then, in any order and potentially in parallel, check if any instance of any of the failure formulae hold. Our solver's "motto" would be: if it's not known to be impossible, then it must be possible. We provide this algorithm in Algorithm 1.

Algorithm 1 Solving constraints in independent negative constraint domains

---

Precondition: I are conditional equalities
Precondition: F are failure rules

function SOLVE(C) ▷ $C \in \mathscr{C}_{\mathscr{C}}$ is a $\overline{R} \cup \{\texttt{==}\}$-indexed family of sets
  if $\sigma \leftarrow$ UNIFY($C.\texttt{==}$) **then** ▷ if UNIFY succeeds, σ is the *mgu* of $C.\texttt{==}$
    for all $\pi_{j_0}(\overline{d_{j_0}}) \cdots \pi_{j_k}(\overline{d_{j_k}}) \rightarrow rr \mid t? \in I$ **do**
      ▷ $\pi_{j_i}(\overline{d_{j_i}})$ a constraint atom pattern
      ▷ $\forall (j_i, j_{i+1}) j_i \neq j_{i+1}$
      ▷ $0 \leq j_i < |I|$
      for all $\langle c_0, \ldots, c_k \rangle \in \langle C.\pi_{j_0}, \ldots, C.\pi_{j_k} \rangle$ **do**
        $\theta \leftarrow$ MATCH($\langle \pi_{j_0}(\overline{d_{j_0}}), \ldots, \pi_{j_k}(\overline{d_{j_k}}) \rangle, \langle c_0, \ldots, c_k \rangle$)
        if $((t?)\theta)\sigma$ **then**
          if $\sigma \leftarrow \sigma \circ (rr)\theta$ **then** ▷ The composition can fail
          else return `false`
        else
      for all $\pi_{j_0}(\overline{d_{j_0}}) \cdots \pi_{j_k}(\overline{d_{j_k}}) \rightarrow \bot \mid t? \in F$ **do**
        $\theta \leftarrow$ MATCH($\langle \pi_{j_0}(\overline{d_{j_0}}), \ldots, \pi_{j_k}(\overline{d_{j_k}}) \rangle, \langle c_0, \ldots, c_k \rangle$)
        if $((t?)\theta)\sigma$ **then return** `false`
        else
      return `true`
  else return `false`

The following family of sets is an example constraint of the constraint language from this chapter. This constraint is one of those built during the execution of the example from Chapter 1. This family of sets is indexed by the set {`listo`, `absento`, `not-pairo`, `==`, `nullo`}.

```
{  {(listo 0)},
   {(nullo 3)}
   {(not-pairo 2)},
   {(== 0 (cons 1 2))},
   {(absento 3 0), (absento 2 1)},  }
```

In the following, we demonstrate both that this description captures constraint microKanren languages—including those used in practice—and that this class excludes ones we wanted to exclude—suggesting that this class is precisely the class of languages we wanted to capture.

3.2. microKanren Constraint Systems

In this section we construct and instantiate constraint domains like those of Section 3.2 for miniKanren constraints of the form required by the CLP Scheme. We first remark on the term language over which our microKanren CLP languages will compute and the primitive programs that partition the primary function symbols.

3.2.1. Term Language and Primitive Predicates. This constraint language and framework permits constraints over any term language matching the requirements we described. For the remainder of this section, though, we will keep to the same particular, fixed term language of the previous example because we are describing the behavior of an existing language family.

As noted in Chapter 1, few if any miniKanren implementations explicitly specify which host-language values constitute the embeddings' terms or how programmers ought to construct them. However, miniKanren term languages are usually languages of binary trees, built of a single binary functor, `cons`—like Prolog's `cons/2`—over some countably infinite number of constants. Then `cons` is the only non-constant function symbol. We note such a term language is not unique among miniKanrens or other logic programming implementations in Lisp-like languages [221]. Table 3.1 presents a grammar for these term languages.

Term Language			
t	::=		Terms
	|	x	Term Variables
	|	c	Constants (as specified)
	|	$t :: t'$	Pairs

TABLE 3.1. The Kanren Term Language

Specifying this structured tuple of

- Racket programs for recognizing variables,
- finite sets of posary constructors with their arities,
- finite sets of Racket programs for recognizing finite sets of ground terms and,
- finite sets of Racket programs for recognizing infinite sets of ground terms

defines a specific, particular language of terms over which our miniKanren programmers write constraints. The domain of computation is then the set of ground terms generated by the set of constructors for which one of the programs accepts terms with that primary function symbol. For each constraint system, the computation domain's ($\mathscr{D}_{\mathscr{C}}$) carrier is the set of finite `cons`-labeled binary trees with the Racket symbols, strings, booleans, or the empty list at their leaves. To keep our presentation concise, we will sometimes use host-language operators besides `cons` to construct terms. In principle though, all of our terms are built with `cons`.

We now describe the construction of actual constraint systems. In the course of doing so, we present several more exemplary miniKanren constraint systems defined over the term language of Table 3.1. For each example, we will describe the domain of the constraint computation, the constraint theory, and the function that is its solver. We can know from the results of Chapter 2 that each of these example constraint systems bear the required relationships will hold between the solver, theory, and domain.

3.3. miniKanren Constraints over this Term Algebra

This section and the remainder of the chapter rely on Racket's macro system [45, 60] to instantiate the solver of the domain. Although we specify all of the constraint domain's components, we macro-generate only the solver and leave implicit the construction of the domain's other components. Even so, we can read off portions of the theory and of the domain because of their dual interpretation as logic and program. Further, we define $\mathscr{D}_{\mathscr{C}}$ as some privileged model we induce from the theory $\mathscr{T}_{\mathscr{C}}$.

In describing the macro that constructs a constraint system, we will present and discuss the `make-constraint-system` macro in pieces, marking where we have omitted aspects of the definition. We use the Unicode glyph "…" to mark elisions, and this should not be confused with the "`...`" post-fix operator of Kohlbecker [123] as used in Racket macro pattern languages. The constraint system macros also introduce constraint goal constructors for each member of the constraint domain index set. We construct these goal constructors here, but we will defer their explanation to Section 3.5.

3.3.1. Variables, Substitutions, Unification, etc. As we saw earlier in Chapter 3, we parameterized our notion of term equality by the particular posary constructors of the term language and the constants described by the primitive predicates. We will similarly parameterize our first-order unification algorithm to unify over our particular term language. In this section, from the perspective of implementing constraint systems, we will use "variables" as a shorthand for the "logic variables" of the implemented language, or the generators of the relevant term algebra. When we mean lexical variables of the implementing language, we'll say so explicitly.

One common operation on terms is to "substitute through" a given term, uniformly and simultaneously, each occurrence of a particular variable, replacing each by the same specified term. This operation is called *substituting* (or simply `subst` for short). In formal term-rewriting systems, the operation is often written $[t/x]t'$, replacing each occurrence of variable x with term t throughout t'[6].

[6]This is at least one of the more common notations. See Steele for longer discussion [203].

```
(define-syntax-rule (make-subst var? (con d ...) ...)
  (rec (sub x v t)
    (match t
      [(? var?) (if (equal? x t) v t)]
      [(con d ...) (con (sub x v d) ...)]
      ...
      [else t])))
```

Listing 3.1. Parameterized implementation of `subst` for solver

We define `make-subst` in Listing 3.1 with **`define-syntax-rule`**. The **`define-syntax-rule`** form is an easy way to construct simple syntax-rewrite rule macros in Racket. The first argument is a pattern that specifies how to invoke the macro. The pattern's first element, `make-subst`, is the name of the macro we are defining. Its second argument is a template to be filled in with the appropriate pieces from the pattern. Provided with a function for `var?` and match patterns for each of the constructors, the `make-subst` macro generates an anonymous function that performs the desired substitution operation on a triple of variable, term-replacing-variable, term.[7]

In contrast with its use as a verb, as a noun "substitution" refers to a data structure carrying variable assignments. We use these logic variables differently than we use the standard lexical variables of functional programming. Unlike an environment, a substitution may associate variables with almost *any* other term—including other unassociated variables. A substitution we consider may, for instance, associate a variable `x` with a term containing an unassociated variable `y`. Therefore, subsequently giving an association to `y` may also impact the meaning of `x`. Adding an association of a term and a previously unassociated variable can impact the values of an unbounded quantity of other variables. We uniformly represent the substitution data structure as an association list between variables (as represented in the embedding), with variable-laiden terms (as represented in the embedding). We use an association list for its simplicity and ease of implementation.

[7] This implementation assumes Racket already implements **`match`** patterns for each of the term languages' constructors; if not, the constraint language implementer would need to add them, using something like the `~struct` or `define-match-bind` of the Racket `generic-bind` library.

```
(define-syntax-rule (make-subst-all var? (con d ...) ...)
  (rec (w* t s)
    (match t
      [(? var?)
       (cond
         [(assoc t s) => cdr]
         [else t])]
      [(con d ...) (con (w* d s) ...)]
      ...
      [else t])))
```

LISTING 3.2. Parameterized implementation of `subst-all` for solver

We rely on the primitive host-language function `assoc` to check if `u` is the first element of a pair in substitution `s`. If so, `assoc` returns the pair; if not, `#f`. When, rather than replacing a *single* variable by another, we instead wish to uniformly and simultaneously replace occurrences of any of a *list* of variables by corresponding values, we use this similarly-derived `subst-all` method. When substituting in parallel like this, we say that we substitute *across* a substitution and *through* the term. Structurally it is very similar to Listing 3.1. Unlike many other languages, Racket's (Scheme's) **`cond`** accepts any value as its first argument, and any value except `#f` is considered true enough (or "truthy"). The **`cond`**-block in Listing 3.2 takes advantage of this, using the **`=>`** ("arrow syntax") to send any non-false value to the one-argument function `cdr`.

Not just any association of variables to terms qualifies as a substitution. To ensure certain well-formedness conditions (e.g. that all terms represent only finite structures), we must ensure that no variable is associated to a term that *occurs* within it. The macro of Listing 3.3 defines a function that checks if a given variable occurs in a term.

With these pieces in hand, we can macro-generate a method that, given a substitution and a pair of a variable and a term with which to possibly extend that substitution, return an extended substitution if possible, and **`false`** if those two terms do not appropriately extend the substitution. The use sites of the resulting function ensure that the term `t` is up-to-date with respect to the present substitution `s`.

```
(define-syntax-rule (make-occurs? var? (con d ...) ...)
  (rec (o? x v)
    (match v
      [(? var?) (equal? x v)]
      [(con d ...) (or (o? x d) ...)]
      ...
      [else false])))
```

Listing 3.3. Parameterized implementation of `occurs?` for solver

```
(define-syntax-rule (make-ext-s var? diag ...)
  (let ([occurs? (make-occurs? var? diag ...)]
        [subst (make-subst var? diag ...)])
    (λ (x t s)
      (cond
        [(occurs? x t) false]
        [else
         (cons `(,x . ,t)
           (~for/list ([($: a d) s])
              (cons a (subst x t d))))])))))
```

Listing 3.4. Parameterized implementation of `ext-s` for solver

These pieces help construct a concrete implementation of unification for the term language of Listing 3.5. This implementation follows the general unification algorithm alluded to in Algorithm 1. We parameterized the implementation of unification by the term language. Instantiating these parameters gives a concrete implementation from the parameterized implementation of the general unification algorithm. This macro generates a `unify` similar to microKanren's, but operating over the idempotent substitutions discussed above. Because unification is a two-way pattern matching, the macro constructs two different patterns for each posary constructor. For this reason the macro takes in as parameters pairs of pattern variables for each constructor.

The resulting anonymous function (hereafter `unify`) is a fairly pedestrian unification implementation. If both terms are the same, return the substitution parameter. If not, but one's a variable, attempt to return an extended substitution, and similarly in the opposite case. If neither term is a variable, then both terms have a primary function symbol. If, for any

```
(define-syntax-rule (make-unify var? subst-all (c p1 p2) ...)
  (let ([ext-s (make-ext-s var? (c . p1) ...)])
    (rec (unify u v s)
      (let ([u (subst-all u s)] [v (subst-all v s)])
        (match* (u v)
          [(u v) #:when (equal? u v) s]
          [((? var?) v) (ext-s u v s)]
          [(u (? var?)) (ext-s v u s)]
          [((c . p1) (c . p2))
           (for/fold ([s s])
                     ([t1 (list . p1)]
                      [t2 (list . p2)])
             #:break (not s)
             (unify t1 t2 s))]
          ...
          [(_ _) false])))))
```

LISTING 3.5. Parameterized implementation of `make-unify` for solver

of the term language's posary function symbols both terms begin with that function symbol, then fold `unify` across the immediate subterms. If none of those situations manifest, then the two terms are not unifiable, and fail. This generated `unify` is not especially performant. Under a deep embedding, we could guarantee a uniform structure and instant access to the primary function symbol of non-variable terms. We required only a shallow embedding of the term language; by contrast, this shallow embedding forces us to match against and destruct pairs of terms to access their subterms.

The `unify` generated for miniKanren's term language is an unusual special case. For a term language with no more than one posary constructor, this macro's `unify` performs no superfluous matching for posary terms. Such languages are a sort of "base case" for which both unification over deep and shallow term embeddings has at most a single recursive case for compound terms. The `unify` of a language with n posary constructors requires n matches before failing on two posary terms of different primary function symbols. Our term construction betrays an unfortunate reliance on the direct representation of terms. This reflects a contingent decision to maintain some continuity with the typical miniKanren implementation, not a necessary limitation of macro-generating constraint solvers. We could,

```
(define-syntax-rule
  (make-fail-check subst-all ([(b x ...) ...] [(p? fa  ...) ...]))
  (λ (s)
    (~for*/or ([($list x ...) b] ...)
       (and (p? (subst-all fa s) ...) ...))))
```

LISTING 3.6. Building execution of failure rules for solver via `make-fail-check`

for instance, mix a shallow embedding of a constraint system with a deep embedding of the underlying term language. Even in the best of cases, however, our system generates a deficient implementation of unification: idempotent substitutions can cause exponential blow-up in the size of terms over more compact representations. We discuss ameliorating some of these problems in Section 3.4.

Supposing that for the constraint in question we have constructed a substitution (like from `unify`) that reduces every $\mathscr{T}$-term of the constraint into some canonical form. Say that we can use it to "wring out" all of the equality information of the constraint. Suppose further that we have access to members of the constraint's family of sets via the index set. With such a substitution and access to the constraints' elements, we could use one of the failure rules of the specification to look for a particular kind of failure.

The `make-fail-check` macro takes a failure rule: a sequence of negative atomic constraint patterns over which the failure is defined and a templated sequence of conditions that must be met to cause failure. The macro generates a function from a term-normalizing substitution to a boolean. This boolean reflects if the constraint violates that particular rule. The `~for*/or` operation operates across each tuple from $\overline{r}_1 \times \cdots \times \overline{r}_k$ in the constraint, and returns the value `#t` if for any such tuple all of the condition for failure hold. We can `subst-all` across all terms in the failure condition test.

It takes more than an implementation of `unify` to construct a normalizing substitution. Even after using `unify` to solve the atomic equality constraints, the resulting substitution may not be a normalizing substitution due to rewrite rules. Rewrite rules also have a syntax of negative atomic constraint patterns and condition templates. In between those two pieces,

rewrite rules also have a sequence of *rewrites*. These rewrites are two equal-length lists, the first of pattern variables and the second of terms in $\mathscr{G}$—i.e. finite functions from pattern variables to constants.

We only require a pattern-match to set-up one of these rules to prepare to execute the checks. The result of a rewrite is either failure (because of a clash), or a similar substitution with some variable uniformly replaced with a constant. When the atomic equality constraints are consistent, and where θ is an mgu for those atomic equality constraints, then the rewrite system of these rewrite rules on the set of terms $\theta[\mathscr{T}(\Sigma, X)]$ satisfies the strong Church-Rosser property. As such, the rules can be executed in any order, and in fact any instance of any rule can be executed in any order. No matter the order, each instance of each rule needs testing only once, and any resulting substitution will be equivalent modulo variable renaming. Because these rules only ever introduce new equalities, no assignment of a variable to a ground constant can cause a rule to "fire" when it would not have otherwise, and because unification produces a most general unifier, all resulting substitutions under any ordering will be equivalent up to variable renaming. Further, if any one ordering causes failure, all other orderings must as well. These rules describe *defining formulae* for constants—at most assigning a constant to a variable—as sequences of negative atomic constraints.

The more general atomic equality constraints can merge equivalence classes of variables without grounding them to some particular ground term. Because of these limited kinds of rewrites that defining formulae express, we can eschew a fix-point algorithm for executing these rules.

In practice, the solver applies a rewrite rule across each instance in the constraint, one after another, of the rule's pattern. Our system will execute each instance of each rule in the order listed. As mentioned, such a system need not even test all a rule's instances together; this behavior in our systems is merely a contingent design decision. Internally, the system describes the behavior of equality constraints as an unconditional rewrite rule whose instances must all be executed first. The `make-normlzr` macro takes in both definitions of `subst-all` and `unify`, along with the internal representation of rewrite rules. In this representation, the assignments are listed as sequences of variables and the constants to

```
(define-syntax-rule (make-normlzr subst-all unify
                      ([(b x ...) ...] [vs cs] [(p? fa ...) ...]))
  (λ (s)
    (~for*/fold ([s s])
                ([($list x ...) b] ...)
                #:break (not s)
                (if (and (p? (subst-all fa s) ...) ...)
                    (for/fold ([s s])
                              ([t1 (list . vs)]
                               [t2 (list . cs)])
                      (unify t1 t2 s))
                    s))))
```

LISTING 3.7. Building execution of rewrite rules for solver via `make-normlzr`

which they are assigned. For each $\overline{r}$-rule listed, the system tests in turn every negative atomic $\overline{r}$-constraint of the general constraint being tested. This amounts to trying all tuples of appropriate $\overline{r}$-constraints and accumulating up, from the initial substitution input, the augmented substutition that results. If the substitution ever becomes `#f`, indicating the equality constraints themselves are already unsatisfiable, the execution of this rule short circuits with failure as the result. By combining these two pieces together we construct the full solver for a constraint. Internally, constraints are represented by a hash-map from identifiers in $\overline{R} \cup \{\texttt{==}\}$ to lists of n-tuples of terms, where n is appropriate for the $\overline{r}$ in question.

The `make-solver` macro once again takes in both definitions of `subst-all` and `unify`; it must have the definitions in order to pass them along to subsidiary macros. The `make-solver` macro constructs a function that accepts a constraint, and introduces the names $\overline{r}$ into scope for each list of tuples under that name in the constraint. The solver first executes each rewrite rule in sequence. If this sequence fails to generate a substitution, then the function returns `#t`, indicating the set is inconsistent. If this sequence produces a valid substitution, then the solver uses that substitution to check if any of the failure tests indeed fail.

The actual implementation of `make-constraint-system` is the only technically sophisticated macro in the implementation, and not particularly so. The pattern accepts all of the pieces that define a constraint system, including two identifiers for the names of the

```
(define-syntax-rule
  (make-solver subst-all unify (cid ...) (rr ...) (p ...))
  (λ (S)
    (let ([cid (hash-ref S 'cid)] ...)
      (cond
        [((compose (make-normlzr subst-all unify rr) ...) '())
         => (or/c (make-fail-check subst-all p) ...)]
        [else #t]))))
```

LISTING 3.8. Implementation of `make-solver`

```
(define-syntax-parser make-constraint-system
  [(_ #:var? var?
      #:posary-constructors ((c:id . n:nat) ...)
      #:infinite-types (ip:id ...)
      #:finite-types (fp:id ...+)
      #:== ==
      #:=/= =/=
      #:primitive-predicates ((ppn:id ((~datum one-of) fp/ip ...+)) ...)
      #:term-structural-functions ((sfn:id sfcls ...+) ...)
      #:recursive-predicates ((rpn:id [(t ...) body] ...+) ...)
      #:constraints (((rcn:id x ...) nrp) ...)
      #:rewrite-rules (rr:rewrite-rule ...)
      #:failure-rules (fr:fail-rule ...)
      #:sugar-constraints (((sugn:id suga:id ...) b) ...))
   …])
```

LISTING 3.9. Pattern for implementation of `make-constraint-system`.

binary constraints representing equality and disequality constraints in the generated CLP language. The recursive predicate definitions use a homogenized, IFF syntax as described in Section 2.5.2 and Shepherdson [194].

The two syntax classes of Listing 3.10 provide surface syntax for failure rules and rewrite rules, respectively. Each de-sugar to their own respective internal form introduced as a syntax attribute for the class. These remove the sugar, and in the case of the rewrite-rule, replace the pairs separated by `=>` with a list of variables and a list of constants.

We separate the definition of the template into two pieces to discuss separately. The actual internal definition of the predicate on constraints, `invalid?`, is the most sophisticated part of the template, and we discuss it on its own. It lexically introduces the primitive

```
(begin-for-syntax
  (define-syntax-class fail-rule
    #:attributes (norm)
    (pattern ((~literal for-all)
              [(cid:id x:id  ...+) ...+]
              (~datum #:fail-when) [gpapp ...+])
             #:with norm #'([(cid x  ...) ...] [gpapp ...])))
  (define-syntax-class rewrite-rule
    #:attributes (norm)
    (pattern ((~literal for-all)
              [(cid:id x:id  ...+) ...+]
              #:rewrite
              [(v (~datum =>) c) ...+]
              #:when
              [gpapp ...+])
             #:with norm #'([(cid x  ...) ...]
                            [(v ...) (c ...)]
                            [gpapp ...]))))
```

Listing 3.10. Syntax classes for failure rules and rewrite rules.

predicates formed from the programs of respectively finite or infinite co-domain of the partition terms provided in the pattern. Within that scope it defines each recursive predicate by the sum of the clauses with that relation symbol as head and the appropriate number of arguments. It attempts to match each clause against the head and subsequently attempt the body. Failing in the alternate case ensures that each clause results in a boolean value, and the surrounding `(or …)` provides disjunction here covering all of the cases. This is precisely where we introduce the closed world assumption in the constraint system. In that same scope it also introduces the functions describing morphisms, possibly constructed using one or more of the functions `ppn`. We introduce in each the special case of acting as identity on variables. Within these defined, it then locally introduces `subst-all` and `unify`, and finally proceeds to invoke `make-solver`, with the names of the equality constraints and all of the negative constraints, the rewrite rules together with the added internal rule for unconditionally rewriting equality constraints, and finally the list of failure rules in

```
(define-syntax-parser make-constraint-system
  […
   (with-syntax
     ([(p1 ...) (stx-map make-pattern #'(n ...))]
      [(p2 ...) (stx-map make-pattern #'(n ...))]
      …)
     #'(…
         (define invalid?
           (let ([ppn (or/c fp/ip ...)] ...)
             (letrec ([rpn (λ args
                             (or (match args
                                   [(list t ...) body]
                                   [else false])
                                 ...))]
                      ...
                      [sfn (match-lambda**
                             [((? var? X)) X]
                             sfcls ...)]
                      ...)
               (let* ([subst-all (make-subst-all var? (c . p1) ...)]
                      [unify
                       (make-unify var? subst-all (c p1 p2) ...)])
                 (make-solver subst-all unify (== =/= rcn ...)
                  [rr.norm ... ([(== t1 t2)] [(t1) (t2)] [])]
                  [([(=/= a d)] [(equal? a d)])
                   ([(rcn x ...)] [nrp]) ... fr.norm ...])))))
         …))])
```

LISTING 3.11. `make-constraint-system` template's implementation of `invalid?`.

normalized form, together with the definitions of the negative constraints themselves (which are a kind of failure rule of their own), and the automatically supplied failure rule for disequality constraints.

The system constructs match patterns for each of the constructors using the given number input, and constructing a list of that many unique identifiers via `generate-temporary`. Because this is an effectful operation, we actually execute it twice to make two distinct match patterns when constructing `unify`.

```
(define-for-syntax (make-pattern ns)
  (build-list (syntax->datum ns) generate-temporary))
```

Listing 3.12. Implementation of `make-pattern` function

```
(define-syntax-parser make-constraint-system
  […
   (with-syntax
     (…
      [S0 (syntax-local-introduce #'S0)])
     #'(begin
         …
         (define S0
           (make-immutable-hash eqv '((==) (=/=) (rcn) ...)))
         (define == (make-constraint-goal-constructor invalid? '==))
         (define =/= (make-constraint-goal-constructor invalid? '=/=))
         (define rcn (make-constraint-goal-constructor invalid? 'rcn))
         ...
         (define (sugn suga ...) b) ...))])
```

Listing 3.13. Remaining pattern for `make-constraint-system`'s implementation

The remainder of the template introduces an identifier `S0` into scope to use as the initial, empty constraint. We also introduce host-language level functions that act as implementations of each constraint as a goal constructor in the shallow embedding of the language's implementation. These take the definition of `invalid?`; this identifier is otherwise unavailable to the CLP language user. Finally, we introduce the "sugar constraints" that are just shallow wrappers around a function producing specialized versions of some set of constraints from the constraint family.

This `make-constraint-goal-constructor` introduces the shallowly embedded CLP language's goals; these make up the language's interface to the constraint solver. The implementation partially instantiates a heavily curried function once for each member of the index set. The `invalid?` identifier gives each access to the solver. The element of the index set will also serve here as a hash key. We discuss the macro's implementation in Listing 3.19 and

explain the rest of its behavior in context in Section 3.5. In Section 3.3.1.1, we demonstrate further examples of some generated constraint systems beyond the example constructed thus far in this chapter.

3.3.1.1. *A constraint system of* {==, =/=}. We first begin with a simple example—in fact the simplest constraint system our macros can construct. We use a system with *just* equality and disequality[8] constraints to exhibit constructed miniKanren constraint systems. As such, there are good amounts of technical machinery we do not actively use in this example. We describe this constraint system as a useful starting point, and as a point of comparison when moving forward to more complicated examples. One of our disequality constraints fails not merely when the terms of the constraint are syntactically distinguishable, but when terms fail to unify viz. the present substitution. This is to say disequality constraints, as we define them, behave soundly like that of Comon and Rémy [43], and unlike those of Prolog II. Barták [13] terms CLP over finite trees with syntactic equality and disequality constraints CLP(H).

We exhibit the solver for constraints of this domain generated by our constraint system in Listing 3.14. Any primitive constraint in this miniKanren constraint system will be one of those two kinds. The framework through which we build the solver, and the rest of the constraint system, includes as given a routine for unification (with occurs-check) of two terms in a valid substitution that produces a most general unifier of all equality-constrained pairs terms. We will introduce no additional Horn-clause predicates over the underlying structure. We also export no negative constraints beyond =/=, which our constraint systems include per force. Since we introduced no additional structural predicates over terms, we need no additional failure checks. Since we have no additional negative constraints in the constraint domain, all equalities expressed in any constraint will be explicitly written with

[8]Some authors (e.g. Colmerauer [40], Lassez et al. [137], and Makowsky [151]) refer to our disequality constraints as inequality constraints. We refer specifically to the $\neq$ relation "not-equal-to", and of more general inequalities ($\leq$, $\geq$, etc.) we will say no more.

==. This is to say that every equality the constraint implies comes solely from the ==-literal portion of that constraint. Furthermore, in this specification we will introduce no sugar constraints over this basic constraint domain.

```
(make-constraint-system
 #:var? number?
 #:posary-constructors ((cons . 2))
 #:infinite-types (symbol? string? pair?)
 #:finite-types (boolean? null?)
 #:== ==
 #:=/= =/=

 #:primitive-predicates  ()
 #:term-structural-functions ()
 #:recursive-predicates ()
 #:constraints ()
 #:rewrite-rules ()
 #:failure-rules ()
 #:sugar-constraints ())
```

LISTING 3.14. Racket definition of a solver for equality and disequality constraints

In Listing 3.14 we provide most of the definition of a solver for equality and disequality constraints. Since the term language remains fixed, and we will in this section continue to use == and =/= for our equality and disequality constraints, we will elide these elements in the subsequent descriptions. Furthermore, as the implementations of `unify` and other functions essential to unification are both large and consistent throughout, we will omit their expansions in the following. When possible, we have eliminated empty binding forms and hand-substituted through redexes to aid the presentation.

Even accepting these hand-simplifications, the generated function quickly becomes unwieldy to read and digest. This exhibits on its own the benefits of the parameterized implementation of constraint systems. The user can provide a high-level, logical characterization of the constraints and their implementations, and avoid the details of the implementation and actual execution. We take advantage of the benefits of this approach in describing the subsequent and increasingly more complicated constraint systems. In Sections 3.1.3 to 3.1.9,

```
(define invalid?
  (let* ([subst-all …]
         [unify …])
    (λ (S)
       (let ([== (hash-ref S '==)] [=/= (hash-ref S '=/=)])
         (cond
           [(~for*/fold ([s '()])
                        ([($list t1 t2) ==])
                        #:break (not s)
                        (for/fold ([s s])
                                  ([t1 (list t1)]
                                   [t2 (list t2)])
                          (unify t1 t2 s)))
            => (λ (s)
                 (~for*/or (([$list a d] =/=))
                    (and (equal? (subst-all a s) (subst-all d s))))))]
           [else #t])))))
```

LISTING 3.15. Racket implementation of an `invalid?` for a solver of `==` and `=/=` constraints

we implemented the full complement of standard "constraint miniKanren" constraints. In the next section, we add constraints beyond those usually used for quines and many of the other standard examples.

3.3.1.2. *Additional Exemplary Constraints.* Unlike the constraint domains we have so far constructed, when building the following constraints and constraint domains, we intend to demonstrate some newly expressible constraints (some of dubious merit) as well as some recently added to miniKanren constraint systems. Some of the latter inspired this work. The constraint domain(s) will share the basic structure in common with those discussed above.

One example we can build is a system with a left-leaning list constraint. By a "left-leaning list" we mean treating the left-side of the tree as the "spine" rather than the usual right side. The following tree is an example of such a list: `'(() . (b . (c . (d . e))))`. We need an auxiliary, non-term function in the constraint system's domain. This function's interpretation in the domain is that of the following Racket function:

$$car*(\texttt{X}) \begin{cases} \texttt{X} & \text{if } \texttt{(non-pair? X)} \\ car*(\texttt{Z}) & \text{if } \texttt{(pairr? X)} \text{ and } \texttt{X} = \texttt{(cons Z \_)} \end{cases}$$

We can also straightforwardly implement pair and non-boolean constraints. The former are not especially useful, because unification with two fresh variables would indicate this just as well. However, they do suggest both some of the versatility of our system, and this near duplication of functionality is perhaps unexpected. It is interesting to compare this latter kind of constraint with the failed `booleano` constraints example of Section 3.3.2.

Two other interesting kinds of constraints in this example are the `succeed` and `fail` constraints. They require no auxiliary constraint relation symbols, and we build them with primitive predicates `any-term?` and `no-term?` recognizing respectively all terms and no terms in the term language.

```
(make-constraint-system
 …
 #:primitive-predicates
 ((no-term? (one-of))
  (any-term? (one-of boolean? null? pair? symbol? string?))
  (booleann? (one-of boolean?))
  (non-pair? (one-of boolean? null? symbol? string?))
  (non-list-constant? (one-of boolean? symbol? string?)))
 #:term-structural-functions
 (((car* [((? non-pair? X)) X]
         [((cons Z _)) (car* Z)])))
 #:recursive-predicates ()
 #:constraints
 ([(fail? t) (any-term? t)]
  [(succeed t) (no-term? t)]
  [(lllisto l) (non-list-constant? (car* l))]
  [(pairo l) (non-pair? l)]
  [(non-booleano x) (booleann? x)])
 #:rewrite-rules ()
 #:failure-rules ()
 #:sugar-constraints ())
```

LISTING 3.16. Racket definition of a solver with a left-leaning list constraint, and others

```
(has-null? X)              ⟵  (nulll? X)
(has-null? (cons _ Z))     ⟵  (has-null? Z)
(has-null? (cons Y _))     ⟵  (has-null? Y)
(has-symbol? X)            ⟵  (symbol? X)
(has-symbol? (cons _ Z))   ⟵  (has-symbol? Z)
(has-symbol? (cons Y _))   ⟵  (has-symbol? Y)
(one-of? X X _ _)          ⟵  true
(one-of? X _ X _)          ⟵  true
(one-of? X _ _ X)          ⟵  true
(revH? '() X X)            ⟵  true
(revH? (cons X Y) Z W)     ⟵  (revH? Y (cons X Z) W)
```

FIGURE 3.1. Recursive predicate definitions for exemplary solver

```
(improper-listo X)              ≜  ¬(nulll? (rac X))
(nrevHo X Y Z)                  ≜  ¬(revH? X Y Z)
(non-nullo X)                   ≜  ¬(equal? X '())
(all-but-symbol-or-booleano x)  ≜  ¬(sym-or-bool? x)
(devoid-of-nullo T)             ≜  ¬(has-null? T)
```

FIGURE 3.2. Negative constraint definitions for exemplary solver

It follows logically that some term being both list and a non-pair imply that term's disequality with `'()`, but nowhere do our systems rewrite or capture this knowledge. Indeed this is by design, since that is incidental to the question of consistency. We can distinguish the implementations of `succeed` and `fail` constraints from the implementations in Section 3.6.1 of primitive `succeed` and `fail` *goals*. We will consider next a second example constraint system. Rather than odd primitive predicates or term-structural functions, this second example explores unorthodox recursive predicates.

The `has-null?` predicate recurs like `mem?` over a tree-structure, but rather than comparing one term against another (sub-)term, it tests the (sub-)structure against a predicate. The `one-of?` predicate tests for membership in a tuple of subsequent arguments, and does so with n clauses for $n+1$ many arguments. This could instead have been written recursively with a list-membership operation, but this isn't wrong and while less general is perfectly correct for our intended use case. It is curious that clauses' heads' terms are all just at most one-level of term-structure over $\mathscr{C} \cup Z_i$.

We write the negative constraints in the now-standard fashion. We discuss in Section 3.3.2 that `booleano` is disallowed; note however that `symbol-or-booleano` is permitted—in *this* constraint system. This property exemplifies a key feature of our design criteria: since this system lacks primitive predicate or predicates to exclude the symbols, and since our language of writing recursive predicates doesn't permit any computable function to describe all of the symbols (as symbols are non-structural constants), there is no way to induce any finite constraint from these combinations of booleans and symbols. When implementing `non-nullo`, we relied on the underlying `equal?` method and a constant, rather than the primitive predicate. This too demonstrates another duplication of functionality one might not expect in the most parsimonious system.

There are, in addition, interactions that we needed to consider, and sugar constraints we might choose to add. In Listing 3.17 on page 72 we exhibit the full specification of the constraint system described for this example. We implemented non-palindrome and non-mirror-image constraints as sugar constraints over the included `nrevHo`.

Users will notice some redundancy when writing specifications; experienced functional programmers may yearn here for higher-order constructions. The constraint specification language is indeed less expressive than many existing, more fully-featured languages. However, there is something to be said for a syntax and languages that expresses *precisely* the expressivity needed to capture all and only the constraint systems of interest.

3.3.2. Non-examples. Some of the important properties we described and relied upon earlier in this chapter *fail* to hold for the following illustrative non-examples. We include them partly as warnings. The booleans are one of the simplest non-trivial finite domains. As such, many of these failed examples make use of the booleans, or attempt to permit boolean constraints. Boolean constraints themselves most simply exhibit the problem.

Attempting to include a primitive predicate `non-boolean?` via (a syntactically valid but inadmissible) definition like `(non-boolean? (one-of null? pair? symbol? string?))` is the first step in exhibiting this specification bug. This predicate is dangerously and impermissibly co-finite. Similarly, attempting to use `non-boolean?` not as a constraint itself, but as the

building block, say for, some predicate `ends-in-non-bool?`, and then using that as the basis for a negative constraint. That putative `ends-in-a-booleano` constraint is by itself fine, but when combined with `not-pairo` causes a real problem. The constraint designer should be able to add a new constraint without having to reconsider the whole architecture—that is one of our design goals. At most he should need to consider how this newly-added constraint interacts with others "at the limit". This error/non-example comes from and corrects one of my early mistakes in Hemann and Friedman [85].

We cannot write a predicate expressing "a is non-member of b" with finitely many clausal formulae and without negation. This means that although we have `absento`, our constraint systems cannot express a "`presento`" (equiv. "`membero`"). Expressing such a predicate in the clausal language of this chapter would require another, second level of negation. We know this via the syntactic characterization from either Makowsky [151], Volger's "crisp theory" paper [217], or Vel [215].

By similar reasoning and argumentation, we cannot express a general predicate `tail-does-not-end-in?`, which makes impossible an `ends-ino` constraint. This seems like a straightforward generalization of the `listo` and `improper-listo` constraints. However these two opposites are permitted because they are similar extensions of mutually exclusive primitive predicates.

```
(make-constraint-system
 …
 #:primitive-predicates
 ((non-sym-non-bool? (one-of null? pair? string?))
  (nulll? (one-of null?))
  (non-list-constant? (one-of boolean? symbol? string?)))
 #:term-structural-functions
 ((rac [((? non-pair? X)) X]
       [((cons _ Z)) (rac Z)]))
 #:recursive-predicates
 ((one-of? [(W W _ _) true]
           [(W _ W _) true]
           [(W _ _ W) true])
  (one-of-mem? [(T W _ _) (mem? W T)]
               [(T _ W _) (mem? W T)]
               [(T _ _ W) (mem? W T)])
  (mem? [(X X) true]
        [(X (cons _ Z)) (mem? X Z)]
        [(X (cons Y _)) (mem? X Y)])
  (improper-list? [(X) (non-list-constant? (rac X))])
  (has-null? [(X) (nulll? X)]
             [((cons _ Z)) (has-null? X Z)]
             [((cons Y _)) (has-null? X Y)])
  (revH? [('() X X) true]
         [((cons X Y) Z W) (revH? Y (cons X Z) W)]))
 #:constraints
 ([(improper-listo X) (nulll? (rac X))]
  [(nrevHo X Y Z) (revH? X Y Z)]
  [(non-nullo X) (nulll? X)]
  [(symbol-or-booleano x) (non-sym-non-bool? x)]
  [(devoid-of-nullo t) (has-null? t)])
 #:rewrite-rules
 ()
 #:failure-rules
 ([for-all ([improper-listo i] [nrevHo a b c])
           #:fail-when [(one-of? i (rac a) (rac b) (rac c))]]
  [for-all ([nrevHo a b c] [symbol-or-booleano sb])
           #:fail-when [(one-of? sb (rac a) (rac b) (rac c))]]
  [for-all ([nrevHo a b c] [devoid-of-nullo t])
           #:fail-when [(one-of-mem? t (rac a) (rac b) (rac c))]]
 #:sugar-constraints
 ([(non-mirror-imageo X Y) (nrevHo X '() Y)]
  [(non-palo X) (non-mirror-imageo X X)]))
```

Listing 3.17. Racket definition of a solver with unorthodox recursive predicates

3.4. Potential future improvements, enhancements, and alternative designs

Here, we suggest a number of different improvements or alternate approaches to the design described above. Some of these are merely more complex and thus less obviously correct than the more straightforward approach to specifying correct constraint systems we followed above. Others extensions add functionality, and are independent of our design decisions.

Iterated Unification Problems At present, each execution of a constraint begins anew. We do not accumulate any substitution information, even though for each "branch" of the computation the constraint grows monotonically. One improvement would be treating equality constraints as instances of an *iterated unification problem*, rather than repeated instances of the *general unification problems*—meaning keep around a thus-far accumulated substitution.

Triangular Substitution The `unify` function of Listing 3.5 uses direct representations of terms in the substitutions it constructs. So-called "triangular" substitutions [11] are an analogous but generally more efficient data structure. Here, a given variable `x` that we have now solved for may occur in previously-bound terms and we do nothing to remove indirections that result from extensions to the substitution. Such definitions of substitution are especially amenable to structure-sharing and implementation with persistent data structures. This decision necessitates changes in the other functions that modify or access the substitution data structure. There are other, more efficient persistent structures than association lists we could also use here [44].

Linear-time Unification Implementation These triangular substitutions relate closely to vastly improved, linear time unification algorithms. For instance, the algorithm of Paterson and Wegman [170] or the modified algorithm of Martelli and Montanari [153] are both linear time. In practice, these may be inferior to more complex and finely-tuned

structures; see Siekmann [198, §3.1.1] or Albert et al. [3, 4] for more information. Several authors have already implemented such advanced unification algorithms for miniKanrens[9] based on this, or possibly improved versions that are better in practice.

Optional Automatic Constraint Name Generation At present, the user is required to include, both for hygiene in implementation and for some sanity in construction, the precise names for the negative constraints. This is useful for providing non-obvious names to the negative constraints; things like `absento` spring to mind. Often, however, the user manually gives some variation of the obvious name to the negative constraint. It would be great to allow automatically naming with the obvious names for negative constraints.

Automatic Partition-based Constraint Failure Definitions Constraints like `symbol` and `not-symbol` are negative constraints defined in terms of a primitive predicate built from the program-based partition have an obvious failure case. Whenever the constraints on a single term (say a variable $x \in X$) describe the total set of primitive predicates, this must cause a failure. These can be automatically calculated, so in principle the system's user should not need to manually provide these. Performing this calculation would either necessitate some more complexity for user input (to determine which constraints are negations of primitive predicates), complicate the implementation macros for the additional computation, or possibly both.

Staged specifications Our system's macro interface currently demands the whole specification all at once. We should want to pass in the minimial, initial part that then generates a macro that takes in the next part, that then generates a macro that takes in the next part, and so forth. We could use the specification's prior parts in macro-generating checks of the latter. This design also permits an architect to partially specify an implementation—say, defining a term language and primitive relations, but allowing a downstream constraint writer to specify the remaining parts of the constraint domain, who then passes the complete language off to a programmer.

[9] See `github.com/cbrooks90/martelli-montanari` or `github.com/mvcccccc/C311Pub/blob/master/mk.rkt` for two such implementations.

3.5. The microKanren Language

Modern state of the art CLP languages (for instance many Prologs) come equipped with decades of features and tools. In addition to negation (`not`), they also carry `assert` and `retract`, that allow the dynamic introduction and removal of facts and rules from the database, cuts (`!`) that prune the search space, as well as I/O operations, printing, and other such "utility features". Our languages are significantly less featured. The main facilities of this constraint-independent portion are introducing variables, structuring larger programs, and controlling their execution (i.e., search). Each of these facilities' behavior is independent of the particular constraint domain. Separating the control flow and variable introduction from the constraint management fixes the logic programming language's structure (a language "spine") against which to add and explore constraints via specifying constraint domains. In this section, we describe this constraint-independent portion of the system for building a miniKanren CLP language and develop an implementation. Since we parameterized this segment by the exact constraint domain this portion of the language and its implementation is common to all of the constraint domain-generated constraint miniKanren languages. The language designer instantiates the constraint domain $\mathscr{C}$ to complete the constraint logic programming language's definition. For many languages, the implementation of the constraints themselves and the constraint failure-check predicates dwarf the remaining part of the microKanren framework.

We emphasize this section's portion of the languages' implementation is not an interpreter, but a shallow embedding into some functional host language—as is each member of the class of full language implementations we generate. We develop here *compositional* embeddings of logic programming; this portion of each logic programming language embedding amounts to a compositional, executable semantics for our programs as host language expressions. The pure, relational programming languages this system constructs include the language of the `append` example of Section 1.6 on page 17. We note the choice of whether to shallowly (deeply) embed the control of the language is independent of the choice to shallowly (deeply) embed the terms. Although our language implementations do not include

this facility, we take the liberty of using the constraint simplification of typical Kanren implementations, and restrict our examples' constraints to a set typical of implementations with well behaved simplifiers. To benefit the reader, we include types as comments. These types resemble those of, e.g. Spivey and Seres [202]. We label some of them more precisely, because we take particular concern for and more precisely control termination that Haskell demands of them.

3.5.1. Preamble on Search. Typically CLP languages' implementations default to some particular, search strategy (e.g. depth-first search, breadth-first search, or iterated deepening depth-first-search). More complex languages' implementations may give the programmer some fine-tuned control for selecting, modifying, or switching between search strategies. Implementers can build complex, "blended" strategies in shallow embeddings via layering monad transformers over basic search monads, a la Schrijvers et al. [188]. Other applications of search techniques use added heuristics, either explicitly provided by the user, or internally set or generated, to augment the search behavior.

We do not have a similar approach for varying the selection rule. microKanren achieves weak independence of the selection rule, as described in Jaffar et al. [111], but we do not achieve full independence of the selection rule, because we implement a left-to-right literal selection strategy. All of our implementations rely on a left-to-right selection rule that has been the standard choice since the earliest implementations of Prolog [42].

The implementations we construct do not search with some particular, precise search strategy. Nor does the implementation expose any search guidance heuristic to the programmer—directly. The program structure of the particular miniKanren program and query to execute are also inputs in determining a concrete search behavior. The control embedded in the implementation deterministically derives the precise search behavior from these inputs though. So, we rather say that these languages implement a search meta-strategy, and all the languages we implement will share the same search meta-strategy. We will describe the operational behavior of the implementation-specific portions that

MicroKanren Datatypes		
`Goal`	::	`State` $\rightarrow$ `Stream`
`State`	::	`Constraint` $\times$ `Nat`
`Stream`	::	`Mature` $\mid$ `Immature`
`Mature`	::	() $\mid$ `State` $\times$ `Stream`
`Immature`	::	`Unit` $\rightarrow$ `Stream`

TABLE 3.2. MicroKanren Datatypes

determine search beginning in Section 3.6. In short, we show how to implement complete search strategies that avoid much of the overhead associated with a breadth-first search or other traditional complete search techniques.

3.5.2. microKanren terminology. We first explain some terminology fundamental to describing our implementations. Table 3.2 summarizes this information.

Goals. We implement goals as functions that take a *state* and return a *stream* of states. They consist of primitive constraints like `(== x y)`, relation invocations like `(append 'x q '(x b c))`, and their closure under operators that perform conjunction, disjunction, and variable introduction.

Relation. A miniKanren relation has a different logical meaning than a collection of Horn clauses, closer instead to a *completed predicate*.

State. We execute a program p by attempting an initial goal in the context of zero or more relations. The program proceeds by executing a goal in a *state*, which holds all the information accumulated in the execution of p. Most importantly, the state contains a constraint as a data structure that holds the accumulated primitive constraints. The state also contains a *counter* for assigning unique identifiers to fresh variables. Every program's execution begins with an *initial state* devoid of any constraint information and a new variable count.

Streams. Executing a goal in a state `S/c` (connoting a pair of a `s`tate and a `c`ounter) yields a stream. A stream may take one of three shapes:

empty: The stream may be empty, indicating that the goal is unachievable in `S/c`.

answer-bearing: A stream may contain one or more resultant states. In this case, each element of the stream is a *different* way to achieve that goal from `S/c`. Here, we mean "different" in terms of control flow (i.e., disjunctions); the same state may occur many times in a single stream. Our streams are not necessarily infinite; there may be finitely many ways to achieve a goal in a given state. We call these first two shapes *mature*.

immature: An immature stream is a delayed computation that will return a stream when forced.

The final step of running a program is to continually force the resultant stream until it yields a list of answers. microKanren programs however, are not guaranteed to terminate. Invoking the initial goal may create an *unproductive stream* [199]: repeated applications of `force` will never produce an answer. This is the one and only source of non-termination; all other operations in our implementation are total. This property exemplifies Kleene's normal form theorem [121].

3.5.3. microKanren Syntax. We eschew here an abstract syntax parameterized for this portion of these languages. Instead, we directly express programs in the particular, slightly more cumbersome concrete syntax of our translation to the embedded Racket implementation. We implement language constructs one at a time and we also provide interstitial examples. We believe this code to be sufficiently similar to Prolog for the practiced logic programmer and we could nearly compile programs in this example's language's concrete syntax to pure Prolog programs, although with important differences in behavior. We discuss the particular implemented search behavior in parallel with our development of the implementation.

3.6. Finite, Depth-first Search microKanren Implementation

Our initial embedding is similar to Kiselyov's [118], in that it implements a depth-first search that only works for finite search trees. We proceed to define and describe five basic components of our embedding. These are: two basic goals `succeed` and `fail` that respectively

```
(define ((succeed) S/c) (list S/c))
(define ((fail) S/c) '())
```

Listing 3.18. Definitions of `(succeed)` and `(fail)` goals

succeed and always fail; the binary goal constructors `disj` and `conj` that represent the disjunction and conjunction of two predicates, and the goal constructor `call/fresh` that implements the existential closure of a single variable.

3.6.1. `(succeed)` and `(fail)` goals. Our language includes two atomic goals `(succeed)` and `(fail)`, that unconditionally succeed and fail, respectively. The former is logically equivalent to an empty clause in Prolog, and we treat the latter as a canonical unsatisfiable goal. We distinguish between the primitive `(succeed)` and `(fail)` *goals* of Listing 3.18, and succeed and fail the *atomic constraints* of Section 3.3.1.2. Both sets can coexist, provided we implement them with distinct names.

The Racket `#hasheqv(…)` value denotes a hash map; here the accumulated constraint. We once again use the Unicode "…" to note an elision.

```
> ((fail) `(,S0 . 0))
'()
> ((succeed) `(,S0 . 0))
'((#hasheqv((== . ()) …) . 0))
```

3.6.2. Constraint goal constructors. In Listing 3.19 we present the definition of `make-constraint-goal-constructor`, a function that defines the atomic constraints' shallow embeddings. The function takes a definition of the solver, `invalid?`, and the field of the constraint store to implement. This function returns the goal constructor implementing that class of atomic constraints. A goal constructor is a function that accepts (one assumes the appropriate number of) term arguments as a tuple. Being a *goal constructor*, the return value of this function is a goal. A goal accepts a state, and this state package contains an indexed constraint store that's of primary interest here. To make this discussion more concrete, we will discuss in particular the implementation of the goal constructor `==`.

```
(define (((make-constraint-goal-constructor invalid? key) . ts) S/c)
  (let ([S (hash-update (car S/c) key ((curry cons) ts))])
    (if (invalid? S) '() (list `(,S . ,(cdr S/c))))))
```

LISTING 3.19. Definition of `make-constraint-goal-constructor`

Given say, two terms `u` and `v`, the goal constructor `==` then returns a function expecting `S/c`. This function is a goal. When executed, this goal extracts the state `S` (the first element of the pair `S/c`) and updates the state's field for `==` by adding the pair of the two terms `u` and `v`. The remainder of `==`'s definition relies on the underlying solver. If the solver succeeds on the augmented constraint, we create a new state by adding the current counter, and make a stream with only that state. We use `list` to construct singleton streams, and **`quasiquote`** and **`unquote`** to construct states. If `unify` returns `#f`, we return `()`, the empty stream.

We will see that `call/fresh`, `conj`, and `disj` are also goal constructors. The last microKanren operator, `call/initial-state`, is not a goal constructor. Instead, it executes a goal and may yield a list of states. For the time being though, we can explicitly invoke our goals in the initial state.

With only the goal constructor `==` (or with just atomic constraint goal constructors), the result of invoking any goal with the initial state is a mature stream. In fact, the result is a mature stream of length zero or one. Either the stream is empty, indicating for instance that the two terms are not equivalent, or the stream is non-empty and indicates the terms are equivalent.

```
> ((== '#t 'z) `(,S0 . 0))
'()
> ((== '#t '#t) `(,S0 . 0))
'((#hasheqv((== . ((#t #t))) …) . 0))
> ((== '(#t . #f) '(#t . #f)) `(,S0 . 0))
'((#hasheqv((== . (((#t . #f) (#t . #f)))) …) . 0))
```

For the moment, no matter what terms we unify, the constraint represents only the cumulative success or failure. For more expressive answers our logic language needs variables.

3.6.3. `call/fresh`. The syntax of Prolog implicitly introduces new logic variables before unifying with clauses' heads. Unlike Prolog, our languages demand the user introduce new logic variables explicitly, in an action separate from unification. The `call/fresh` goal constructor scopes a new logic variable over a goal. Our embedding uses the host language's lexical binding structure to introduce the variable scope and its function application to associate the new host lexical variable with the (representation of) the new logic variable. To this end, `call/fresh` takes as its argument a **λ** abstraction over a goal[10]. This **λ** expression `f` binds a logic variable to the goal-scoped lexical variable. The host's variable shadowing ensures variables' names are unambiguous in context. Without this shadowing, an embedding would have to explicitly represent freshness and uniqueness and maintain invariants on logic variables.

```
#| (Var → Goal) → Goal |#
(define ((call/fresh f) S/c)
  …)
```

As we know, the logic variables are an enumerable set X away from the ground terms $\mathscr{G}$. The function `var` enumerates X. The state's counter is the next index into the enumeration of X. Invoking `var` creates the next variable from `c`. The expression `(f (var c))` evaluates to a goal. The resultant goal is then invoked in a newly created state with the present constraint store and an incremented index.

```
#| (Var → Goal) → Goal |#
(define ((call/fresh f) S/c)
  (let ((c (cdr S/c)))
    ((f (var c)) `(,(car S/c) . ,(+ c 1)))))
```

The next example demonstrates that programs' terms can now contain logic variables.

```
> ((call/fresh (λ (x) (== x 'a))) `(,S0 . 0))
'((#hasheqv((== . ((0 a))) …) . 1))
```

[10]The `call/fresh` operator's argument should specifically always be a **λ** expression. Its body is either a goal expression, or nearly so but for free variables.

3.6.4. `conj` and `disj`. All programs in the language developed thus far have at most one atomic constraint. The binary goal combinators `disj` and `conj` permit composite goals that express the disjunction or conjunction of their arguments.

```
#| Goal × Goal → Goal |#
(define ((disj g1 g2) S/c) ($append (g1 S/c) (g2 S/c)))

#| Goal × Goal → Goal |#
(define ((conj g1 g2) S/c) ($append-map g2 (g1 S/c)))
```

We define `disj` and `conj` in terms of two other functions, `$append` and `$append-map`, that we define in Section 3.7.1. The following examples demonstrate `disj` and `conj` in combination with the goal constructors from before.

```
> ((disj
     (call/fresh (λ (x) (== 'z x)))
     (call/fresh (λ (x) (== '(s z) x))))
   `(,S0 . 0))
'((#hasheqv((== . ((z 0))) …) . 1)
  (#hasheqv((== . (((s z) 0))) …) . 1))
> ((call/fresh
     (λ (x)
       (call/fresh
         (λ (y)
           (conj
             (== y x)
             (== 'z x))))))
   `(,S0 . 0))
'((#hasheqv((== . ((z 0) (1 0))) …) . 2))
```

The streams computed by all programs in the language developed thus far will always be empty or answer-bearing; in fact, the streams will be fully computed. The result of an atomic constraint goal must be a finite list of length 0 or 1. If both of `disj`'s arguments are goals that produce finite lists, then the result of invoking `$append` on those lists is itself a finite list. If both of `conj`'s arguments are goals that produce finite lists, then the result of invoking `$append-map` with a goal and a finite list must itself be a finite list. If `call/fresh`'s argument `f` is a function whose body is a goal, and that goal produces a finite list, then `(call/fresh f)` evaluates to such a goal.

Invoking a goal constructed from these operators in the initial state returns a list of all successful computations, computed in a depth-first, preorder traversal of the search tree generated by the program. The list monad underlies this implementation. `$append-map` is bind, `$append` is mplus, and the calls to `list` and the primitive goals `(succeed)` and `(fail)` are `return` and `mzero`.

3.7. Depth-first search with infinite branches

In this second phase of implementing our embedding, we define two additional operators—`define-relation` that closes recursive definitions, and `call/initial-state` that runs a program. We will solve a problem of host language non-termination in conjunctions and disjunctions, and we redefine `$append` and `$append-map` to accommodate this solution. This second embedded implementation resembles those of Seres [190], Spivey and Seres [202], and Seres et al. [191] with several modifications and additions to guarantee termination in a call-by-value host.

3.7.1. Recursion and `define-relation`. We will enrich our implementation to allow recursive relations. Much of logic programming's power comes from writing relations that refer in their definitions to themselves (e.g. `append`) or to one another. At present there are several obstacles. Suppose we used **`define`** to build a function `peano` that purports to be the embedding of a relation that holds for a particular encoding of Peano numbers.

```
(define (peano n)
  (disj
    (== n 'z)
    (call/fresh
      (λ (r)
        (conj
          (== n `(s ,r))
          (peano r))))))
```

What happens when we use the `peano` relation in the program below? One would hope to generate some Peano numbers.

```
> ((call/fresh
     (λ (n)
       (peano n)))
   `(,S0 . 0))
```

We invoke `(call/fresh …)` with an initial state. Invoking that goal creates and lexically binds a new fresh variable over the body. The body, `(peano n)`, evaluates to a goal that we pass the state `(#hasheqv() . 0)`. This goal is the disjunction of two subgoals. To evaluate the `disj`, we evaluate its two subgoals, and then call `$append` on the result. The first evaluates to `((#hasheqv((== . ((0 z))) …) . 1))`, a list of one state.

Invoking the second of the `disj`'s subgoals however is troublesome. We again lexically scope a new variable, and invoke the goal in the body with a new state, this time `(##hasheqv() . 2)`. The `conj` goal has two subgoals. To evaluate these, we run the first goal in the current state, which results in a stream. We then run the second of `conj`'s goals over each element of the resulting stream and return the result. Running this second goal begins the whole process over again. In a call-by-value host, this execution won't terminate. Simply using **`define`** in this manner will not suffice to implement relations.

We instead introduce the `define-relation` operator. This operator permits recursive relations, and with multiple uses of `define-relation` we can create mutually recursive relations[11]. Unlike other operators of Section 3.5, `define-relation` is a macro. We do implement `define-relation` in terms of Racket's **`define`**.

```
(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) S/c) (delay/name (g S/c))))
```

This macro expands a name, arguments, and a goal expression into a **`define`** expression with the same name and number of arguments and whose body is a goal: it takes a state and returns a stream. Unlike the other goals we've seen before, this goal returns an immature

[11]For predicates undefined with `define-relation`, our embedding behaves like Prolog's with the `unknown` flag set to its default value, `error`. In our embedding this error comes from the host language.

stream. When given a state `S/c`, this goal returns a promise that evaluates the original goal `g` in the state `S/c` when forced, returning a stream. A promise that returns a stream is itself an immature stream.

`define-relation` does two useful things for us: it adds the relation name to the current namespace, and it ensures that the function implementing our relation is total. It turns out that we will *never* re-evaluate an immature stream. Unlike **`delay`**, **`delay/name`** doesn't *memoize* the result of forcing the promise, so it is like a "by name" variant of **`delay`**[12]. However, like the promises **`delay`** creates, our promises are evaluated at most once. The garbage collector can then consume used, discarded promises. This is a property of microKanren rather than something built into **`delay/name`**.

We are forced to implement `define-relation` as a macro so the expression `g` is not be evaluated prematurely: the objective is to delay the invocation of `g` in `S/c`. In a call-by-value language, a function would (prematurely) evaluate its argument and will not delay the computation. We revisit the `peano` example, this time using `define-relation`. Relation invocations must now terminate. Instead, the goal `(peano n)`, when invoked, immediately returns an immature stream.

```
(define-relation (peano n)
  (disj
    (== n 'z)
    (call/fresh
      (λ (r)
        (conj
          (== n `(s ,r))
          (peano r))))))
```

We can also write recursive relations whose goals quite clearly will never produce answers.

[12]We could have used `(λ () …)`, procedure invocation, and `procedure?` rather than **`delay/name`**, `force`, and `promise?`. Constructing a procedure with **`λ`** delays evaluation, and then testing `procedure?` suffices. We prefer Racket's special-purpose primitives because we shouldn't be testing for just any procedure. Without adding and checking for a tag, we cannot know if a given procedure represents a delay. However, implementers targeting other languages can use anonymous procedures if these more precise primitives aren't available.

```
(define-relation (unproductive n)
  (unproductive n))
```

We now introduce `$append` and `$append-map`. Their definitions are like those of `append` and `append-map`, standard list functions in many languages (e.g. Scheme [196]) but augmented with support for immature streams.

```
(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append (force $1) $2)))
    (else (cons (car $1) ($append (cdr $1) $2)))))
```

If the recursive argument to `$append` is an immature stream, we return an immature stream, which, when forced, continues appending the second to the first. Likewise, in `$append-map`, when `$` is an immature stream, we return an immature stream that will continue the computation but still forcing the immature stream[13].

```
#| Goal × Stream → Stream |#
(define ($append-map g $)
  (cond
    ((null? $) '())
    ((promise? $) (delay/name ($append-map g (force $))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))
```

After these changes, it's possible to execute a program and produce neither the empty stream nor an answer-bearing one. We might produce instead an immature stream.

```
> ((call/fresh
     (λ (n)
       (peano n)))
   `(,S0 . 0))
#<promise>
```

[13]In languages without macros, the programmer could explicitly add a delay at the top of each relation. This has the unfortunate consequence of exposing streams' implementation.

3.7.2. `call/initial-state`. When invoking the first goal from the initial state, we must do something special to resolve this. At a bare minimum, we expect to get at least one answer if our program expresses a *satisfiable* statement, and we can hope to get the empty list if there are no answers. The `call/initial-state` operator ensures that if we return, we return with a list of answers.

```
#| Maybe Nat⁺ × Goal ↛ Mature |#
(define (call/initial-state n g)
  (take n (pull (g `(,S0 . 0)))))
```

`call/initial-state` takes an argument `n` for the number of answers to retrieve. `n` may just be a positive natural number, in which case we return at most that many answers. Otherwise, it is `#f`, indicating microKanren should return *all* answers. The `call/initial-state` operator takes a goal as its second argument. The function `pull` consumes a stream and returns a mature stream, if `pull` in fact terminates. `pull` is a partial function; some streams are unproductive and cannot be matured. `pull` brings microKanren streams into the *delay monad* [29, 71]. Whereas before we always returned a list (representing a non-deterministic choice of answers), under this new model we have either no values, a value (possibly more than one) now, or we have something we can search later for a value. Since `pull` forces an actual value out of a promise if possible, it is akin to `run` in the delay monad.

```
#| Stream ↛ Mature |#
(define (pull $) (if (promise? $) (pull (force $)) $))
```

`take` consumes both the mature stream from `pull` and `n`, that argument dictating whether to return all, or just the first n elements of the stream. We can see `take` as the fusion of an operation to mature (if possible) a stream up to a prefix of length n, and an operation to take the first n elements off of such a prefix, if possible. `take` resembles run in the `list` monad. We can also see this operation as the *unfold* of some `pull`-like operation.

```
#| Maybe Nat⁺ × Mature ↛ List |#
(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
            (take (and n (- n 1)) (pull (cdr $)))))))
```

Our microKanren is now capable of creating, combining, and searching for answers in infinite streams. `take` and `call/initial-state` are also partial functions since they rely on `pull`. These are the only non-total functions in the microKanren implementation.

```
> (call/initial-state 2
    (call/fresh
      (λ (n)
        (peano n))))
'((#hasheqv (== . ((0 z))) …) . 1)
  (#hasheqv((== . ((1 z) (0 (s 1)))) …) . 2))
```

3.8. Interleaving, Complete Search

Although microKanren is now capable of creating and managing infinite streams, it doesn't manage them as well as one might hope. Consider executing the following program:

```
> (call/initial-state 1
    (call/fresh
      (λ (n)
        (disj
          (unproductive n)
          (peano n)))))
```

We should like the program to return a stream containing the `n`s for which `unproductive` holds, and in addition, the `n`s for which `peano` holds. We know from Section 3.7.1 that there are no `n`s for which `unproductive` holds, but infinitely many for `peano`. The stream should contain only `n`s for which `peano` holds. It's perhaps surprising, then, to learn that this program loops infinitely.

Streams that result from using `unproductive` will always be, as the name suggests, unproductive. When executing the program above, such an unproductive stream will be the recursive argument `$1` to `$append`. Unproductive streams are necessarily immature. According to our definition of `$append`, we always return the immature stream. When we force this immature stream, it calls `$append` on the forced stream value of (the delayed) `$1` and `$2`. Since `unproductive` is unproductive, this process continues without ever returning any of the results from `peano`. Such surprising results are not solely the consequence of goals with unproductive streams. Consider the definition of `church`.

```
(define-relation (church n)
  (call/fresh
    (λ (b)
      (conj
        (== n `(λ (s) (λ (z) ,b)))
        (peano b)))))
```

The relation `church` holds for Church numerals. Using a newly created variable `b`, it constructs a list resembling a λ-calculus expression whose body is the variable `b`. It uses `peano` to generate the body of the numeral. We can thus use it to generate Church numerals in a manner analogous to our use of `peano`. Although the resulting stream from the program below is productive, it only contains those elements for which `peano` holds.

```
> (call/initial-state 3
    (call/fresh
      (λ (n)
        (disj
          (peano n)
          (church n)))))
```

Our implementation of `$append` in Section 3.7.1 induces a depth-first search. Depth-first search is the traditional search strategy of Prolog and can be implemented quite efficiently. Depth-first search is however an *incomplete* search strategy, and in our implementation some streams can reflect this by burying some answers infinitely deeply. The stream that results from a `disj` goal produces elements of the stream from the second goal only after exhausting the elements of the stream from the first.

As a result, even if answers exist microKanren may fail to produce them. We will remedy this weakness in `$append`, and provide microKanren with a simple complete search. We want microKanren to guarantee each and every answer should occur at a finite position in the stream. Fortunately, this doesn't require a significant change.

```
#| Stream × Stream → Stream |#
(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))
```

This one change to the `promise?` line of `$append` is sufficient to make `disj` *fair* and to transform our search from an incomplete, depth-first search to a complete one. When the recursive argument to `$append` is an immature stream, we return an immature stream which, when forced, continues with the *other* stream first. The stream `$2` may also be partially computed. If so, then `$append` will process `$2` until it reaches the immature stream at `$2`'s tail. The function `$append` will process this immature stream in the same way.

Our streams are either (potentially empty) lists of states in the case of a fully computed stream, or (potentially empty) improper lists of states with a promise in the final `cdr`, in the case of partially computed streams.

In the case that `$1` is fully computed, `$append` appends `$2` to `$1`. Fully computed streams are finite, so after producing the finite quantity of elements from `$1`, we can then produce elements from `$2`, if they exist.

In the second case, if `$1` is only partially computed, then it has some potentially-empty finite prefix. We append those elements to a promise that, when forced, will continue by `$appending` `$2` to the result of forcing the promise that was previously the last `cdr` of `$1`. The result of forcing this newly created promise, if `$2` is immature, will be another promise, this time with a waiting call to `$append` on the stream that results from forcing the original last `cdr` of `$1` and the stream that results from forcing `$2`. If `$2` is productive, it will mature

in a finite number of invocations (possibly 0, if it was mature to begin with). So if `$2` is productive, there can be only a finite number of finite prefixes of `$1` produced before `$2` matures.

Of course, the stream that results from `$appending` `$2` to `$1` may *itself* be an argument to a call to `$append`. The stream that results from the execution of a program is created by successively `$appending` smaller streams, either in evaluating a `disj`, or as used in the implementation of `conj`. The reasoning we use above holds for arbitrary streams, so taking answers from the returned stream amounts to a complete search for the program, as Rozplokhas et al. [185] also show.

```
> (call/initial-state 3
    (call/fresh
      (λ (n)
       (disj
         (peano n)
         (church n)))))
'((#hasheqv((== . ((0 z))) …) . 1)
  (#hasheqv((== . ((1 z) (0 (s 1)))) …) . 2)
  (#hasheqv((== . ((1 z) (0 (λ (s) (λ (z) 1))))) …) . 2))
```

This last change completes the definition of a constraint microKanren language. The complete search technique describes a kind of interleaving depth-first search [120][14]. Interestingly, we haven't reconstructed some particular, fixed, complete search strategy. Instead, the search strategy of microKanren programs is program- and query-specific. The particular definitions of a program's relations, together with the goal from which it's executed, both generate and dictate the order in which we explore the search tree. In other similar embeddings (e.g. Hinze [95], Kiselyov et al. [120], and Spivey and Seres [202]) relying on non-strict evaluation simplifies the implementation task. The standard, straightforward translation of their embeddings to a call-by-value host sacrifices some of the elegance of their implementations. Wadler et al.'s [218] standard "turn-crank" transformation to add lazy streams in an eager host adds more delays than necessary to retain completeness.

[14]Though similarly named, this is different from Meseguer's [156] "Interleaved Depth-first Search".

Spivey and Seres implement a breadth-first search—also a complete search—but this implementation requires a somewhat more sophisticated transformation than does ours and constrains the search beyond what is strictly necessary to achieve completeness. We achieve a simpler implementation of a complete search by using the delays as markers for interleaving our streams. As advertised, we can use microKanren to write real programs. The below expansion of the basic `append` relation into a microKanren program.

```
(define (append l s o)
  (λ (S/c)
    (delay/name
      ((disj
         (conj
           (== l '())
           (== s o))
         (call/fresh
           (λ (a)
             (call/fresh
               (λ (d)
                 (conj
                   (== l `(,a . ,d))
                   (call/fresh
                     (λ (r)
                       (conj
                         (== o `(,a . ,r))
                         (append d s r))))))))))
       S/c))))
```

At first blush it seems like simplifying the language so much places a burden on the language user both in writing programs and interpreting their results. microKanren may not be especially convenient or friendly for the working logic programmer, but it is a serviceable logic programming language implementable in a call-by-value language and requiring only a minimal group of features from its host. We will see in Section 3.10 a handful of straightforward macros that both provide a nicer surface syntax in which to write programs, and also recover the pre-existing surface syntax of programs in miniKanrens with constraints. The push to more pure relational programming is one force driving the need for new constraints. We close this chapter, however, by introducing helpful auxiliary non-logical and extra-logical extensions to the core language.

3.9. Impure Extensions

The microKanren presented in Section 3.10 is a complete purely declarative logic programming language. In this section we add some of Prolog's impure operators for additional control mechanisms.

Naish shows that Prolog's cut (`!`) is a combination of a deterministic if-then-else and don't-care nondeterminism [162]. We implement these as separate operators, `ifte` and `once`, inspired by Kiselyov et al. [120]; `ifte` is also similar to the `cond/3` found in several Prologs [18].

The operator `ifte` takes three goals as arguments: if the first succeeds, then we execute the second against the result of the first and discard the third. If the first fails, then we execute the third and discard the second. Providing the identifier `loop` makes the body of the **`let`** recursively scoped. **`let`** scopes this name over the **`let`**'s body. If `(g0 S/c)` returns a promise, we don't want to immediately continue forcing it. That might make our search incomplete again—`$` might not be productive. So instead, we return a promise, which, when forced, itself forces `$` and then tests the value against our three cases.

```
(define ((ifte g0 g1 g2) S/c)
  (let loop (($ (g0 S/c)))
    (cond
      ((null? $) (g2 S/c))
      ((promise? $) (delay/name (loop (force $))))
      (else ($append-map $ g1)))))
```

```
> (call/initial-state #f
    (call/fresh
      (λ (q)
        (ifte (== 'a 'b) (== q 'a) (== q 'b)))))
'((#hasheqv((== . ((0 b))) …) . 1))
```

`once` takes a goal `g` as an argument and returns a new goal as its result. This resulting goal behaves like `g` except that, where `g` would succeed with a stream of more than one element, this new goal returns a stream of only the first.

```
(define ((once g) S/c)
  (let loop (($ (g S/c)))
    (cond
      ((null? $) '())
      ((promise? $) (delay/name (loop (force $))))
      (else (list (car $))))))
```

For the same reasons as `ifte`'s definition, `once`'s definition creates a function named `loop` and uses it in the second clause of the **`cond`**.

```
> (call/initial-state #f
    (call/fresh
      (λ (q)
        (once (peano q)))))
'((((0 . z)) . 1))
```

Together, these two operators provide the power of Prolog's cut. Use of these operators can increase the efficiency of our programs. These operators, however, can mangle the connection between logic programming and logic, ultimately costing us some of the flexibility of logic programs that `append` demonstrates.

3.10. Recovering miniKanren

In this section, we describe how to in fact recover the initial miniKanren language. The microKanren implementation of `append` in Section 3.8 exemplifies why users might want a set of higher-level and more sophisticated operators with which to write programs and view the results. miniKanren programs are often composed of multiple relations much larger and more complicated than `append`. We layer the higher-level syntax of miniKanren (`fresh`, `conde`, `run`, `conda`, and `condu`) over microKanren via some straightforward macros. Goal constructors like **`==`** and the `define-relation` macro transfer directly.

3.11. miniKanren Implementation

As a first step to reconstructing miniKanren, we create operators `disj+` and `conj+` that allow us to write more than just the binary disjunction and conjunction of goals. The `disj+` (`conj+`) of a single goal is just the goal itself. For more than one goal, we recursively `disj` (`conj`) the first goal onto the result of the recursion. We use **`define-syntax`** and **`syntax-rules`** to implement recursive macros.

```
(define-syntax disj+
  (syntax-rules ()
    ((_ g) g)
    ((_ g0 g ...) (disj g0 (disj+ g ...)))))
```

```
(define-syntax conj+
  (syntax-rules ()
    ((_ g) g)
    ((_ g0 g ...) (conj g0 (conj+ g ...)))))
```

3.11.1. `conde` and `fresh`. With `disj+` and `conj+`, we are able to construct miniKanren's `conde` as a macro that merely rearranges its arguments. miniKanren's `conde` is the `disj+` of a sequence of `conj+`s:

```
(define-syntax-rule (conde (g0 g ...) (g0* g* ...) ...)
  (disj+ (conj+ g0 g ...) (conj+ g0* g* ...) ...))
```

We build the `fresh` of miniKanren, which introduces zero or more fresh variables, as a recursive macro using `call/fresh` and `conj+`:

```
(define-syntax fresh
  (syntax-rules ()
    ((_ () g0 g ...) (conj+ g0 g ...))
    ((_ (x0 x ...) g0 g ...)
     (call/fresh (λ (x0) (fresh (x ...) g0 g ...))))))
```

3.11.2. `run`**.** The last pure miniKanren form we reconstruct is `run`, the external interface that allows us to execute a miniKanren program. The `run` operator takes as arguments a positive natural number n or `#f`, indicating the number of answers to return (similar to `call/initial-state`); a query variable q (in parentheses); and a non-empty sequence of goal expressions.

miniKanren programs, like microKanren programs, may not terminate. Traditionally though, if the program does terminate, miniKanrens will format the returned answers in terms of the query variable and return them in a list. In constraint logic programming, answers are a collection of constraints called the "answer constraint". miniKanren simplifies answers and presents them with respect to the query variable. Our constraint logic programs can introduce a large number of auxiliary variables in the course of their execution. Rather than returning the values of all of these variables, the user will prefer to see the value of the query variable (and variables associated with it). This process is called *answer projection* [61, 112]. Our implementations, however, will not implement answer projection, subsumption, or a number of other nice features in presenting the simplified result. To implement these features across a family of languages parameterized by their constraint domain, we would need to describe these mechanisms generically, rather than just the specific, particular instances implementers have constructed in the past. Instead, we simply return the answer constraint, which is in fact a multi-set of the collected constraints. Eschewing this additional feature, though, makes implementing a `run` interface especially straightforward.

```
(define-syntax-rule (run n (q) g0 g ...)
  (call/initial-state n (fresh (q) g0 g ...)))
```

In fact, we can expand a miniKanren `run` expression into calls to the microKanren primitives and helper functions in terms of which they are defined. Listing 3.20 shows the expansion of Listing 1.1.

One imagines adding even more succinct syntax. Perhaps some `define-relation/conde` and `define-relation/matche` (see Keep et al. [116]) forms that would take a homogenized head and a sequence of clause bodies and expand to their obvious respective underlying

```
> (call/initial-state #f
    (call/fresh
      (λ (q)
        (call/fresh
          (λ (l)
            (call/fresh
              (λ (s)
                (conj
                  (== `(,l ,s) q)
                  (append l s '(t u v w x)))))))))))
```

LISTING 3.20. An expansion of the `append` invocation of Listing 1.1

structures. One imagines perhaps a special form of equality constraints that permit embedded term-structural (primitive) constraints. Such an extended equality constraint might then expand to sequences of primitive constraints. We could embed these and other additional syntactic forms similarly.

3.12. Impure miniKanren extensions

We introduce here the language of full miniKanren programs, with additional impure operators. We recover the impure miniKanren operators `conda` and `condu`, which provide committed choice and committed choice with a "don't-care" nondeterminism, respectively. When we add the impure miniKanren extensions `conda` and `condu` to our model of these negated literals, it looks like we get something like *extended logic programs* with constraints.

As a first step we implement `ifte*`, which nests `ifte` expressions. It takes a sequence of lists containing two goal expressions each, followed by a single goal expression at the end and transforms these into a sequence of nested `ifte` expressions, using the last goal as the final `ifte`'s **`else`** clause.

```
(define-syntax ifte*
  (syntax-rules ()
    ((_ g) g)
    ((_ (g0 g1) (g0* g1*) ... g)
     (ifte g0 g1 (ifte* (g0* g1*) ... g)))))
```

With this, we can implement `conda` and `condu` as macros. `conda` takes a sequence of sequences of two or more goal expressions each, except the last which is a sequence of one or more goals. With `conj+`, we transform this syntax into an `ifte*` expression:

```
(define-syntax-rule (conda (g0 g1 g ...) ... (gn0 gn ...))
  (ifte* (g0 (conj+ g1 g ...)) ... (conj+ gn0 gn ...)))
```

We implement `condu` by adding `once` to each first element of each sequence, and building a `conda` from the result:

```
(define-syntax-rule (condu (g0 g1 g ...) ... (gn0 gn ...))
  (conda ((once g0) g ...) ... ((once gn0) gn ...)))
```

In the next chapter we will see some of the example use-cases facilitated by these constraint miniKanren languages, as well as demonstrating some steps forward and improvements in relational programming techniques in miniKanren. In doing so, we will clarify some heretofore nebulous aspects of common programming practices with constraint miniKanren languages.

Chapter 4 Examples, Uses and Techniques

In describing the constraint systems of Chapter 3, we have exhibited some typical use cases. Constraints such as those defined in Chapter 3, when used in concert with the constraint-independent portion of the language implementation described in Section 3.10, enable solutions of novel programming exercises while clarifying and simplifying the solutions to some previously solved problems.

In this chapter we present and explain several larger uses of such constraints in miniKanren logic programs for novel or interesting problems. In several cases we contrast our present solutions to a less desirable recourse in the absence of readily-definable constraints. We do not suggest that in the absence of our novel constraints, these problems are not amenable to (constraint) logic programming. Nor, by "readily-definable", we do not mean to say that such constraints were impossible absent our work. Instead, we suggest just that the engineering effort otherwise required in implementing such constraints makes each less likely to be implemented and made available to the programmer. In the course of these examples, we will indicate constraints that were rarely if ever implemented in miniKanrens prior to the work of this thesis or otherwise uncommon to CLP languages. Many examples of this chapter use miniKanren's advanced pattern-matching syntax extension `matche` from Keep et al. [116].

4.1. Quine and quine-like program generation

Quine generating is one of the most frequently-demonstrated miniKanren programming examples. A *quine* [98], or "self-replicating program" is a program whose output is its own listings. Such a program is a fixed point of its evaluator (or its execution environment) when taken as a function from programs to outputs. The "quine" entry in the *New Hacker's Dictionary (Jargon File)* [174] mentions the following as a classic:

```
((lambda (x)
  (list x (list (quote quote) x)))
 (quote
   (lambda (x)
     (list x (list (quote quote) x)))))
```

A constraint logic programmer can implement a languages' interpreter as a computable relation between expressions and their values. With this definition at hand, the programmer writes a relatively short query for a fixed point of the relation, and the answer quickly returns. We describe in this section how constraints aid a programmer in implementing quine generators and related programs.

Querying a relation for its fixed point is useful well beyond generating quines. Indeed, we first demonstrate this general technique using the `lengtho` relation of Listing 4.1. We use this as a preliminary example before moving to the more complicated interpreter program. The binary `lengtho` relationship holds between a list and its length in little-endian binary. We modified this example from Chapter 7 of *The Reasoned Schemer, 2nd Ed* [65] and it uses the miniKanren arithmetic suite also seen in Kiselyov et al. [119].

```
(define-relation (lengtho l n)
  (conde
    ((nullo l) (== '() n))
    ((fresh (a d)
       (== l `(,a . ,d))
       (fresh (res)
         (pluso '(1) res n)
         (lengtho d res))))))
```

LISTING 4.1. The `lengtho` relation

We query this relation in Listing 4.2 for three lists that are backwards-binary number encodings of their own length.

```
> (run 3 (q) (lengtho q q))
(() (1) (0 1))
```

Listing 4.2. A use of the `lengtho` relation

The evaluation relationship is more sophisticated than the list-length relationship. Consider the Racket implementation of a functional interpreter for a Scheme-like language capable of expressing quines. We will not recapitulate here background in designing interpreters, but there is very little here out of the ordinary. We modified this functional implementation from Indiana University's C311/B521 course.

Programmers implementing relational interpreters benefit from logic languages with auxillary constraints because the programmer can write directly in the domain of discourse—here the interpreted programming language—and in doing so the relational version can closely resemble the functional version.

We do not dwell on the particulars of translating a functional program to a miniKanren relational program or the particular considerations for relational interpreters; a reader interested in the latter should consult Byrd et al. [25, 27]. For our purposes it is sufficient to see the miniKanren interpreter for this same language closely resembles the functional implementation, with the addition of several constraints that constrict the domain of the function. Instead, we highlight the benefits of constraints like those enabled by our constraint systems provide when building relational interpreters.

Under a straightforward encoding of a small Scheme-like language's interpreter, equations alone are insufficient to restrict the interpreted languages' variables to particular subsets of terms (e.g. restricting a **λ** expression's binding to the set of the symbols). We necessarily use a first-order representation of closures and environments for interpreters written in miniKanren. We implement closures as lists with tags, to distinguish them

```
(define (lookup vars vals y)
  (match-let ((`(,x . ,vars^) vars)
              (`(,v . ,vals^) vals))
    (cond
      ((equal? x y) v)
      ((not (equal? x y)) (lookup vars^ vals^ y)))))

(define (valsof args vars vals)
  (cond
   ((equal? args '()) '())
   (else (let ((v (valof (car args) vars vals))
               (vs (valsof (cdr args) vars vals)))
           `(,v . ,vs)))))

(define (eval exp)
  (valof exp '() '()))

(define (valof exp vars vals)
  (match exp
    [`,exp #:when (symbol? exp) (lookup vars vals exp)]
    [`(λ (,x) ,b) #:when (symbol? x)
     `(closure ,x ,b ,vars ,vals)]
    [`(quote ,v) v]
    [`(list . ,args) (valsof args vars vals)]
    [`(,rator ,rand)
     (match-let ((`(closure ,x ,b ,vars^ ,vals^) (valof rator vars vals))
                 (a (valof rand vars vals)))
       (valof b `(,x . ,vars^) `(,a . ,vals^)))]))
```

LISTING 4.3. Functional interpreter for a Scheme-like language that expresses quines.

from the values of actual list expressions values. Here, `absento` constraints exclude raw closures from the evaluation relation. The `absento` constraints are also critical to preventing `λ` expressions from capturing primitives of the language.

We express the binary `eval` relation as a specialized version of a more general quartery relation `valof`. These auxiliary parameters of `valof` are environments, initially empty. The quaternary relation `valof` describes the relationship between programs in an empty environment and those programs' values. We often use the specialized `eval` relation when writing queries.

```
(define-relation (lookup x vars vals o)
  (fresh (y vars^ v vals^)
    (== `(,y . ,vars^) vars)
    (== `(,v . ,vals^) vals)
    (conde
      [(== x y) (== v o) (listo vars^) (listo vals^)]
      [(=/= x y) (lookup x vars^ vals^ o)])))

(define-relation (valof exp vars vals o)
  (conde
    [(symbolo exp) (lookup exp vars vals o)]
    [(fresh (x b)
       (== `(λ (,x) ,b) exp)
       (absento 'λ vars)
       (symbolo x)
       (== `(closure ,x ,b ,vars ,vals) o))]
    [(== `(quote ,o) exp)
     (absento 'quote vars)
     (absento 'closure o)]
    [(fresh (es)
       (== `(list . ,es) exp)
       (absento 'list vars)
       (valsof es vars vals o))]
    [(fresh (rator rand)
       (== `(,rator ,rand) exp)
       (=/= rator 'quote) (=/= rator 'list)
       (fresh (x b vars^ vals^ a)
         (valof rator vars vals `(closure ,x ,b ,vars^ ,vals^))
         (valof rand vars vals a)
         (valof b `(,x . ,vars^) `(,a . ,vals^) o)))]))

(define-relation (valsof es vars vals o)
  (conde
    [(== `() es) (== '() o)]
    [(fresh (e es^)
       (== `(,e . ,es^) es)
       (fresh (v vs)
         (== `(,v . ,vs) o)
         (valof e vars vals v)
         (valsof es^ vars vals vs)))]))
```

Listing 4.4. A relational miniKanren interpreter

```
(define-relation (eval exp o)
  (valof exp '() '() o))
```

Listing 4.5. Definition of the help relation `eval`

```
> (run 3 (q) (eval q q))
```

Listing 4.6. Querying for quines

With this relational interpreter, we can also ask and answer more sophisticated questions about program relationships beyond quines. The disequality constraints in some of the next several queries describe disequalities between programs of the interpreted language. For the first of these examples, we generate *twines*. A twine, or "twin quine" is a program that, when evaluated, produces a program that, when evaluated, produces the original program. By including this disequality, we query specifically for twines that are not themselves quines.

```
> (run 1 (p) (fresh (q) (=/= p q) (eval p q) (eval q p)))
```

Such cycles of evaluation generalize to thrines and beyond in a natural way. As a second class of related examples, consider the below query for a 3-cycle of programs `p`, `q`, and `r` for which evaluating the current program on the next program, yields the prior program.

```
> (run 1 (p q r)
    (eval `(,p ,q) r) (eval `(,q ,r) p) (eval `(,r ,p) q))
```

This query is trivially satisfiable; `(quote quote)` evaluates in Scheme to `quote`. To achieve a more interesting result, we once again include disequality constraints between interpreted-language programs, as in the twines example.

```
> (run 1 (p q r) (=/= p q) (=/= q r) (=/= r p)
    (eval `(,p ,q) r) (eval `(,q ,r) p) (eval `(,r ,p) q))
```

This query responds with results, which we omit here for space but include in Appendix C. What is important here is not the queries' precise answers, but that such answers exist and our ability to readily express and modify queries to find them. As far as we know,

there is no canonical name for such program cycles with this property. We did not know that such a cycle existed until we experimented and queried for one. The ability to rapidly prototype and test new constraints begets platforms for experimenting with such queries.

We consider here a third example querying and finding "mirror-image" programs. We wrote a relational interpreter almost identical to that of Listing 4.4, but with the language's syntax reversed. That is, the program `((λ (arg) (list arg)) (quote cat))`, in this reversed language we write `((cat etouq) ((arg tsil) (arg) adbmal))` for that same function. Naturally enough, we named this relation `lave`, which calls to `folav`. The one interesting difference is that testing for `tsil` expressions now requires a relation `tsil-reporpo`, a mirrored "proper list" relation. The mirrored interpreter's definition is also in Appendix C. With both of these relational interpreters together, we can query for programs that are well-formed in both languages.

```
> (run 3 (q) (fresh (a b) (eval q a) (lave q b)))
('etouq
 ((λ (_0) adbmal) (sym _0))
 ((λ (adbmal) adbmal) (λ (λ) adbmal)))
```

One *can* write a quine-generating relational interpreter in miniKanren using only equality constraints. Such an interpreter, however, does not carry that superficial resemblance to the functional version as does `lengtho` to `length`. The purely-equational definition of Listing 4.7 is due to Nada Amin and Tiark Rompf [7].

The interpreter itself and the environment lookup mechanisms seem superficially similar to the relational interpreter of Listing 4.4. Subtle differences here, however, betray the additional complexities hefted upon the implementation in a language with so spartan a constraint set. Firstly, consider the use of `=/=` in `lookup`. Without primitive disequality constraints, the interpreter writer must implement disequalities relationally. This relation cannot implement recursively the general miniKanren disequality constraints, as in many cases this would fail to terminate. Since this particular interpreter uses disequalities to

constrain only variables of the interpreted language, it suffices to implement disequalities over such variables. We encode these variables as Peano numbers as in Section 3.7.1 on page 83.

```
(define-relation (lookup e i v)
  (fresh (j vj er)
    (== e `((,j . ,vj) . ,er))
    (conde
      ((== i j) (== v vj))
      ((=/= i j) (lookup er i v)))))

(define-relation (valof e t v)
  (conde
    ((fresh (idx)
       (== t `(x . ,idx))
       (lookup e x v)))
    ((fresh (idx t0)
       (== t `(λ (x . ,idx) ,t0))
       (== v `(closure ,e ,idx ,t0))))
    ((fresh (t0)
       (== t `(quote ,t0))
       (== v `(code ,t0))))
    ((fresh (t₁ t₂ e0 idx0 t0 v₂)
       (== t `(,t₁ ,t₂))
       (valof e t₁ `(closure ,e0 ,idx0 ,t0))
       (valof e t₂ v₂)
       (valof `((,idx0 . ,v₂) . ,e0) t0 v)))
    ((fresh (t₁ t₂ c₁ c₂)
       (== t `(list₂ ,t₁ ,t₂))
       (valof e t₁ `(code ,c₁))
       (valof e t₂ `(code ,c₂))
       (== v `(code (,c₁ ,c₂)))))))
```

Listing 4.7. A purely-equational relational interpreter in miniKanren

Without `absento` to prevent the interpreter from generating closures in the initial program, we must instead resort to tagging all terms and values of the language; in order to distinguish a term from a value, we implement special relations for each set. In Listing 4.9 we present the implementation of `val?`. `val?` relies on `env?`, `nat?`, and `expr?`. We include these with the full implementation in Appendix C, but the implementation as expressed so

```
(define-relation (=/= n₁ n₂)
  (conde
    [(fresh (pn₂)
       (== n₂ `(s . ,pn₂))
       (== n₁ '()))]
    [(fresh (pn₁)
       (== n₁ `(s . ,pn₁))
       (== n₂ '()))]
    [(fresh (pn₁ pn₂)
       (== n₁ `(s . ,pn₁))
       (== n₂ `(s . ,pn₂))
       (=/= pn₁ pn₂))]))
```

LISTING 4.8. An unequal-variables relation for the interpreter of Listing 4.7

far sufficiently demonstrates the disconnect between the functional host language's implementation and the encumberances of the relational language with only equality constraint primitives.

```
(define-relation (val? o)
  (conde
    ((fresh (e idx t)
       (== `(closure ,e ,idx ,t) o)
       (env? e)
       (nat? idx)
       (expr? t)))
    ((fresh (t)
       (== `(code ,t) o)
       (expr? t)))))
```

LISTING 4.9. A value relation for the interpreter of Listing 4.7

Moreover, these design requirements we impose on the interpreter of Listing 4.7 restrict its language to 2-lists. 2-lists are sufficient for implementing quines, but this is a more limited language than that of Listing 4.4 and we cannot extend its lists to lists of arbitrary length without further complicating the language.

4.2. Imperative Language Interpreters and Program Inversion

Our implementation technique for languages' interpreters also accommodates languages with imperative control features and mutable state. We exhibit in this section how constraints also aid the programmer in implementing interpreters for languages with these features. The examples of this section, chiefly based around a relational interpreter for a Flowchart [74] implementation of the MP (miniPascal) language [192], come from unpublished work with Dan Friedman & Robert Glück in 2013. We defer the full implementation of this interpreter to Appendix C and present here a few illustrative components. This first series of example queries to this interpreter are tree traversals. The model of relational language implementation admits reversible programming; that is, we can execute standard MP programs both forwards and backwards. These examples too make heavy use of the miniKanren constraints we can readily define in constraint microKanren.

The program `preorder-traverse` performs a preorder traversal over a binary tree with data on internal nodes and collects those data as the result. The program executes by first initializing the program's local variables: the current `traversal` and the `todo` stack are set to empty, and the `incomplete?` flag is set to `'(true)`. The program executes through a single main loop. While the tree traversal is still incomplete, if the current node is a non-leaf, then proceed down the left branch, and push the center and right nodes onto the `todo` stack. If the current node is a leaf node and the todo stack is non-empty, then add the center node of top entry in the to-do stack to the traversal, set the current tree to the right-hand side, and loop to continue. This program considers an interior node without a right and left child bad data, and in that case the program reports an error. When the tree is empty, we set the `incomplete?` flag to false, which terminates the loop.

Behaviorally, our relational implementation of the MP interpreter differs from the corresponding functional implementation in that, upon a state change, we `cdr` to the variable in question and then rebuild the front of the environment. The relational implementation maintains the ordering of variables in the environment. We could also have split the environment to begin with and mandated that globals begin bound. This may aid program

```
(define preorder-traverse
  '((:= incomplete? '(true))
    (:= traversal '())
    (:= todo '())
    (while flag
      ((if tr
         ((:= todo (cons (cdr tr) todo))
          (:= tr (car t)))
         ((:= traversal (cons tr traversal))
          (if todo
            ((if (car todo)
                 ((:= traversal (cons (car (car todo)) traversal))
                  (:= tr (cdr (car todo)))
                  (:= todo (cdr todo)))
                 ((:= incomplete? '())
                  (:= traversal (cons 'Error traversal)))))
            ((:= tr '())
             (:= incomplete? '()))))))))))
```

LISTING 4.10. Preorder tree traversal program in MP

generation when running queries with all fresh variables. We demonstrate these programs executing in an inverse, non-standard mode, but they can also be executed as expected in the usual forward mode.

```
> (run 5 (q)
    (run-programo
      '(tr)
      '(todo traversal incomplete)
      preorder-traverse
      q
      '((todo ()) (traversal (7 6 5 4 3 2 1)) (incomplete? ()) (tr ()))))
(((1 2 3 4 5 6 . 7))
 ((1 2 (3 4 . 5) 6 . 7))
 (((1 2 . 3) 4 5 6 . 7))
 (((1 2 3 4 . 5) 6 . 7))
 ((((1 2 . 3) 4 . 5) 6 . 7)))
```

LISTING 4.11. Tree traversals of the `traverse` program of Listing 4.10 in the MP interpreter

We leave the implementation of the `traverse-graph` program to Appendix C, as this imperative program is comparatively large. To use the program, we initialize sets of global and local variables, and traverse a directed graph, represented as an association list of nodes to neighbors. If the graph contains disconnected components or cycles, our program instead reports those errors. From the first query of Listing 4.12, we see that we can produce a graph traversal given a graph. From the second we see also that, given part of a graph and a termination state, we can create completions of that graph so the entire graph terminates at that final state.

```
> (run 1 (q)
    (run-programo '(w g) '(c cc pw x cont badflag)
     traverse-graph
     '((A) ((A B C) (B D) (C) (D A C)))
     q))
(((c A)
  (cc (B C))
  (pw (A))
  (x ((A B C) (B D) (C) (D A C)))
  (cont ())
  (badflag ())
  (w ())
  (g ((A B C) (B D) (C) (D A C)))))
> (run 5 (a d)
    (run-programo '(w g) '(c cc pw x cont badflag)
     `((,a ,d) ((A B C) (B D) (C) (D A C)))
     '((c ())
       (cc ())
       (pw ())
       (x ())
       (cont ())
       (badflag success)
       (w ())
       (g ()))))
((A B) (A C) (B D) (D A) (D C))
```

LISTING 4.12. Uses of a stateful graph traversal program to both traverse graphs and to generate parts of graphs from the state after traversal

Relational interpreters for languages with imperative constructs, such as MP, can provide other avenues to approach "inverse programming" and could be a worthwhile target for research in partial evaluation. Through our techniques for developing relational programs from functional ones, a developer might need only to write a properly constrained functional, "forward direction", version of their program, and rely on a multi-modal relational interpreter for that language to evaluate the inverse program, and rely on partial evaluation to eliminate much of the interpretive overhead.

4.3. Relational type-checking and inference

Another common set of miniKanren programming examples are multi-modal type checker/inferencer/inhabiters for various small programming languages. Depending on the mode in which we execute it, this single program can check if a program types at a given type, infer a given program's type, or find a program that inhabits a given type. Such programs are especially nice to implement in a logic language. Whenever we can directly express a judgment as a term structural relationship, such as the application case of Listing 4.13, then implementing that judgment is mostly translating the syntax. An implementation's components *not* directly transferred from judgments—for example, the structure of environments that allow type-checking `let`-bound polymorphic functions `let`-bindings[1]—can be more complex. In Appendix C we provide the remainder of the implementation, due in part to Spencer Bauman and presumably similar to Pan and Bryant's [169] approach.

In earlier implementations, miniKanren programmers often implemented a `not-in-envo` relation like that of Listing 4.15 that allowed us to specify that certain symbols were absent from the environment. This relation expresses an important property for correctly checking the types of functions shadowing primitive forms. This implementation has the unfortunate consequence though of *generating* particular environments in which the judgment holds, rather than representing them generally. miniKanren programmers have taken to instead

[1]Functions are otherwise monomorphic, and monotypic variables stand for a single, distinct type. Only `let` introduces polytypic variables. In a Haskell-like language, we could use higher rank types to achieve a similar effect for λs.

```
(define-relation (⊢ Γ e τ)
  (conde
    …
    ((fresh (rator rand)
       (== `(,rator ,rand) e)
       (fresh (τₓ)
         (⊢ Γ rator `(,τₓ → ,τ))
         (⊢ Γ rand τₓ))))))
```

LISTING 4.13. Application case in a miniKanren-based implementation of a type-checker

```
> (run* q
    (⊢ '() '(let ([f (λ (x) #t)])
               (if #t (f (f "cat")) (f #t))) q))
'(Bool)
```

LISTING 4.14. A miniKanren-based type-checker polymorphically typing a let-bound **λ** expression

```
(define-relation (not-in-envo x env)
  (conde
    [(== '() env)]
    [(fresh (y _ rest)
       (== `((,y ,_) . ,rest) env)
       (=/= y x)
       (not-in-envo x rest))]))
```

LISTING 4.15. Implementation of a `not-in-envo` relation for a type environment

ensuring this property using an `absento` constraint to exclude a given term from an environment structure. However, this has several deficiencies. An environment, a finite list of pairs, is structurally more complex than an arbitrary tree. An `absento` constraint can overly restrict the environment and exclude **`let`** or **`lambda`** from the environment altogether, rather than just as type variables. In principle, types and type variables should be of different sorts. The work-around in Section 4.1 was to redesign the interpreters' environment as a pair of lists. Moreover, `absento` constraints alone are also *insufficient* to properly constrain the environment's structure. A type judgment may still hold even if the environment is

constrained to an improper list; in fact miniKanren almost always generates partially-determined environments specified up to an improper list structure. In Listing 4.16, we implement the more precise `non-in-envo` constraint and a constraint interaction to forbid improper-list environments. This gives the benefits a constraint without the deficiencies of `absento`.

These environments could have been constrained to have an even more precise structure (e.g. the left of each pair could be constrained to a symbol). Surely, if having a boolean on the left-hand side makes an environment invalid, then that would be correct to specify. We could add this additional structure as another way for the environment relationship to fail to hold. Under the closed-world assumption, specifying additional failure cases is as easy as more tightly specifying the success. These design decisions *are* the constraint's specification. We chose to instead treat an environment pair beginning without a type variable as harmless noise-data. In our system the constraint writer explicitly states these decisions in the constraints' definitions. The CLP programmer separately describes `lookup`'s behavior on these strange environments.

```
#:recursive-predicates
(…
 (in-env?
  [(X (cons (cons X (cons _ '())) _)) true]
  [(X (cons (cons _ (cons _ '())) R)) (in-env? X R)]))
#:constraints
([(not-in-envo x ne) (in-env? x ne)]
 …
 [(improper-listo l) (equal? (cdr* l) '())])
#:rewrite-rules ()
#:failure-rules
([for-all ([improper-listo i] [not-in-envo x l])
          #:fail-when [(equal? (cdr* i) (cdr* l))]]
 …)
```

Listing 4.16. Implementing a `not-in-envo` constraint and its interactions

4.4. Relational Implementations of Natural Logics

In this section we exhibit exemplary Kanren implementations of proof search in a number of *natural logics* [15]. These logics range from the Aristotelian syllogistic to those with the reasoning power of full first-order logic and even beyond. They are *natural* in the sense they admit argument and proof roughly on the level of natural language structures themselves. This is an alternative to the familiar approach from most introductory logic courses: first translate an argument into a formal language, and then analyze the argument in that setting.

```
(define-relation (A φ Γ proof)
  (matche φ
    [(∀ ,a ,a) (== φ proof)] ; Axiom
    [,x (membero x Γ) (== proof `(,x in-Γ))] ; Lookup
    [(∀ ,n ,q) ; “Barbara” inference
     (fresh (p proof1 proof2)
       (== `((,proof1 ,proof2) => ,φ) proof)
       (A `(∀ ,n ,p) Γ proof1)
       (A `(∀ ,p ,q) Γ proof2))]))
```

LISTING 4.17. A `matche`-based miniKanren implementation of $\mathscr{A}$

Being declarative and logic-based, miniKanren makes constructing proof searches for these logics and experimenting with them straightforward. The implementations of this section illustrate benefits of new constraints for implementations of natural logics. We exhibit three models for declaratively implementing syllogistic logics:

(1) A “raw miniKanren” implementation encoding parts of the logic with the standard miniKanren constraints over the standard terms.

(2) Hemann et al.’s [87] implementation that relies on features from an experimental cKanren fork [5].

(3) A constraint microKanren implementation in a new constraint language.

For small logics like $\mathscr{A}$, the logic of “All” syllogisms, these models are similar. We see in Listing 4.17 that the main function implementing $\mathscr{A}$ proof search relies only on equality constraints. The program `A` expresses a 3-place relationship between a formula `φ`, an environment `Γ`, and a proof of `φ` from `Γ`.

We start to see differences when implementing larger logics like $\mathscr{R}^{*\dagger}$ [158], a relational syllogistic logic with recursively specified terms and full noun negation. Standard miniKanren forces us to coax the problem into the fixed, pre-existing CLP language. Listing 4.18 shows part of an $\mathscr{R}^{*\dagger}$ implementation in a modern Racket implementation of miniKanren that exemplifies this issue. We defer the remainder of the implementation to Appendix C. This implementation fakes custom constraints using the built-in miniKanren datatype constraints and negative numbers as type-tags. The implementation stipulates that `0` uniquely tags bottom, `-1` tags constants, and likewise `-2` for unary atoms, `-3` for binary atoms, and `-4` for variables. Listing 4.18 shows constraint-like relations to enforce the type tags of these faux constraints, and Listing 4.19 shows host-language data constructors. Here, we demonstrate the implementation of constants and unary atoms and literals; their binary analogues are similar. We made the design choice here in implementing the language to allow the same symbol for both unary and binary atoms. These design choices are captured only in the implementation of these faux-constraints and perhaps comments. As such, the choices are not especially well articulated, nor separated from the remainder of the logic program's implementation, nor open for automated checking. This has the additional drawback of generating the two instances of a literal rather than a single constraint. The programmer's intention was to combine them together as a constraint. Because these faux-constraints are just custom relations over the actual primitive constraints, the language implementation treats the clauses as two separate choices when searching. Further, information about the implementation of these ersatz constraints is tied in with the implementation of what was intended as the underlying logic program, so we have little hope of cleaning up the representation during the answer projection phase.

Contrast this approach to Hemann et al.'s [87] $\mathscr{R}^{*\dagger}$ implementation in an experimental cKanren dialect. This implementation, as exemplified by Listing 4.21, uses advanced, experimental cKanren features. It demonstrates a similar declarative style of implementing constraint systems. Some of the key features, including its constraint interaction definitions, are similar to those we can express in constraint miniKanren. Furthermore, it adds some special reification for these constraints.

```
(define-relation (constanto c)
  (fresh (sym)
    (symbolo sym)
    (== c `(-1 . ,sym))))

(define-relation (un-atomo a)
  (fresh (sym)
    (symbolo sym)
    (== a `(-2 . ,sym))))

(define-relation (un-literalo l)
  (conde
    ((un-atomo l))
    ((fresh (a)
       (un-atomo a)
       (== l `(not ,a))))))
```

LISTING 4.18. Faux constraint implementation with miniKanren constraints

```
(define (make-constant sym) `(-1 . ,sym))
(define (make-un-atom sym) `(-2 . ,sym))

(define/match (negate-un-literal n)
  [(`(not (-2 . ,(? symbol? x)))) `(-2 . ,x)]
  [(`(-2 . ,(? (symbol? x))) `(not (-2 . ,x)))])
```

LISTING 4.19. Translation function for faux miniKanren constraints of Listing 4.18

```
> (run* (q) (un-literalo q))
'(((-2 . _0) (sym _0)) ((not (-2 . _0)) (sym _0)))
```

LISTING 4.20. Execution and reification of faux-constraint literals

Listing 4.21 implements constraints for representing unary versions of both literals and atoms. Listing 4.22 shows these constraints act also as type tags for subsets of the ground terms.

```
(define-attribute unary-atomo
  #:satisfied-when symbol?
  #:incompatible-attributes (number bin-atomo))

(define-attribute un-literalo
  #:satisfied-when symbol-or-negated-symbol?
  #:incompatible-attributes (number bin-literalo bin-atomo))

(define-constraint-interaction
  [(un-literalo x) (unary-atomo x)] => [(unary-atomo x)])

(define/match (symbol-or-negated-symbol? s)
  [((? symbol?)) #t]
  [((cons 'not (? symbol?))) #t]
  [(x) #f])
```

LISTING 4.21. cKanren constraint definitions with violations, interactions, and satisfaction conditions

```
> (run* (q) (un-literalo q))
'((_0 : (un-literalo _0)))
```

LISTING 4.22. Execution and reification of cKanren-constraint literals

In this second implementation, we considered and dismissed the alternate design of Listing 4.19 because it was impractically slow for even medium-size relational proof search queries. None of these implementations promise top-end performance, but that was a consideration for this implementation.

Contrast these with a third, further approach in Listing 4.25. This gets some of the same benefits of the cKanren-based implementation. cKanren provides a much more powerful implementation. This is in part reflected in the size of the language implementation itself, as well as its trajectory. We have significantly less implementation overhead. We have also some different guarantees, tighter guarantees about the solver. In this constraint microKanren model, we add new infinite types primitives, and exclude certain classes of host language symbols altogether. We also add a second singleton set to prepare for a non-`not` constraint.

```
(define ((make-exotic-class l) c)
  (and (symbol? c)
       (let ([str (symbol->string c)])
         (and (memv (string-ref str 0) l)
              (string->number (substring str 1)) true))))

(define un-atom? (make-exotic-class (list #\p #\q)))
(define bin-atom? (make-exotic-class (list #\r)))

(define-attribute unary-atomo
  #:satisfied-when un-atom?
  #:incompatible-attributes (number bin-atomo))

(define-attribute bin-atomo
  #:satisfied-when un-atom?
  #:incompatible-attributes (number unary-atomo))
```

LISTING 4.23. Alternate construction of the faux miniKanren constraints from Listing 4.18

We define `not-un-lit?` structurally, as opposed to implementing another special class for non-unary literals. We must define it structurally, because in our system all terms with the same `pfs` need to be recognized by the same predicate. That means each instance of each constructor needs to be limited to within one predicate.

It takes somewhat more effort in constraint microKanren to implement the necessary constraint interactions from the base partition up. In this system the constraint implementer has to write a great deal of explicit constraint interaction predicates that the second system does not require. These kinds of interaction, however, as we mentioned in Section 3.4, can and should be automatically generated. Constraint microKanren explicitly employs a closed world assumption across a general term language; beyond the incidentally-disjoint predicates over basic Scheme terms, the other implementation does not have this property.

Since constraint microKanren that guarantees this independence of atomic negative constraints, we know that a constraint is solvable if the equalities are consistent, and in that every indexed n-tuple of atomic negative constraints is consistent.

```
(define ((make-ordinary-syms l) c)
  (and (symbol? c) (not (eqv? c 'not))
       (not (memv (string-ref (symbol->string c) 0) l))))

(define sconst? (make-exotic-class (string->list "abcd")))
(define svar? (make-exotic-class (list #\x #\y #\z)))
(define plain-sym? (make-ordinary-syms (string->list "abcdpqrxyz")))
(define is-not? ((curry eqv?) 'not))

(make-constraint-system
 #:var? number?
 #:posary-constructors ((cons . 2))
 #:infinite-types (pair? sconst? un-atom? bin-atom? svar? plain-sym?)
 #:finite-types (null? is-not?)
 …)
 …
 #:recursive-predicates
 ([not-un-lit? [((cons X _)) (not-not? X)]
               [((cons 'not X)) (not-un-atom? X)]
               [(X) (not-pair-or-un-atom? X)]])
```

Listing 4.24. A third construction of relational logic constraints

```
([for-all ([un-lito u]  [sconsto t])    #:fail-when ([equal? u t])]
 [for-all ([un-lito u]  [bin-atomo v])  #:fail-when ([equal? u v])]
 [for-all ([un-lito u]  [svaro w])      #:fail-when ([equal? u w])]
 [for-all ([un-lito u]  [plain-symo x]) #:fail-when ([equal? u x])]
 [for-all ([un-atomo u] [sconsto t])    #:fail-when ([equal? u t])]
 [for-all ([un-atomo u] [bin-atomo v])  #:fail-when ([equal? u v])]
 [for-all ([un-atomo u] [svaro w])      #:fail-when ([equal? u w])]
 [for-all ([un-atomo u] [plain-symo x]) #:fail-when ([equal? u x])]
 …)
```

Listing 4.25. A subset of the interactions required for the basic sets of constraints

A constraint microKanren constraint designer will start to feel the price of this guarantee as the need for finer and finer grained partitions over the terms to implement more constraints. This is the tension in our system between expressivity of the constraint language and the independence guarantee. Another limitation, and one that we had hoped for, is that we cannot express `negated-version-of` as a constraint, because these homogeneous atomic

constraints would not be independent of one another. We further compare and contrast our model of constraints to both Alvis et al.'s [6] approach and Alvis's later approach in the advanced experimental cKanren in Chapter 5.

In this chapter we have seen several extended examples of miniKanren programs made possible by, or greatly benefit from, the addition of constraints beyond standard equality and mechanisms for quickly adding and experimenting with them. Section 4.1 includes two Scheme-subset languages capable of expressing quines, one of which uses constraints and the other that does not. The example of Section 4.2 shows an interpreter for an imperative language useful for expressing program inversions and testing preconditions of annotated programs. In Section 4.3 we saw the implementation of relational type inferencers, and in Section 4.4 we included several related natural logics that provided reasoning both inside and outside the "Aristotelian border". These cases show how using a language with suitable constraints can clarify a programmer's intent. Being able to rapidly implement and test constraints leads the programmer to better model the problem domain, and all the attendant benefits of a higher-level logic language that fits the problem.

Chapter 5 Related Work

We express programmed constraints of LP languages via negated logical formulae understood in a closed world. Already, Chapter 1 introduces much of the general history and background of the field of CLP and Chapter 2 gives a theoretical background to the research we conducted within the CLP Scheme of Jaffar and Lassez. Here, we relate our work in the context of certain closely-related research efforts and within the arc of the more recent research in the field.

5.1. Functional Embeddings of Logic Programming

The functional and logic programming language communities have had a decades long and storied exchange. One byproduct of the cross-pollination of ideas across communities is a proliferation of embeddings of logic programming in functional host languages. These are too numerous to exhaustively list, but Komorowski's [124] QLOG is an early exemplary deep Lisp embedding. Implementers often position these as integrated, mixed-paradigm programming environments. Many subsequent systems' designers have similar purposes, and it is often only the intended usage pattern that distinguishes an embedded logic language from a mixed paradigm environment. Using a deep, interpreter-based embedding like Carlsson [31], Nilsson [165], and Wallace [221] side-steps some of the important issues we address with our work. Other shallow, compositional embeddings of logic programming come closer to the constraint-independent portion of the LP language embeddings we began describing in Section 3.5.

Robinson and Silbert's [181, 182] LogLisp is an early shallow Lisp-based embedding; they were initially motivated by the relative ease of extending an embedded language. LogLisp also offers complex search behavior beyond the standard depth-first search. Wand's [222]

alternative embedding is similar to LogLisp but more machine-oriented. This embedding implicitly takes syntactic equality as the only constraint, hidden within the Prolog-style syntax, and it is not immediately clear how this approach scales to a more general constraints framework.

We directly write our languages' relations in an if-and-only-if (IFF), bi-implicational form [48, p. 103] instead of interpreting implications through Clark's predicate completion. This is a small syntactic difference, but it does let us honestly give predicates a kind of closed-world meaning.[1] We do not introduce negation into our pure logic programs explicitly, and in the pure relational sub-language we do not permit general negation. Felleisen's [56] *Transliterating Prolog into Scheme* also permits pure definite logic programming with relations expressed in an IFF form. This system implements a stateful, strictly depth-first search for a single answer, and omits the `occurs?` check. We instead aim to implement pure relational programming through pure functional programming over finite structures.

Many existing pure functional LP embeddings reside in a lazy host language, for laziness precludes directly manipulating state. Hinze [94, 95] and Seres and Spivey [190, 202] both exhibit purely functional Haskell embeddings of LP features. The languages components we developed in Section 3.5 are in the main similar, though all three were developed independently. Hinze uses his embeddings to demonstrate the expressivity of monadic functional programming, and he captures Prolog's depth-first search behavior in his backtracking. Seres uses her embedding to express logic program transformations using host-language program equivalences, and she also explores and generalizes different search techniques.

[1] As Shepherdson [194] makes the case,

> Since one of the merits of logic programming is supposed to be making a rapprochement between the declarative and procedural interpretation of a program, in the interests of `Wysiwym`—What you say is what you mean—logic programming, I think that if you mean "iff" you should write "iff"; if you want to derive consequences of *comp(P)* you should write *comp(P)*, and if in order to carry out this derivation it is necessary to go via P then this should be done automatically.

Theirs rely on a lazy host, though; we distinguish our embedding by its search behavior in our specifically eager, functional host. We also importantly differentiate our work from most of the preceding by our distinct approach to negation. Byrd [24] directly precedes us in several respects. Friedman et al. [64] express their Scheme embedding functionally, but with a heavier reliance on macros that make their embedding less obviously compositional and intertwine the implementations of core functionality with the surface syntax. Further, they interleave beyond what is minimally necessary in general to maintain a complete search, which we require.

These earlier works rarely, if ever, explore constraints much beyond syntactic equality, and do not address generically embedding classes of them. However, Seres [190] and Byrd [24] probably come closest to the constraint-independent portion of our embedding, and we do continue their agendas in that we address several problems they highlighted.

Researchers have explored advanced search behavior both in logic programming and elsewhere. Clark et al. [36] describes some of the meta-control expressions for programs in a Prolog free of non-logical operators. Naish [161] and Vasak [214] survey the non-standard control and meta-control facilities of many Prolog implementations, including various kinds of intelligent backtracking and entire separate control meta-languages. These features require the programmer's separate, deliberate intervention beyond writing the declarative logic program. As we have mentioned, Spivey and Seres express breadth-first search in their embedding, via specially managed streams. Perhaps the closest approach to our particular model of nondeterminism is Kiselyov et al.'s [120] "Backtracking, interleaving, and terminating monad transformers: (functional pearl)". In a Haskell setting, they describe adding interleaved backtracking via monad transformers, and suggest fair search in logic programming as an application. They do not, however, go so far as to suggest taking the shape of the user's program and query as an heuristic for the minimal generally-necessary interleaving for implementing a complete search. We separated constraint solving and search to clarify the common portion of a parameterized CLP language; Schrijvers et al. [188] offer a different motivation. They implement different advanced search strategies via monad

transformers over basic search monads. It's not yet clear where miniKanren's interleaving depth-first search fits in their framework, or what benefits additional monad transformers over this search might bestow.

5.2. Functional Logic Programming

Like other functional embeddings of logic programming, our work superficially resembles "functional logic programming" (FLP) languages, as it does provide a kind of impoverished admixture of the logic and functional paradigms. However, our embeddings of logic languages inside functional programming meta-languages instead have more complicated semantics, or simply reduce to the semantics of the functional meta-language, and in any case express no underlying, united formalism. Developers came to FLP languages in part from their experiences with those earlier mixed paradigm amalgamated systems, so the ancestral resemblance is not surprising. Robinson [178], for instance, begins advancing LogLisp toward a more fully integrated functional-logic programming language. True functional logic languages should have a simpler unified semantics that encompass behaviors of both pure functional programming as well as logic programming. We draw much of this section from summaries of Aït-Kaci and Nasr [2], Bellia and Levi [14], and Hanus [79, 80].

Aït-Kaci and Nasr [2] describe a variety of ways to mix functional and logic programming. However, the term "functional-logic programming" now more commonly describes languages implemented via two general approaches. Functional-logic languages' designs proceed from either introducing logic variables to a reduction-based model of functional programming evaluation, or from letting programmers specify equational axioms that describe functions' behavior. In FLP these (conditional) equations act as (conditional) rewrite rules. When the atom to evaluate is not ground, evaluation would seem stuck. The two general approaches then are to either search for the right instantiation, or to delay that computation until the non-ground portions become sufficiently instantiated to proceed.

The former strategy, *narrowing*, uses some form of search to find an instantiation that solves these E-unification problems. Unification in even small equational theories is generally undecidable. In practice, then approaches restrict the allowed rewrite rules' forms to some set

expressive enough for programmers and still permits an efficient enough algorithm. An FLP language should evaluate fully ground atoms via a deterministic rewrite sequence like their functional language counterpart would. However, searching to sufficiently instantiate non-ground atoms can lead to a huge explosion in complexity. Much narrowing research involves designing restrictions that limit such explosion. *Basic narrowing*, like earlier restrictions to linear input resolution, restricts narrowing steps to positions inside one of the original program clauses or inside the query. *Selection based narrowing* corresponds to the selection rule of SLD. Certain other narrowing restrictions correspond to specifying a language's evaluation order. ALF [78] and BABEL [157], for instance, are narrowing-based languages.

With all forms of narrowing, however, the system must still sometimes guess the value with which to instantiate. The alternative is Aït-Kaci and Nasr's [2] *residuation* approach. This approach delays the evaluations of insufficiently-instantiated atoms. This avoids searching for values that other parts of the computation would eventually make manifest, like the way SLDNF manages negative literals. This strategy, unlike narrowing, is incomplete generally, and this can lead to floundering-like behavior. Residuation works for certain classes of programs though, and can be more efficient than narrowing. Languages like Escher [142], Le Fun [2], Life [1], and NUE-Prolog [160] rely on residuation or residuation-like behavior. For instance NUE-Prolog provides FLP by first transforming function definitions to predicates before executing in NU-Prolog, so it operationally behaves *as though* via residuation. Curry [81], meanwhile, uses both lazy narrowing as a general strategy as well as residuation to address concurrency. Finally, Braßel et al. [21] offer an intriguing approach for translating functional logic programs into monadic functional programs that preserve the flexibility to employ, e.g., different search strategies.

5.3. CLP and the CLP Scheme

We described early constraint logic programming and the development of the CLP Scheme in Section 1.2. The scheme was designed to help solve the problem of too many different one-off constraint systems, so myriad different collections of constraints and programming languages using those constraints fit within that framework [110]. Given this

abundance, we will focus on constraint systems that relate directly to our languages' Herbrand/symbolic constraints. Marriott and Stuckey [152] introduce constraint programming generally while also giving special consideration to syntactic equality and disequality constraints in logic programming.

Colmerauer [41] first introduces disequality constraints on infinite trees in Prolog II; these infinite data structures obviate the `occurs?` check. Barták [13] and Maher [147] define CLP(H) for the special case of equality constraints and negative atomic disequality constraints over finite trees. In several papers, Smith and Hickey [201] describe CLP($\mathscr{FT}$), a Prolog-like CLP language over finite trees but with universally quantified disequality constraints. Our term structures are similar, and their universal disequality constraints are more expressive than ours, as our disequality constraints are limited to the standard existentially quantified logic variables. They tailor their results toward applications in partial evaluation, and their universally quantified constraints (i.e. "U-constraints") [200] bear a strong relationship to Chan's [32] constructive negation.

We mentioned in Section 1.2.1 a different 1980s-era inspiration for CLP. This style was driven more by research in constraint satisfaction problems using constraint propagation to reduce the search space. The scheme's demand to express constraints via Horn formulae and to axiomatize the domain can make some domains difficult to express, and complicates using some OR based techniques like constraint propagation or letting constraints impact search behavior. Guo [77] and Höhfeld and Smolka [100] suggested related approaches that "turn CLP inside out", and instead treat CLP predicates as recursive definitions of sets of complex constraints defined over the primitive constraints of the domain. Then the full CLP program expresses the standard kind of constraint programming so well tuned for OR/AI techniques.

Alvis et al.'s [6] cKanren is an early constraint-based miniKanren that takes this different approach. Alvis et al. take finite domains as their prototypical example, and they use domain restriction and constraint propagation to solve constraints. Unlike languages generated

by our framework, their cKanren projects and minimizes answer constraints, and prettily format the results. Presently their solver uses a nondeterministic fix-point algorithm to solve constraint interactions.

Alvis's [5] subsequent iterations of cKanren utilized a more general, CHR-based technique written in a kind of event-driven programming style. Her more comprehensive system[2] aims to support not just constraints like our negative independent constraints, but also CLP(FD) constraints and beyond. She does not aim to characterize such a family of constraint languages by their theories, nor to find a class that admit our particular strategy for solving. Accommodating this more expressive constraint system forced her outside of our "sweet spot" of a simple solver, and into a more complicated constraint-interaction approach using an event-based programming model implementation and a fix point technique. This comes at the price of a significantly more complex implementation and the system takes a vastly larger code base to implement. Much of it makes heavy use of Racket-exclusive macro-level programming features. This is part of why we envisioned another approach to extending microKanren with constraints. These experimental branches are in an unstable state and incompatible with more recent releases of Racket. Alvis [5] indicates the project has stalled, and her development of it is indefinitely suspended. This is unfortunate; the cKanren examples in Hemann et al. [87] indicate how nice the complete system could be for an end user.

As the "mini-", "micro-" "c-" modifiers suggest, there is an earlier language "Kanren" [66]. Kanren, from the Japanese meaning "relation", is also a programming language based on relation composition and relation extension in the way many functional languages are based on the extension and composition of functions. miniKanren is named with respect to the earlier Kanren, but the languages have diverged significantly. Kanren has distinct

[2]cKanren does not so heavily enforce, and ultimately blurs the boundary between an explicitly fixed CLP language and a constraint-augmented programming language system. The project pivoted to Kraken, a prototype constraint-logic programming language implementation: `https://github.com/calvis/kraken`.

syntax, semantics, and design goals. miniKanren is "mini-" in the sense that as a language it makes more demands of the users and its implementations provides less automated support, and did not address constraints.

We usually see the constraint system's definition as picking out a member of a CLP language family and generating a *black-box solver*. Each of our languages give the CLP programmer some predefined set of constraints with which to program. The programmer uses these constraints to define a problem; the solver provides an answer without any programmer input as to how. In general, a sound black box solver gives truthful answers to binary questions about some logical relationship of constraints. Further, as discussed in Chapter 2, our solvers are always complete solvers, as opposed to the more common practice of incomplete solvers that may return "unknown" as an answer.

Members of the CLP community have constructed a number of frameworks and "shells" for building CLP languages over one or a variety of domains. In that sense such shells resemble also describe constraint-system parameterized families of languages. Lim and Stuckey [141] envision similar uses for their framework as we envision for constraint microKanren. Their aim is to allow the inclusion of any solver via access to an API. They give a CLP language implementation wrapped over this solver or these solvers, and their approach enables quicker development of constraints over various domains. We both separate control from the actual solving, and like our approach they require modifying the unifier. However, our approaches differ both in the manner we integrate constraint solving into a logic programming framework and in the styles of solvers we support. They support integrating existing solvers, while we support direct encodings of the theory. In terms of implementation, they provide an imperative solution based on WAM extensions, whereas we embed our framework in a general-purpose functional programming language. Further, the style of logic programming—the control mechanism, as well as trivialities like the surface LP syntax and term language—are instead the standard Prolog.

Constraint solvers are described as "black-box" in contrast with the glass-box style. In the latter style, CLP programmers specify or influence the control behavior of constraint solving. In some models, programmers can even define new kinds of constraints with their

programs. Frühwirth's [68] constraint-handling rules (CHR) and Kowalski et al.'s [129] *forward propagation rules* (FPR) are both glass box solving approaches, CHR being the more popular style. Frühwirth and Abdennadher [69] introduces many kinds of constraints with CHR and also uses them to implement arc consistency algorithms. Kowalski et al. defines FPRs via IFF definitions like we do our constraint predicate definitions, though their goals (comparable here to our constraints) are conjunctions of disjunctions. Like our systems' constraints, Frühwirth defines goals as conjunctions of atoms, though their CHRs are embedded in Prolog like languages and bound by their syntactic restrictions.

We can alternately view a program in a particular one of our CLP languages as something like a two-strata logic program, combining the program's actual predicates with the solver's logical specifications of predicates. Since we express our constraints' domains in part by giving a theory of that domain, similar theories are relevant to our work even if described for other reasons. A great deal of the related work focuses specifically on equations and disequations. Colmerauer [40] studies solving specifically equations and disequations on finite or infinite trees, inspired by his work on Prolog II. Maher [149] gives complete first-order axiomatizations for the theories of finite, infinite, and rational trees, of both finite and infinite languages. See Djelloul et al. [50] both for a more detailed introduction, and for continued related work combining constraints within those theories. For instance, Djelloul et al. add an operator (constraint) for labeling a given tree as finite. These are on the whole broader than our concern here, as we cannot express universally quantified constraints, nor do we entertain the algebra of infinite trees.

Solving combinations of equations and subterm relationships (like the positive version of our `absento` constraint) in tree algebras are less widely studied. Venkataraman [216] shows that the existential theory of free algebras over a first-order languages with equality, a subterm relation, and enough function symbols is decidable, but that the full first order theory is undecidable. Tulipani [208] revisits those results and also studies these questions in the algebras of infinite and rational trees. These are more general than the problems we intended to address; they study full positive and negative uses of the subterm relationship, while we restrict ourselves to negative uses.

We restrict ourselves to negative uses of subterm relationships to maintain the special form of the *independence of negated constraints* of Maher [147]. Lassez and McAloon [139] first characterize this independence property in the context of constraint logic programming. They point out this same property undergirds Colmerauer's earlier work, and exploit this property in implementing an algorithm to efficiently bring linear arithmetic constraints to a canonical form. Lassez and McAloon [138] later address the phenomenon in a more general setting. Several other kinds of constraint domains have this same independence of negative constraints property, including not only those described by Maher [149], but also feature trees with infinitely many sorts and features [110]. The standand Prolog-like trees are essentially *part* of these domains, and ours. This is uncommon in general—in many others domains the constraints and the objects of the constraint domain are separate and apart from the LP trees constructed *over* those elements.

Makowsky [150] describes important properties of the models of universal-existential Horn clause theories, pseudo-term structures, and their relationship to the consistency of the closed-world assumption. Pseudo-term structures are structures for which each element has existentially-closed primitive positive defining formula. For every element of a pseudo-term structure, there is some existential primitive positive formula that defines that element. Makowsky shows that a first-order theory admits initial models iff it is a partially functional $\forall\exists$-Horn theory. Theories admitting $\exists^+$-generic structures have the intersection property that their classes of models are closed under arbitrary limits. Volger [217] independently and contemporaneously gives results similar to Makowsky. In [150, §6.1], Makowsky re-proves Colmerauer's result as one instance of a more general result; the same proof works for any set of negated atomic- or negated $\exists^+$-formulae. Vel [215] gives a proof-theoretic characterization of a more general version of the independence property over formulae, for instance, of any sequence of quantifiers. These characterizations relate to the independence of an homogeneous set of negative atomic constraints; the theory of each such constraint is a universal existential horn theory and it therefore has a pseudo-term model. When combining these separate theories together, we consider a variant of the k-independence property from Cohen et al. [37].

5.4. Negation in Logic Programming

Because our LP languages' constraints are negations of LP predicates, our approach has much to do with general negation in LP. Given its widely-understood promise of programming purely in logic, programmers naturally expect LP languages to express general negation with its standard behavior. One of the longest-standing areas of LP language design is to provide reasonable semantics to negation, and exploring different definitions of both "negation" and "reasonable". Gabbay and Sergot [70, §1 Appendix] describe at least five different general flavors of negation that could interest a logic programmer, and whole volumes including Apt and Bol [9], Dix [49], Kunen [132], and Shepherdson [194, 195] just summarize and survey work addressing negation in logic programming. These approaches tend to be accounts of full, general predicate negation in logic programming; this seems more difficult than our limited use of negation to pre-defined constraints under Herbrand interpretations of some universal language. The computational complexity of full first-order theorem proving and the unfortunate language pragmatics of needing to explicitly define negatives discourage classical logic as a semantics for general logic programs. Various other approaches take as a program's semantics either a logical theory derived via transformation of the program, e.g. the program's completion, or instead via some canonical model, like the least Herbrand model, or specially restricted classes of models. Apt and Bol suggest Wallace [220] for an evaluation of the merits of both approaches. An early approach that is a common starting point for negation is Clark's [34] "negation as failure" (NF) rule. In fact, the NF view of negation also dates back to PLANNER [108]. This has a connection to pessimistic default reasoning and non-monotonic logic, treating the failure to prove as proof of the negation. This is often a reasonable shorthand, both because it can be efficiently executed, and simplifies the programmer's task. For instance, in programming a train schedule, assuming the negation unless the positive version succeeds obviates listing all the destinations and times for which a train does *not* depart.

The NF rule, as used in Clark[3] is compatible with both the completed database (CDB) and Reiter's [175] closed-world assumption (CWA). Both are studied in comparison to SLD with the NF rule. The CDB, via Clark's [34] predicate completion, is what you get from getting completed versions of all the predicates in the database (DB) of clauses. The CWA says that if a ground literal isn't implied by the database, then we assume it to be false. The CWA and CDB are in general different: one or the other could be inconsistent, and even when they're both consistent they might be incompatible (see Shepherdson [193]).

The use of negation as failure in traditional logic programming relates to our constraints. Our CLP languages *do* evaluate ground atomic negative constraints to failure when their ground positive counterparts succeed. We give our constraints names distinct from simply negative versions of their predicates. Such renaming to avoid explicit negation is also familiar to logic programmers, this approach dates back at least to 1979. We however also allow evaluation of non-ground terms in constraints. We take Kunen "universal language" approach, ensuring the language is "big enough" no matter the program.

Some approaches to negation also restrict the class of programs for which negation is allowed or meaningful, but not so far in the way we do so. Clark's SLDNF procedure expects a definite clause database, and instead will only allow negative literals in the queries. Extending the syntax to *program clauses* a la Lloyd that permit negation in normal logic programs adds additional complexity still, and we do not include any special mechanisms or fixed control to delay our constraints to only ground terms.

Our languages permit a limited form of normal logic programs; they permit negated literals in predicates' bodies only for pre-defined constraints. Furthermore, these constraints are permitted only in their negative form. Our approach works in part *because* we *stratify* the logic program into constraint and logic program portions, and because the limitations on constraints guarantees constraint checking terminates. Apt et al.'s [8] "stratified programs" are those programs where no relation "depends negatively" on itself. Our programs resemble 2-level stratified logic programs, with the CLP program in the top stratum,

[3]Basically, as in Prolog. Clark's query evaluation procedure (QEP) is SLD + NF, which Lloyd succinctly calls SLDNF.

and the constraints, treated logically, in the lower stratum. The *call-consistent* programs, those programs where no relation "depends negatively, oddly" on itself, encompass the stratified programs. It follows from a theory of Sato [187] that those programs will all have an Herbrand model. Sato describes the relationships between these and larger classes of programs. There, in the second level, we have that we get some NAF behavior. Shepherdson's [195] finite tree property ensures, for us, that the evaluation of any atomic negative constraints check will terminate with success or failure. Clark [34, Theorem 4] first showed that hierarchical databases imply the finite tree property.

Chapter 6 Summary and Future Work

In this final chapter, we summarize the argument of this thesis and the impact of the work it presents, describe some directions for future work, and conclude.

6.1. Summary

We have presented a framework for developing microKanren-like CLP languages as instances of the CLP Scheme. It supports customary miniKanren constraints as well as interesting and useful new ones. By decoupling the constraint management from the inference, control, and variable management, our work clarifies the semantics of microKanren. Our major contributions are:

Kernel Logic Language. We have exhibited the parameterized constraint microKanren language family and a framework for generating executable semantics for CLP in an eager, functional host language.

miniKanren Syntactic Extensions. We demonstrated a translation mechanism from pure miniKanren programs to microKanren programs via host-language macros.

Parameterized Constraint Systems. We situate microKanren constraints within the CLP Scheme and provide logical, algebraic, and operational semantics for constraint systems.

Interrelated Semantics. We exhibited these languages' relationship via the shared common portion of these embeddings' implementations that is an executable semantics.

Problem-solving Aids. Along the way we also provide and demonstrate a collection of novel constraints, and several novel uses of logic programming for relational interpreters.

Several minor results emerge from situating this work inside the CLP Scheme. These contributions include: cataloging, clarifying, correcting, and translating the terminological gap between miniKanreners and the larger logic programming community. We also help to characterize and clarify a sea of existing implementations.

We envision our framework as a lightweight tool for rapidly prototyping constraint sets. Our results are useful for CLP language implementers wanting to test-drive their model constraint systems. Language designers can now explore and test constraint definitions and interactions without building or modifying a complicated, dedicated solver tailored to some other application. Academics, professionals, and hobbyists can now roll their own CLP systems. We also imagine our system as an educational artifact to provide functional programmers a minimal executable instance for constructing CLP Scheme-constraint systems. As Seres [190] says in her 2001 dissertation:

> There are several promising research avenues based on this implementation: our favourite ones are an algebraic specification of a constraint language, and subsequent applications [sic] the program transformation from functional programming. Both the implementation and the examples may help functional programmers realise how close this constraint-based style is to functional programming, and might lead to a further cross-fertilisation of the methodologies for these declarative styles.

We further suggest that allowing the logic programmer to implement their constraint-logic programming language in logic programming is a novel end in itself.

Enumerating a minimum host language feature set will help improve the microKanren embeddings in the various host languages. This makes room for more shallow embeddings and implementations in other styles. We can improve the efficiency of non-Scheme implementations, like the JavaScript implementation. Rather than needing to first re-implement the miniKanren term language structure, the JavaScript Kanren implementer can now meaningfully use a direct embedding into a native JavaScript term language, and change to a different concrete implementation of unification over this new term language. Removing the

overhead of the binary tree term language should improve performance. More generally, this change allows the constraint designer to reason about constraints' definitions and constraint interactions across such changes to the term language.

We did not intend to generate efficient, state-of-the-art CLP languages. Even without performance numbers or a full benchmark test suite, we know our generated language implementations do not compete with state of the art systems. Instead of efficiency, our aim is a simple, general framework for implementing constraints in microKanren, and we hope to have followed Robinson's [180] dictum:

> I think that it's important to go for elegance and beauty in these mathematical engineering quests. You can't really go far wrong if it's beautiful.

6.2. Future Work

In this section we briefly outline some natural next steps in a research agenda starting from this dissertation. This future work includes suggestions for improving existing systems, making programming in logic languages more declarative, and simplifying the construction of relational constraint logic programs.

6.2.1. Constraint System Performance. We have deliberately rejected certain common but implementation-complicating features. Although we have preferred simplicity over optimized performance, we hope in future research to investigate some of the following low-hanging ideas for improvements.

With the term "redundancy" we collect here two related areas of future work. Various notions of redundancy appear in the context of logic programming, and problem solving more broadly, but we call something redundant when "it can be removed without affecting the system of concern" [115]. We have seen in this dissertation that the constraint solving algorithm can render a constraint redundant.

Our generated constraint solvers are not at all adapted for incrementally solving constraints. In the execution of a program our implementations will repeatedly solve from scratch (portions of) the same constraint problem, instead of solving incrementally and

memoizing the intermediate solutions. We also face redundancy in the constraint store, to the point of even adding wholly duplicative constraints. We know that duplicating a constraint in the store by adding another copy is redundant. Since a problem's complexity is as much a function of that problem's degree of redundancy as it is a function of the size of the problem instance, redundancy can causes inefficiency, confusion, and heartache. In extreme cases redundancy can make otherwise tractable problems effectively impossible. Addressing this general issue seems likely to have significant benefits.

Beyond these suggestions for improving the constraint system and those of Section 3.4, we hope to implement various other simple optimizations including early projection [61], or to optionally call out to an appropriate dedicated constraint solver. Future, subsequent researchers or users of this system wanting our style of control might integrate existing solvers, rather than using our approach to solving, to create an Echidna-style shell [141] for constraint microKanren languages. One can envision a user building solver-aided languages in Racket with Constraint microKanren, which offers promising suggestions for future work [207].

Michael Ballantyne has shown how to implement the standard miniKanren constraints using attributed variables [104]. This would be useful to incorporate into our parameterized constraint systems. We hope to develop many of these optimizations as sequences of correctness-preserving transformations from our kind of straightforward embedding implementations, and we also hope to evaluate their performance impact.

6.2.2. Presenting Answer Constraints. Since our system's solvers only ensure consistency and do not simplify the constraint set during the solving process, the next step forward is to build a parameterized constraint simplifiers framework like we have for solvers. This is what the designers of miniKanren implementations typically call the "reification"[1] of answer constraints. In implementing this we hope to support answer constraint simplification, answer projection, entailment, determinacy detection, and prettily printing answers.

[1]This usage of meaning of "reification" is independent and distinct from Friedman and Wand's [67] usage in the literature on reflection.

We may find that the specification of constraint simplification and printing mechanisms are even bigger than those of the constraint systems and that they may generate more code. However, it may be that by limiting our systems to these negative, atomic-independent symbolic constraints, we can generically fashion this simplifier and answer projector for our kinds of constraints where it would be more difficult in general. We expect this to even more closely resemble a CHR-style approach.

6.2.3. Deep Embeddings and LP Hosts. Moving forward we want to explore deep embeddings, and many attendant research questions follow from that idea. Rosenblatt et al. [183] have begun work in this direction for guided synthesis problems. We have already constructed two deep embeddings (i.e. interpreters) that implement search using respectively a stream-based and a success and failure continuation-based model of interleaved backtracking. These models extend the behavior of Hughes's [103] backtracking monads, which are designed for use in a lazy language. These implementations mediate our host language's strictness with explicitly marked delay positions. These demarcations also inform the search's interleaving behavior. Our two different implementations of extending the standard backtracking model by delays and interleaving suggests future work comparing these two models and proving their equivalence a la Hughes. Specifically we expect to extend the work of Danvy et al. [46] and Wand and Vaillancourt [223] to relate our models of interleaved backtracking. In part this would be reminiscent of Chung-chieh Shan's prior unpublished work connecting an early Kanren with continuation-based backtracking and an implementation with list-based backtracking monad. Hinze [93] demonstrates that our stream model of nondeterminism is asymptotically worse than a context-passing implementation so this may lead in certain cases to improved performance. We might hope to prove the equivalence of the two implementations by deriving the same abstract machine via correctness-preserving transformations.

We might like also to formally connect the deep and shallow approaches to implementing our embeddings. This might follow Gibbons and Wu [73] in connecting deep and shallow embeddings, and it may be we find it easier to connect the embeddings with continuation-based backtracking than the embeddings with stream-based backtracking.

Moving to a deep embedding carries a host of attendant benefits and opens new questions to approach. That basic work we described enables asking important questions and permits foundational research on the search and its completeness. These two models fully characterize our microKanren's search, and the context-passing implementations give a better foundation to precisely explain how microKanren's search works and what it does. It provides another context in which to formally characterize the search's fairness and to prove that property holds.

This will also let us, in future work, compare microKanren's interleaving to the search of older, more traditional, miniKanren implementations and with other existing work beyond LP. These other approaches to search include other complete variants of DFS and Seres's [190] technique for implementing breadth-first search. We should be able to connect our microKanren-style search to Schrijvers et al.'s [188] monadic search transformers technique and describe our approach as a search transformer. This surely also provides a better setting to continue exploring, with the ultimate aim of back-porting a fair queue-based conjunction to the shallow embedding without resorting to employing full breadth-first search.

We hope to explore how programmers can already achieve similar results to the examples we demonstrate in a variety of existing LP languages in their already existing favorite logic language. By building a deep embedding hosted in Prolog, we can introduce microKanren's search strategy, and perhaps its constraint set, to Prolog as a Prolog meta-interpreter. This work could also bridge some of the connections between miniKanren and traditional LP community.

Further, the Prolog meta-interpreter for miniKanren could lead to some performance improvements. There is a wealth of existing logic programming research focused on improving performance of existing implementations and existing feature sets. This opens the door to using many well-known Prolog techniques, and this thesis anticipates theoretically-driven

approaches to improving performance. There is room to apply a whole host of the tools of the formal study of programming languages to this particular instance. For example, we could then apply partial evaluation [92, 145] to hopefully achieve more efficient compilation, and even derive specialized logic engines, a la Biernacki and Danvy [17]. Optimizing this implementation could improve our ability to compile constraint microKanren. So our research also addresses performance optimization issues important for compiler writers implementing fast CLP systems. In addition to these specific next steps, our approach also reinvigorates some older, time-worn research questions. A whole host of optimizations and performance considerations come when working with constraints over pure relational programs. Our work provides a reason to and a context in which to reconsider from first principles some early decisions of many avenues of logic language design.

6.3. Conclusion

This dissertation presents the microKanren approach to adding constraints beyond equality. In doing so, we hope to have improved understanding and eased the development of constraints in miniKanren. In addition to the aforementioned results and avenues for future research we've introduced, we hope for one further outcome. Quoth Robinson [179], more deeply integrating functional and logic programming seems to address an unfortunate and longstanding issue:

> It has been a source of weakness in declarative programming that there have been two major paradigms needlessly pitted against each other, competing in the same marketplace of ideas. The challenge is to end the segregation and merge the two. There is in any case, at bottom, only one idea.

We hope that this work does some small part to help bridge this divide.

Appendix A microKanren Implementations

We layer over either of these implementations with a suite of macros, and export only the relevant ones. This appropriately hides these underlying primitives' implementations.

A.1. microKanren Implementations with Equality Constraints

The implementation of core microKanren as a Racket embedding.

```
(define ((succeed) S/c) (list S/c))
(define ((fail) S/c) '())

(define (((make-constraint-goal-constructor invalid? key) . ts) S/c)
  (let ([S (hash-update (car S/c) key ((curry cons) ts))])
    (if (invalid? S) '() (list `(,S . ,(cdr S/c))))))

(define ((call/fresh f) S/c)
  (let ((c (cdr S/c)))
    ((f (var c)) `(,(car S/c) . ,(+ c 1)))))

(define ((disj g1 g2) S/c) ($append (g1 S/c) (g2 S/c)))
(define ((conj g1 g2) S/c) ($append-map g2 (g1 S/c)))

(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) S/c) (delay/name (g S/c))))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))

(define ($append-map g $)
  (cond
    ((null? $) '())
    ((promise? $) (delay/name ($append-map g (force $))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))
```

```
(define (call/initial-state n g)
  (take n (pull (g `(,S0 . 0)))))

(define (pull $) (if (promise? $) (pull (force $)) $))

(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
            (take (and n (- n 1)) (pull (cdr $)))))))

(define ((ifte g0 g1 g2) S/c)
  (let loop (($ (g0 S/c)))
    (cond
      ((null? $) (g2 S/c))
      ((promise? $) (delay/name (loop (force $))))
      (else ($append-map $ g1)))))

(define ((once g) S/c)
  (let loop (($ (g S/c)))
    (cond
      ((null? $) '())
      ((promise? $) (delay/name (loop (force $))))
      (else (list (car $))))))
```

A.2. Constraint microKanren Framework Implementation

This implementation requires `srfi/31` and Racket's `generic-bind`, **`contract`**, and `syntax/parse/define` libraries, as well as Racket's `generic-bind`, `racket/match`, `syntax/stx` `racket/syntax`, and `syntax/parse/define` libraries as well as `srfi/1` and `srfi/31` for syntax.

```
(define-syntax-rule (make-subst var? (con d ...) ...)
  (rec (sub x v t)
    (match t
      [(? var?) (if (equal? x t) v t)]
      [(con d ...) (con (sub x v d) ...)]
      ...
      [else t])))
```

```
(define-syntax-rule (make-occurs? var? (con d ...) ...)
  (rec (o? x v)
    (match v
      [(? var?) (equal? x v)]
      [(con d ...) (or (o? x d) ...)]
      ...
      [else false])))

(define-syntax-rule (make-ext-s var? diag ...)
  (let ([occurs? (make-occurs? var? diag ...)]
        [subst (make-subst var? diag ...)])
    (λ (x t s)
      (cond
        [(occurs? x t) false]
        [else
         (cons `(,x . ,t)
           (~for/list ([($: a d) s])
              (cons a (subst x t d))))]))))

(define-syntax-rule (make-subst-all var? (con d ...) ...)
  (rec (w* t s)
    (match t
      [(? var?)
       (cond
         [(assoc t s) => cdr]
         [else t])]
      [(con d ...) (con (w* d s) ...)]
      ...
      [else t])))

(define-syntax-rule (make-unify var? subst-all (c p1 p2) ...)
  (let ([ext-s (make-ext-s var? (c . p1) ...)])
    (rec (unify u v s)
      (let ([u (subst-all u s)] [v (subst-all v s)])
        (match* (u v)
          [(u v) #:when (equal? u v) s]
          [((? var?) v) (ext-s u v s)]
          [(u (? var?)) (ext-s v u s)]
          [((c . p1) (c . p2))
           (for/fold ([s s])
                     ([t1 (list . p1)]
                      [t2 (list . p2)])
             #:break (not s)
             (unify t1 t2 s))]
          ...
          [(_ _) false])))))
```

```
(define-syntax-rule (make-fail-check subst-all ([(b x ...) ...]
                                                [(p? fa  ...) ...]))
  (λ (s)
    (~for*/or ([($list x ...) b] ...)
       (and (p? (subst-all fa s) ...) ...))))

(define-syntax-rule (make-normlzr subst-all unify
                      ([(b x ...) ...] [vs cs] [(p? fa ...) ...]))
  (λ (s)
    (~for*/fold ([s s])
                ([($list x ...) b] ...)
                #:break (not s)
                (if (and (p? (subst-all fa s) ...) ...)
                    (for/fold ([s s])
                              ([t1 (list . vs)]
                               [t2 (list . cs)])
                      (unify t1 t2 s))
                    s))))

(define-syntax-rule
  (make-solver subst-all unify (cid ...) (rr ...) (fr ...))
  (λ (S)
    (let ([cid (hash-ref S 'cid)] ...)
      (cond
        [((compose (make-normlzr subst-all unify rr) ...) '())
         => (or/c (make-fail-check subst-all fr) ...)]
        [else #t]))))

(define (((make-constraint-goal-constructor invalid? key) . ts) S/c)
  (let ([S (hash-update (car S/c) key ((curry cons) ts))])
    (if (invalid? S) '() (list `(,S . ,(cdr S/c))))))
```

```
(begin-for-syntax
  (define-syntax-class fail-rule
    #:attributes (norm)
    (pattern ((~literal for-all)
              [(cid:id x:id  ...+) ...+]
              (~datum #:fail-when) [gpapp ...+])
             #:with norm #'([(cid x  ...) ...] [gpapp ...])))
  (define-syntax-class rewrite-rule
    #:attributes (norm)
    (pattern ((~literal for-all)
              [(cid:id x:id  ...+) ...+]
              #:rewrite
              [(v (~datum =>) c) ...+]
              #:when
              [gpapp ...+])
             #:with norm #'([(cid x  ...) ...]
                            [(v ...) (c ...)]
                            [gpapp ...]))))

(define-for-syntax (make-pattern ns)
  (build-list (syntax->datum ns) generate-temporary))
```

```
(define-syntax-parser make-constraint-system
  [(_ #:var? var?
      #:posary-constructors ((c:id . n:nat) ...)
      #:infinite-types (ip:id ...)
      #:finite-types (fp:id ...+)
      #:== ==
      #:=/= =/=
      #:primitive-predicates ((ppn:id ((~datum one-of) fp/ip ...+)) ...)
      #:term-structural-functions ((sfn:id sfcls ...+) ...)
      #:recursive-predicates ((rpn:id [(t ...) body] ...+) ...)
      #:constraints (((rcn:id x ...) nrp) ...)
      #:rewrite-rules (rr:rewrite-rule ...)
      #:failure-rules (fr:fail-rule ...)
      #:sugar-constraints (((sugn:id suga:id ...) b) ...))
   (with-syntax
     ([S0 (syntax-local-introduce #'S0)]
      [(p1 ...) (stx-map make-pattern #'(n ...))]
      [(p2 ...) (stx-map make-pattern #'(n ...))])
     #'(begin
         (define invalid?
           (let ([ppn (or/c fp/ip ...)]
                 ...)
             (letrec ([sfn (match-lambda**
                            [((? var? X)) X]
                            sfcls ...)]
                      ...)
               (letrec ([rpn (λ args
                               (or (match args
                                     [(list t ...) body]
                                     [else false])
                                   ...))]
                        ...)
                 (let* ([subst-all (make-subst-all var? (c . p1) ...)]
                        [unify
                         (make-unify var? subst-all (c p1 p2) ...)])
                   (make-solver subst-all unify (== =/= rcn ...)
                    [rr.norm ... ([(== t1 t2)] [(t1) (t2)] [])]
                    [([(=/= a d)] [(equal? a d)])
                     ([(rcn x ...)] [nrp]) ... fr.norm ...])))))))
         (define S0
           (make-immutable-hasheqv '((=/=) (==) (rcn) ...)))
         (define == (make-constraint-goal-constructor invalid? '==))
         (define =/= (make-constraint-goal-constructor invalid? '=/=))
         (define rcn (make-constraint-goal-constructor invalid? 'rcn))
         ...
         (define (sugn suga ...) b) ...))])
```

Appendix B miniKanren Implementation

Our revised miniKanren implementation based on the microKanren from Appendix A.

```
(define-syntax disj+
  (syntax-rules ()
    ((_ g) g)
    ((_ g0 g ...) (disj g0 (disj+ g ...)))))

(define-syntax conj+
  (syntax-rules ()
    ((_ g) g)
    ((_ g0 g ...) (conj g0 (conj+ g ...)))))

(define-syntax-rule (conde (g0 g ...) (g0* g* ...) ...)
  (disj+ (conj+ g0 g ...) (conj+ g0* g* ...) ...))

(define-syntax fresh
  (syntax-rules ()
    ((_ () g0 g ...) (conj+ g0 g ...))
    ((_ (x0 x ...) g0 g ...)
     (call/fresh (λ (x0) (fresh (x ...) g0 g ...))))))

(define-syntax-rule (run n (q) g0 g ...)
  (call/initial-state n (fresh (q) g0 g ...)))

(define-syntax ifte*
  (syntax-rules ()
    ((_ g) g)
    ((_ (g0 g1) (g0* g1*) ... g)
     (ifte g0 g1 (ifte* (g0* g1*) ... g)))))

(define-syntax-rule (conda (g0 g1 g ...) ... (gn0 gn ...))
  (ifte* (g0 (conj+ g1 g ...)) ... (conj+ gn0 gn ...)))

(define-syntax-rule (condu (g0 g1 g ...) ... (gn0 gn ...))
  (conda ((once g0) g ...) ... ((once gn0) gn ...)))
```

Appendix C CLP Examples

C.1. Equality constraint Relational Interpreter

```
;; Terms
(define-relation (expr? o)
  (conde
    ((fresh (n)
       (== o `(x . ,n))
       (nat? n)))
    ((== o 'quote))
    ((fresh (n t)
       (== o `(λ (x . ,n) ,t))
       (nat? n)
       (expr? t)))
    ((fresh (t₁ t₂)
       (== o `(,t₁ ,t₂))
       (expr? t₁)
       (expr? t₂)))
    ((fresh (t₁ t₂)
       (== o `(list₂ ,t₁ ,t₂))
       (expr? t₁)
       (expr? t₂)))))
```

Listing C.1. Relational Interpreter using only equality constraints

```
(define-relation (nat? o)
  (conde
    ((== o '()))
    ((fresh (n)
       (== o `(s . ,n))
       (nat? n)))))
```

Listing C.2. Help relation matching unary naturals

```
;; Environment
(define-relation (env? o)
  (conde
    ((== o '()))
    ((fresh (n v e)
       (== o `((,n . ,v) . ,e))
       (nat? n)
       (val? v)
       (env? e)))))
```

Listing C.3. Help relation matching well-formed environments

C.2. Quines, Twines

```
> (run 3 (q) (eval q q))
((((λ (_0) (list _0 (list 'quote _0)))
   '(λ (_0) (list _0 (list 'quote _0))))
  (=/= ((_0 closure)) ((_0 list)) ((_0 quote)))
  (sym _0))
 (((λ (_0) (list ((λ (_1) _0) '_2) (list 'quote _0)))
   '(λ (_0) (list ((λ (_1) _0) '_2) (list 'quote _0))))
  (=/= ((_0 _1)) ((_0 closure)) ((_0 list)) ((_0 quote))
       ((_0 λ)) ((_1 closure)))
  (sym _0 _1)
  (absento (closure _2)))
 (((λ (_0) (list _0 (list ((λ (_1) 'quote) '_2) _0)))
   '(λ (_0) (list _0 (list ((λ (_1) 'quote) '_2) _0))))
  (=/= ((_0 closure)) ((_0 list)) ((_0 quote)) ((_0 λ))
       ((_1 closure)) ((_1 quote)))
  (sym _0 _1)
  (absento (closure _2))))
> (run 1 (p) (fresh (q) (=/= p q) (eval p q) (eval q p)))
(('((λ (_0) (list 'quote (list _0 (list 'quote _0))))
    '(λ (_0) (list 'quote (list _0 (list 'quote _0)))))
 (=/= ((_0 closure)) ((_0 list)) ((_0 quote)))
 (sym _0))
```

Listing C.4. Quine and Twine Query Examples

C.3. Program Cycles

Here we include sample results of searches for program cycles.

```
> (run 1 (p q r)
    (eval `(,p ,q) r) (eval `(,q ,r) p) (eval `(,r ,p) q))
((quote quote quote))
> (run 1 (p q r) (=/= p q) (=/= q r) (=/= r p)
    (eval `(,p ,q) r) (eval `(,q ,r) p) (eval `(,r ,p) q))
((((λ (_0)
     '(λ (_1)
        (list
         'λ
         '(_2)
         (list
          'quote
          (list 'λ '(_0) (list 'quote (_1 '_3)))))))
   (λ (_2)
     '(λ (_0)
        '(λ (_1)
           (list
            'λ
            '(_2)
            (list
             'quote
             (list 'λ '(_0) (list 'quote (_1 '_3))))))))
   (λ (_1)
     (list
      'λ
      '(_2)
      (list 'quote (list 'λ '(_0) (list 'quote (_1 '_3)))))))
  (=/= ((_0 closure)) ((_0 quote)) ((_1 closure)) ((_1 list))
       ((_1 quote)) ((_2 closure)) ((_2 quote)))
  (sym _0 _1 _2)
  (absento (closure _3))))
```

Listing C.5. Query for program cycles

C.4. Mirrored-language Interpreter

This section contains the interpreter for the mirrored language, and `tsil-reporpo`, a help relation needed to describe reversed proper lists. While the syntax of the language is mirrored, the internal representations of closures and environments are not.

```
(define-relation (fo-lavo pxe vars env lav)
  (conde
    [(fresh (v)
       (== `(,v etouq) pxe)
       (absento 'etouq vars)
       (absento 'closure v)
       (== v lav))]
    [(tsil-reporpo pxe vars env lav)
     (absento 'closure pxe)]
    [(symbolo pxe) (lookupo pxe vars env lav)]
    [(fresh (rotar dnar x ydob vars^ env^ a)
       (== `(,dnar ,rotar) pxe)
       (fo-lavo rotar vars env `(closure ,x ,ydob ,vars^ ,env^))
       (fo-lavo dnar vars env a)
       (fo-lavo ydob `(,x . ,vars^) `(,a . ,env^) lav))]
    [(fresh (x ydob)
       (== `(,ydob (,x) adbmal) pxe)
       (== `(closure ,x ,ydob ,vars ,env) lav)
       (symbolo x)
       (absento 'adbmal env))]))

(define-relation (tsil-reporpo pxe vars env lav)
  (conde
    [(== `(tsil) pxe)
     (== `() lav)]
    [(fresh (a d t-a t-d)
       (== `(,a . ,d) pxe)
       (== `(,t-a . ,t-d) lav)
       (fo-lavo a vars env t-a)
       (tsil-reporpo d vars env t-d))]))
```

LISTING C.6. The `fo-lavo` evaluation relation with a split environment

C.5. Relational miniProlog Interpreter

This appendix contains a relational implementation of a miniPascal interpreter a la Sestoft's [192] *The Structure of a Self-applicable Partial Evaluator*. It differs from the functional implementation of this same miniPascal interpreter in that, upon update, we

`cdr` to the variable in question and then rebuild the front of the environment. This implementation maintains the ordering of variables in the environment. We could also have split the environment to begin with and mandated that globals begin bound. This may aid program generation when running with all fresh variables.

```
(define-relation (initialize-local-envo vars out)
  (conde ;; vars
    [(== vars `()) (== out `())]
    [(fresh (a d)
       (== `(,a . ,d) vars)
       (fresh (d^)
         (== `((,a  _) . ,d^) out)
         (initialize-local-envo d d^)))]))

(define-relation (initialize-global-envo vars vals out)
 (conde ;; vars
   [(== vars `()) (== vals `()) (== out `())]
   [(fresh (a d v vs)
      (== vars `(,a . ,d))
      (== vals `(,v . ,vs))
      (fresh (res)
        (== out `((,a ,v) . ,res))
        (initialize-global-envo d vs res)))]))

(define-relation (appendo l s out)
  (conde
    [(== '() l) (== s out)]
    [(fresh (a d res)
       (== `(,a . ,d) l)
       (== `(,a . ,res) out)
       (appendo d s res))]))

(define-relation (run-programo V1* V2* B value* out)
  (fresh (genv lenv)
    (initialize-global-envo V1* value* genv)
    (initialize-local-envo V2* lenv)
    (fresh (env)
      (appendo lenv genv env)
      (evalBlocko B env out))))
```

Listing C.7. Relations for environments and initial program execution

```
(define-relation (evalBlocko B env out)
  (conde ;; B
    [(== B `()) (== env out)]
    [(fresh (h t)
       (== B `(,h . ,t))
       (evalCommandso h t env out))]))

(define-relation (evalCommandso C B env out)
  (conde ;; B
    [(== B `()) (evalCommando C env out)]
    [(fresh (h t)
       (== B `(,h . ,t))
       (fresh (env^)
         (evalCommando C env env^)
         (evalCommandso h t env^ out)))]))

(define-relation (reverseo-env^ acc env^ out)
  (conde
    ((== '() acc) (== out env^))
    ((fresh (a d)
       (== `(,a . ,d) acc)
       (fresh (env^^)
         (== env^^ (cons a env^))
         (reverseo-env^ d env^^ out))))))

(define-relation (update-env V pr env acc out)
  (fresh (a d)
    (== `(,a . ,d) env)
    (fresh (aa da)
      (== `(,aa ,da) a)
      (conde
        ((== aa V)
         (fresh (env^)
           (== env^ `(,pr . ,d))
           (reverseo-env^ acc env^ out)))
        ((=/= aa V)
         (fresh (acc^)
           (== `(,a . ,acc) acc^)
           (update-env V pr d acc^ out)))))))
```

Listing C.8. Relations to evaluate blocks and modify environments

```
(define-relation (evalCommando C env out)
  (conde ;; C
    ;; [(== C `(print-env)) (prt) (== out env)]
    [(fresh (V E)
       (== C `(:= ,V ,E))
       (fresh (val pr)
         (== `(,V ,val) pr)
         (evalExpressiono E env val)
         (update-env V pr env '() out)))]
    [(fresh (E B1 B2)
       (== C `(if ,E ,B1 ,B2))
       (fresh (val)
         (evalExpressiono E env val)
         (fresh (b-exp)
           (isTrueo val b-exp)
           (conde
             ((== b-exp '(true)) (evalBlocko B1 env out))
             ((=/= b-exp '(true)) (evalBlocko B2 env out))))))]
    [(fresh (E B)
       (== C `(while ,E ,B))
       (fresh (val b-exp)
         (evalExpressiono E env val)
         (isTrueo val b-exp)
         (conde
           ((== b-exp '(true))
            (fresh (env^)
              (evalBlocko B env env^)
              (evalCommando C env^ out)))
           ((=/= b-exp '(true)) (== out env))))))]))

(define-relation (lookup-envo E env out)
  (fresh (a env^)
    (== env `(,a . ,env^))
    (fresh (x v)
      (== a `(,x ,v))
      (conde
        ((== x E) (== a out))
        ((=/= x E) (lookup-envo E env^ out))))))
```

Listing C.9. Relations for evaluating commands and environment lookup

```
(define-relation (evalExpressiono E env out)
  (conde ;; E
    [(symbolo E)
     (fresh (a ad)
       (== ad out)
       (lookup-envo E env `(,a ,ad)))]
    [(fresh (Value)
       (== `(quote ,Value) E) (== Value out))]
    [(fresh (E^)
       (== `(car ,E^) E)
       (fresh (a d)
         (== a out)
         (evalExpressiono E^ env `(,a . ,d))))]
    [(fresh (E^)
       (== E `(cdr ,E^))
       (fresh (a d)
         (== d out)
         (evalExpressiono E^ env `(,a . ,d))))]
    [(fresh (E1 E2)
       (== E `(cons ,E1 ,E2))
       (fresh (val1 val2)
         (== out `(,val1 . ,val2))
         (evalExpressiono E1 env val1)
         (evalExpressiono E2 env val2)))]
    [(fresh (E^)
       (== E `(atom ,E^))
       (fresh (val)
         (eval-atomo val out)
         (evalExpressiono E^ env val)))]
    [(fresh (E1 E2)
       (== E `(equal ,E1 ,E2))
       (fresh (val1 val2)
         (eval-equalo val1 val2 out)
         (evalExpressiono E1 env val1)
         (evalExpressiono E2 env val2)))]))
```

Listing C.10. Relation for evaluating an MP expression

```
(define-relation (eval-atomo v out)
  (conde ;; v
    [(fresh (a d)
       (== v `(,a . ,d))
       (== out '()))]
    [(not-pairo v) (== out '(true))]))

(define-relation (eval-equalo v1 v2 out)
  (conde
    ((== v1 v2) (== out '(true)))
    ((=/= v1 v2) (== out '()))))

(define-relation (isTrueo value out)
  (conde
    ((fresh (a d)
       (== `(,a . ,d) value)
       (== out '(true))))
    ((not-pairo value) (== out '()))))
```

Listing C.11. Small MP help relations

C.6. Traverse Graph

These examples demonstrate a variety of graph walks. In the current implementation, the cycle check happens in the program before we check for the presence of the nodes in the graph. This means that when miniKanren runs examples in the inverted modality, and asked for cycles, it doesn't respect the definition of the graph. This could be improved in later versions. Our interpreter does not have a return statement, and so we modified the program from the original implementation. Through a series of flags to modify control, we regained more or less the original behavior, adding a return statement to the interpreter would allow a more clear implementation of the algorithm in miniPascal. It may be that doing so would *disconnect the wires* a la Byrd & Amin.

```
(define traverse-graph
  '((:= flag '(true))
    (:= out '())
    (:= rest '())
    (while flag
      ((if t ;; t is not a leaf
           ((:= rest (cons (cdr t) rest)) ;; center, right
            (:= t (car t))) ;; left
           ((:= out (cons t out))
            (if rest
                ((if (car rest)
                     ((:= out (cons (car (car rest)) out)) ;; center
                      (:= t (cdr (car rest))) ;; right
                      (:= rest (cdr rest)))
                     ((:= flag '())
                      (:= out (cons 'Error out)))))
                ((:= flag '()))))))))
```

Listing C.12. The MP language `traverse-graph` program

C.7. Relational Type-checking and Inference

This section contains the implementation of a relational type inferencer for a small language with polymorphic **`let`**.

```
(define-relation (⊢ Γ e τ)
  (conde
    [(stringo e) (== 'String τ)]
    [(conde
       [(== e '#t)]
       [(== e '#f)])
     (== τ 'Bool)]
    [(fresh (x b)
       (== `(λ (,x) ,b) e)
       (symbolo x)
       (fresh (τₓ τb)
         (== `(,τₓ → ,τb) τ)
         (not-in-envo 'λ Γ)
         (⊢ `((,x (mono ,τₓ)) . ,Γ) b τb)))]
      [(fresh (v e' body)
         (== `(let ([,v ,e']) ,body) e)
         (symbolo v)
         (not-in-envo 'let Γ)
         (⊢ `((,v (poly ,e' ,Γ)) . ,Γ) body τ))]
      [(symbolo e)
       (fresh (τ')
         (lookupo Γ e τ')
         (conde
           [(== `(mono ,τ) τ')]
           [(fresh (e' Γ')
              (== `(poly ,e' ,Γ') τ')
              (⊢ Γ' e' τ))]))]
    [(fresh (t c a)
       (== `(if ,t ,c ,a) e)
       (⊢ Γ t 'Bool)
       (⊢ Γ c τ)
       (⊢ Γ a τ))]
      [(fresh (rator rand)
       (== `(,rator ,rand) e)
       (fresh (τₓ)
         (⊢ Γ rator `(,τₓ → ,τ))
         (⊢ Γ rand τₓ)))]
      [(fresh (f func x)
         (== `(fix (λ (,f) ,func)) e)
         (not-in-envo 'fix Γ)
         (⊢ `((,f (mono ,τ)) . ,Γ) func τ))]))
```

Listing C.13. The relational type inferencer with polymorphic **let**

```
(define-relation (not-in-envo x env)
  (conde
    [(== '() env)]
    [(fresh (y _ rest)
       (== `((,y ,_) . ,rest) env)
       (=/= y x)
       (not-in-envo x rest))]))

(define-relation (lookupo Γ y τ)
  (fresh (x τ′ Γ′)
    (== `((,x ,τ′) . ,Γ′) Γ)
    (conde
      [(== x y) (== τ′ τ)]
      [(=/= x y) (lookupo Γ′ y τ)])))
```

LISTING C.14. An environment restricting relation and relational type environment lookup

C.8. Natural Logic $\mathscr{R}^{*\dagger}$

An implementation of the $\mathscr{R}^{*\dagger}$ logic without custom constraints that fakes domain constraints by using negative number tags and the standard miniKanren symbol constraints.

```
(define (make-un-atom sym) `(-2 . ,sym))
(define (make-bin-atom sym) `(-3 . ,sym))

(define-relation (unary-atomo a)
  (fresh (sym)
    (symbol sym)
    (== `(-2 ,sym) a)))

(define-relation (bin-atomo a)
  (fresh (sym)
    (symbol sym)
    (== a `(-3 . ,sym))))

(define (negate-un-literal n)
  (match n
    (`(not (-2 . ,x)) #:when (symbol? x) `(-2 . ,x))
    (`(-2 . ,x) #:when (symbol? x) `(not (-2 . ,x)))))

(define (negate-bin-literal n)
  (match n
    (`(not (-3 . ,x)) #:when (symbol? x) `(-3 . ,x))
    (`(-3 . ,x) #:when (symbol? x) `(not (-3 . ,x)))))

(define-relation (negate-un-literalo l o)
  (conde
    ((unary-atomo l)
     (== o `(not ,l)))
    ((== l `(not ,o))
     (unary-atomo o))))

(define-relation (negate-bin-literalo l o)
  (conde
    ((bin-atomo l)
     (== `(not ,l) o))
    ((bin-atomo o)
     (== l `(not ,o)))))
```

Listing C.15. A set of constructor functions and basic relations for implementing the $\mathscr{R}^{*\dagger}$

```
(define-relation (un-literalo l)
  (conde
    ((unary-atomo l))
    ((fresh (a)
       (== l `(not ,a))
       (unary-atomo a)))))

(define-relation (bin-literalo l)
  (conde
    ((bin-atomo l))
    ((fresh (a)
       (bin-atomo a)
       (== l `(not ,a))))))

(define-relation (set-termo s)
  (conde
    ((un-literalo s))
    ((fresh (p r)
       (conde
         ((== s `(∃ ,p ,r))
          (un-literalo p)
          (bin-literalo r))
         ((== s `(∀ ,p ,r))
          (un-literalo p)
          (bin-literalo r)))))))

(define-relation (sentenceo s)
  (conde
    ((fresh (p c)
       (== `(∃ ,p ,c) s)
       (un-literalo p)
       (set-termo c)))
    ((fresh (p c)
       (== `(∀ ,p ,c) s)
       (un-literalo p)
       (set-termo c)))))

(define-relation (negate-quant q q^)
  (conde
    ((== q '∀) (== q^ '∃))
    ((== q '∃) (== q^ '∀))))
```

LISTING C.16. Relations for constructing higher-level components of the $\mathscr{R}^{*\dagger}$ implementation

```
(define-relation (negateo s o)
  (fresh (qf1 p c)
    (== `(,qf1 ,p ,c) s)
    (conde
      ((un-literalo c)
       (fresh (qf1^ c^)
         (== `(,qf1^ ,p ,c^) c)
         (negate-quant qf1 qf1^)
         (negate-un-literalo c c^)))
      ((fresh (qf2 q r)
         (== `(,qf2 ,q ,r) c)
         (fresh (qf1^ qf2^ r^)
           (== `(,qf1^ ,p (,qf2^ ,q ,r^)) o)
           (negate-quant qf1 qf1^)
           (negate-quant qf2 qf2^)
           (negate-bin-literalo r r^)))))))

(define-relation (membero x ls)
  (fresh (a d)
    (== `(,a . ,d) ls)
    (conde
      [(== a x)]
      [(=/= a x) (membero x d)])))
```

LISTING C.17. The relations for membership and general negation of $\mathscr{R}^{*\dagger}$ sentences

```
(define-relation (R G phi proof)
  (conde
    ((membero phi G)
     (== `(Gamma : ,G => ,phi) proof))
    ((fresh (p c) ;; D1
       (== `(∃ ,p ,c) phi)
       (unary-atomo p)
       (set-termo c)
       (fresh (q r1 r2)
         (== `((,r1 ,r2) D1=> ,phi) proof)
         (unary-atomo q)
         (R G `(∃ ,p ,q) r1)
         (R G `(∀ ,q ,c) r2))))
    ((fresh (p c) ;; B
       (== `(∀ ,p ,c) phi)
       (unary-atomo p)
       (set-termo c)
       (fresh (q r1 r2)
         (== `((,r1 ,r2) B=> ,phi) proof)
         (unary-atomo q)
         (R G `(∀ ,p ,q) r1)
         (R G `(∃ ,p ,c) r2))))
    ((fresh (p c) ;; D2
       (== `(∃ ,p ,c) phi)
       (unary-atomo p)
       (set-termo c)
       (fresh (q r1 r2)
         (== `((,r1 ,r2) D2=> ,phi) proof)
         (unary-atomo q)
         (R G `(∀ ,q ,p) r1)
         (R G `(∃ ,q ,c) r2))))
    ((fresh (p) ;; T
       (== `(∀ ,p ,p) phi)
       (== phi proof)
       (un-literalo p)))
    ((fresh (p) ;; I
       (== `(∃ ,p ,p) phi)
       (un-literalo p)
       (fresh (c r)
         (set-termo c)
         (== `((,r) I=> ,phi) proof)
         (R G `(∃ ,p ,c) r))))
    …))
```

Listing C.18. Part one of the implementation of proof search for the $\mathscr{R}^{*\dagger}$ logic

```
(define-relation (R G phi proof)
  (conde
    …
    ((fresh (p nq) ;; D3
       (== `(∃ ,p ,nq) phi)
       (unary-atomo p)
       (fresh (q c nc r1 r2)
         (== `((,r1 ,r2) D3=> ,phi) proof)
         (negateo c nc)
         (un-literalo q) ;; these two lines could be better specialized.
         (negate-un-literalo q nq)
         (R G `(∀ ,q ,nc) r1)
         (R G `(∃ ,p ,c) r2))))
    ((fresh (p c) ;; A
       (== `(∀ ,p ,c) phi)
       (un-literalo p)
       (set-termo c)
       (fresh (np r)
         (negate-un-literalo p np)
         (== `((,r) A=> ,phi) proof)
         (R G `(∀ ,p ,np) r))))
    ((fresh (p) ;; II
       (== `(∃ ,p ,p) phi)
       (unary-atomo p)
       (fresh (q r t)
         (unary-atomo q)
         (bin-literalo t)
         (== `((,r) II=> ,phi) proof)
         (R G `(∃ ,q (∃ ,p ,t)) r))))
    ((fresh (p q t) ;; AA
       (== `(∀ ,p (∃ ,q ,t)) phi)
       (unary-atomo p)
       (unary-atomo q)
       (bin-literalo t)
       (fresh (q^ r1 r2)
         (== `((,r1 ,r2) AA=> ,phi) proof)
         (unary-atomo q^)
         (R G `(∀ ,p (∀ ,q^ ,t)) r1)
         (R G `(∃ ,q ,q^) r2))))
    …))
```

Listing C.19. Part two of the implementation of proof search for the $\mathscr{R}^{*\dagger}$ logic

```
(define-relation (R G phi proof)
  (conde
    …
    ((fresh (p q t) ;; EE
       (== `(∃ ,p (∃ ,q ,t)) phi)
       (unary-atomo p)
       (unary-atomo q)
       (bin-literalo t)
       (fresh (q^ r1 r2)
         (== `((,r1 ,r2) EE=> ,phi) proof)
         (unary-atomo q^)
         (R G `(∃ ,p (∃ ,q^ ,t)) r1)
         (R G `(∀ ,q^ ,q) r2))))
    ((fresh (p q t) ;; AE
       (== `(∀ ,p (∃ ,q ,t)) phi)
       (unary-atomo p)
       (unary-atomo q)
       (bin-literalo t)
       (fresh (q^ r1 r2)
         (== `((,r1 ,r2) AE=> ,phi) proof)
         (unary-atomo q^)
         (R G `(∀ ,p (∃ ,q^ ,t)) r1)
         (R G `(∀ ,q^ ,q) r2))))
    ((sentenceo phi) ;; RAA
     (fresh (p np nphi r)
       (negate-un-literalo p np)
       (negateo phi nphi)
       (== `((,r) RAA=> ,phi) proof)
       (R `(,nphi . ,G) `(∃ ,p ,np) r)))))
```

Listing C.20. Part three of the implementation of proof search for the $\mathscr{R}^{*\dagger}$ logic

Bibliography

[1] Hassan Aït-Kaci. "An Overview of Life". In: *Next Generation Information System Technology*. Ed. by Joachim W. Schmidt and Anatoly A. Stogny. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 42–58.

[2] Hassan Aït-Kaci and Roger Nasr. "Integrating Logic and Functional Programming". In: *Lisp and Symbolic Computation* 2.1 (02/1989), pp. 51–89.

[3] Luc Albert, Rafael Casas, and François Fages. "Average-Case Analysis of Unification Algorithms". In: *Theoretical Computer Science* 113.1 (1993), pp. 3–34. URL: https://doi.org/10.1016/0304-3975(93)90208-B.

[4] Luc Albert, Rafael Casas, François Fages, A. Torrecillas, and Paul Zimmermann. "Average Case Analysis of Unification Algorithms". In: *STACS 91, 8th Annual Symposium on Theoretical Aspects of Computer Science, Hamburg, Germany, February 14-16, 1991, Proceedings*. Ed. by Christian Choffrut and Matthias Jantzen. Vol. 480. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1991, pp. 196–213. URL: https://doi.org/10.1007/BFb0020799.

[5] Claire E. Alvis. *Later cKanren Implementations*. Private communication. 2019.

[6] Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. "cKanren: miniKanren with Constraints". In: *Scheme Workshop '11* (2011).

[7] Nada Amin. Constraint-free Relational Quine Generator. URL: https://github.com/namin/logically/blob/2693692029b9271c30247f5843f0dfa38555dc88/src/logically/exp/lf1/quine.clj.

[8] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. "Towards a Theory of Declarative Knowledge". In: *Foundations of Deductive Databases and Logic Programming*. Ed. by Jack Minker. Morgan Kaufmann, 1988, pp. 89–148. URL: https://doi.org/10.1016/b978-0-934613-40-8.50006-3.

[9] Krzysztof R. Apt and Roland N. Bol. "Logic programming and negation: A survey". In: *The Journal of Logic Programming* 19-20 (05/1994), pp. 9–71. URL: https://doi.org/10.1016/0743-1066(94)90024-8.

[10] Krzysztof R. Apt and Maarten H Van Emden. "Contributions to the Theory of Logic Programming". In: *Journal of the ACM* 29.3 (07/1982), pp. 841–862. URL: https://doi.org/10.1145/322326.322339.

[11] Franz Baader and Wayne Snyder. "Unification Theory". In: *Handbook of Automated Reasoning*. Ed. by John Alan Robinson and Andrei Voronkov. Vol. 1. Amsterdam New York Cambridge, Mass: Elsevier MIT Press, 2001, pp. 445–532. URL: https://doi.org/10.1016/b978-044450813-3/50010-2.

[12] Isaac Balbin and Koenraad Lecot, eds. *Logic programming : A Classified Bibliography*. Fitzroy, Victoria, Australia: Wildgrass Books, 1985. URL: https://doi.org/10.1007/978-94-009-5044-3.

[13] Roman Barták. *Constructive Negation in CLP(H)*. Tech. rep. 98/6. Prague: Department of Theoretical Computer Science, Charles University, 07/1998.

[14] Marco Bellia and Giorgio Levi. "The Relation between Logic and Functional Languages: A Survey". In: *The Journal of Logic Programming* 3.3 (1986), pp. 217–236. URL: https://doi.org/10.1016/0743-1066(86)90014-2.

[15] Johann van Benthem. "A Brief History of Natural Logic". In: *Logic, Navya-Nyāya, & Applications: Homage to Bimal Krishna Matilal*. Ed. by Mihir K. Chakraborty, Benedikt Löwe, Madhabendra Nath Mitra, and Sundar Sarukkai. Studies in logic 15. London: College Publications, 2008, pp. 21–42.

[16] Jon Bentley. "Programming Pearls: Little Languages". In: *Communications of the ACM* 29.8 (08/1986), pp. 711–721. URL: http://doi.acm.org/10.1145/6424.315691.

[17] Dariusz Biernacki and Olivier Danvy. "From Interpreter to Logic Engine by Defunctionalization". In: *Logic Based Program Synthesis and Transformation: 13th International Symposium, LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*. Ed. by Maurice Bruynooghe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 143–159. URL: http://dx.doi.org/10.1007/978-3-540-25938-1_13.

[18] Michel Billaud. "Prolog Control Structures: a Formalization and its Applications". In: *Programming of Future Generation Computers: Proceedings of the First Franco-Japanese Symposium on Programming of Future Generation Computers, Tokyo, Japan, 6-8 October 1986*. Ed. by Kazuhiro Fuchi and Maurice Nivat. Elsevier Science Publishers BV. North Holland, 1988, pp. 57–73.

[19] Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. "Experience with Embedding Hardware Description Languages in HOL". In: *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings*. Ed. by Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute. Vol. A-10. IFIP Transactions. Amsterdam, The Netherlands: North-Holland, 1992, pp. 129–156. URL: http://dl.acm.org/citation.cfm?id=645902.672777.

[20] Daniel W. Brady, Jason Hemann, and Daniel P. Friedman. "Little Languages for Relational Programming". In: *2014 Scheme And Functional Programming Workshop*. Ed. by Jason Hemann and John Clements. Washington, D.C., USA: Computer Science Department, Indiana University, 2015. URL: http://cs.indiana.edu/pub/techreports/TR718.pdf.

[21] Bernd Braßel, Sebastian Fischer, Michael Hanus, and Fabian Reck. "Transforming functional logic programs into monadic functional programs". In: *International Workshop on Functional and Constraint Logic Programming*. Springer. 2010, pp. 30–47.

[22] David C Brock, ed. *Understanding Moore's Law: Four Decades of Innovation*. Philadelphia, Pa: Chemical Heritage Foundation, 2006.

[23] Craig Brozefsky. "Core.logic and SQL Killed my ORM". In: *Clojure/West*. San Jose, California, 08/2013. URL: infoq.com/presentations/Core-logic-SQL-ORM.

[24] William E. Byrd. "Relational programming in miniKanren: Techniques, applications, and implementations". PhD thesis. Indiana University, 2009.

[25] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. "A Unified Approach to Solving Seven Programming Problems (Functional Pearl)". In: *Proc. ACM Program. Lang.* 1.ICFP (08/2017), 8:1–8:26. URL: http://doi.acm.org/10.1145/3110252.

[26] William E. Byrd and Daniel P. Friedman. "αKanren: A Fresh Name in Nominal Logic Programming". In: *Proceedings of Scheme Workshop '07, Université Laval Technical Report DIUL-RT-0701*. (*see* webyrd.net/alphamk/alphamk.pdf *for improvements*). 2007, pp. 79–90.

[27] William E. Byrd, Eric Holk, and Daniel P. Friedman. "miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl)". In: *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*. Scheme '12. Copenhagen, Denmark: ACM, 2012, pp. 8–29. URL: http://doi.acm.org/10.1145/2661103.2661105.

[28] William E Byrd and Nada Amin. *Proceedings of the 2019 miniKanren and Relational Programming Workshop*. Tech. rep. TR-02-19. Cambridge, Massachusetts: Computer Science Group, Harvard University, 2019. URL: dash.harvard.edu/bitstream/handle/1/41307116/tr-02-19.pdf.

[29] Venanzio Capretta. "General recursion via coinductive types". In: *Logical Methods in Computer Science* 1.2 (07/2005). Ed. by Henk Barendregt, pp. 1–28. URL: https://doi.org/10.2168/lmcs-1(2:A1)2005.

[30] Mats Carlsson. "On Implementing Prolog in Functional Programming". In: *Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, New Jersey, USA, February 6-9, 1984*. IEEE-CS, 1984, pp. 154–159.

[31] Mats Carlsson. “On Implementing Prolog in Functional Programming”. In: *New Generation Computing* 2.4 (1984). adapted from [30], pp. 347–359. URL: https://doi.org/10.1007/BF03037326.

[32] David Chan. “Constructive Negation Based on the Completed Database”. In: *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*. Ed. by Robert A. Kowalski and Kenneth A. Bowen. MIT Press, 1988, pp. 111–125.

[33] Keith L. Clark. *Logic Programming Schemes and their Implementations*. Tech. rep. UPMAIL Technical Report No. 59. Uppsala Programming Methodology and Artificial Intelligence Laboratory, Computing Science Department, Uppsala University, 03/1990.

[34] Keith L. Clark. “Negation as Failure”. In: *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d’études et de recherches de Toulouse, France, 1977*. Ed. by Hervé Gallaire and Jack Minker. Advances in Data Base Theory. New York: Plemum Press, 1977, pp. 293–322. URL: https://doi.org/10.1007/978-1-4684-3384-5_11.

[35] Keith L. Clark and Sten-Åke Tärnlund, eds. *Logic programming*. Automatic Programming Information Centre (Brighton). Studies in data processing no. 16. Academic Press, 1982.

[36] Keith L Clark, Frank G McCabe, and Steve Gregory. “Co-routining in IC-Prolog”. In: *Logic Programming*. Ed. by Keith L. Clark and Sten-Åke Tärnlund. Automatic Programming Information Centre (Brighton). Studies in data processing no. 16. Academic Press, 1982, pp. 253–266.

[37] David Cohen, Peter Jeavons, Peter Jonsson, and Manolis Koubarakis. “Building tractable disjunctive constraints”. In: *Journal of the ACM* 47.5 (09/2000), pp. 826–853. URL: https://doi.org/10.1145/355483.355485.

[38] Jacques Cohen. “A view of the origins and development of Prolog”. In: *Communications of the ACM* 31.1 (01/1988), pp. 26–36. URL: https://doi.org/10.1145/35043.35045.

[39] Jacques Cohen. “Constraint Logic Programming Languages”. In: *Communications of the ACM* 33.7 (07/1990), pp. 52–68. URL: http://doi.acm.org/10.1145/79204.79209.

[40] Alain Colmerauer. “Equations and Inequations on Finite and Infinite Trees”. In: *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1984, Tokyo, Japan, November 6-9, 1984.* Ed. by Institute for New Generation Computer Technology. OHMSHA Ltd. Tokyo and North-Holland, 1984, pp. 85–99.

[41] Alain Colmerauer. *PROLOG II: Manuel de Référence et Modèle théorique*. Tech. rep. Groupe D’intelligence Artificielle, Université Aix-Marseille II, 1982.

[42] Alain Colmerauer and Philippe Roussel. “The Birth of Prolog”. In: *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*. Ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM, 1993, pp. 37–52. URL: https://doi.org/10.1145/154766.155362.

[43] Hubert Comon and Jean-Luc Rémy. *How to characterize the language of ground normal forms*. Tech. rep. Rapports de Recherche 676. INRIA, 1987.

[44] Sylvain Conchon and Jean-Christophe Filliâtre. “A Persistent Union-Find Data Structure”. In: *Proceedings of the ACM SIGPLAN Workshop on ML*. ACM. Freiburg, Germany, 10/2007, pp. 37–46.

[45] Ryan Culpepper. “Fortifying macros”. In: *Journal of Functional Programming* 22.4-5 (08/2012), pp. 439–476. URL: http://dx.doi.org/10.1017/s0956796812000275.

[46] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. “A unifying approach to goal-directed evaluation”. In: *New Generation Computing* 20.1 (2002), p. 53. URL: http://dx.doi.org/10.1007/BF03037259.

[47] Martin Davis. “The prehistory and early history of automated deduction. Classical Papers on Computational Logic 1957-1966”. In: *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*. Ed. by Jorg Siekmann and Graham Wrightson. Vol. 1. New York: Springer-Verlag, 1983.

[48] Pierre Deransart and Jan Małuszyński. *A Grammatical View of Logic Programming*. Cambridge, MA, USA: MIT Press, 1993.

[49] Jürgen Dix. "Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview". In: *Logic, Action, and Information - Essays on Logic in Philosophy and Artificial Intelligence. Walter de Gruyter, Berlin, New York, 1996 (Based on a meeting held in autumn 1992 in Konstanz, Germany)*. Ed. by André Fuhrmann and Hans Rott. 1996, pp. 241–327.

[50] Khalil Djelloul, Thi-Bich-Hanh Dao, and Thom W. Frühwirth. "Theory of finite or infinite trees revisited". In: *Theory and Practice of Logic Programming* 8.4 (2008), pp. 431–489. URL: https://doi.org/10.1017/S1471068407003171.

[51] Kees Doets. *From Logic to Logic Programming*. Cambridge, Mass: MIT Press, 1994.

[52] Michael Downward. *Logic and Declarative Language*. Routledge, 03/2004. URL: https://doi.org/10.4324/9780203211991.

[53] E. W. Elcock. "Absys: the first logic programming language —A retrospective and a commentary". In: *The Journal of Logic Programming* 9.1 (06/1990), pp. 1–17. URL: https://doi.org/10.1016/0743-1066(0)90030-9.

[54] E. W. Elcock. "Absys: The Historical Inevitability of Logic Programming". In: *Logic Programming, Proceedings of the North American Conference 1989, Cleveland, Ohio, USA, October 16-20, 1989. 2 Volumes*. Ed. by Ewing L. Lusk and Ross A. Overbeek. MIT Press, 1989, pp. 1201–1214.

[55] Herbert B Enderton. *A Mathematical Introduction to Logic*. Elsevier, 2001. URL: https://doi.org/10.1016/c2009-0-22107-6.

[56] Matthias Felleisen. *Transliterating Prolog into Scheme*. Tech. rep. TR182. Computer Science Department, Indiana University, Bloomington, 10/1985. URL: https://help.sice.indiana.edu/techreports/TRNNN.cgi?trnum=TR182.

[57] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. "A Programmable Programming Language". In: *Communications of the ACM* 61.3 (02/2018), pp. 62–71. URL: http://doi.acm.org/10.1145/3127323.

[58] Matthias Felleisen, Michael Hanus, and Simon Thompson. *Proceedings of the Workshop on Functional and Declarative Programming in Education*. Tech. rep. Technical Report 99-346. Computer Science Department, Rice University, 08/1999. URL: http://www.ccs.neu.edu/home/matthias/FDPE99/.

[59] Melvin Fitting. "Bilattices and the Semantics of Logic Programming". In: *J. Log. Program.* 11.1&2 (1991), pp. 91–116. URL: https://doi.org/10.1016/0743-1066(91)90014-G.

[60] Matthew Flatt and PLT. *Reference: Racket*. Tech. rep. PLT-TR-2010-1. http://racket-lang.org/tr1/. PLT Design Inc., 2010.

[61] Andreas Fordan. *Projection in Constraint Logic Programming*. Ios Press, 1999.

[62] Martin Fowler. *Domain-specific languages*. Upper Saddle River, NJ: Addison-Wesley, 2011.

[63] Martin Fowler. *Language workbenches: The killer-app for domain specific languages*. 2005. URL: https://martinfowler.com/articles/languageWorkbench.html.

[64] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, 07/2005, p. 176.

[65] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. *The Reasoned Schemer, Second Edition*. The MIT Press, 03/2018. URL: mitpress.mit.edu/books/reasoned-schemer-0.

[66] Daniel P. Friedman and Oleg Kiselyov. *A declarative applicative logic programming system*. 2005. URL: http://kanren.sourceforge.net/.

[67] Daniel P. Friedman and Mitchell Wand. "Reification: Reflection without Metaphysics". In: *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, August 5-8, 1984, Austin, Texas, USA*. ACM, 1984, pp. 348–355. URL: https://dl.acm.org/citation.cfm?id=800055.

[68] Thom Frühwirth. *Constraint simplification rules*. Tech. rep. 18. European Computer-Industry Research Centre, 1992.

[69] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Berlin Heidelberg New York: Springer Science & Business Media, 2003. URL: https://doi.org/10.1007/978-3-662-05138-2.

[70] Dov M. Gabbay and Marek J. Sergot. "Negation as Inconsistency I". In: *Journal of Logic Programming* 3.1 (1986), pp. 1–35. URL: https://doi.org/10.1016/0743-1066(86)90002-6.

[71] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. "Trampolined Style". In: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.* Ed. by Didier Rémy and Peter Lee. ACM, 1999, pp. 18–27. URL: https://doi.org/10.1145/317636.317779.

[72] Henry Meloni Gérard Battani. *Interpreteur du langage de programmation Prolog*. Tech. rep. Rapport de D.E.A. U.E.R da Luminy, Université d'Aix-Marseille, 1973.

[73] Jeremy Gibbons and Nicolas Wu. "Folding domain-specific languages: deep and shallow embeddings (functional Pearl)". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 339–347. URL: https://doi.org/10.1145/2628136.2628138.

[74] Robert Glück. "A Self-applicable Online Partial Evaluator for Recursive Flowchart Languages". In: *Software - Practice and Experience* 42.6 (2012), pp. 649–673. URL: https://doi.org/10.1002/spe.1086.

[75] Daniel Gregoire. *Web Testing with Logic Programming*. 2013. URL: http://www.youtube.com/watch?v=09zlcS49zL0.

[76] Sumit Gulwani. "Automating String Processing in Spreadsheets Using Input-output Examples". In: *ACM Sigplan Notices*. Vol. 46. 1. ACM. 2011, pp. 317–330.

[77] Yike Guo. "Definitional constraint programming". PhD thesis. Department of Computing, Imperial College, 1994.

[78] Michael Hanus. "Compiling logic programs with equality". In: *Programming Language Implementation and Logic Programming*. Ed. by Pierre Deransart and Jan Maluszyński. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 387–401.

[79] Michael Hanus. "Functional Logic Programming: From Theory to Curry". In: *Programming Logics - Essays in Memory of Harald Ganzinger*. Ed. by Andrei Voronkov and Christoph Weidenbach. Vol. 7797. Lecture Notes in Computer Science. Springer, 2013, pp. 123–168. URL: https://doi.org/10.1007/978-3-642-37651-1_6.

[80] Michael Hanus. "The integration of functions into logic programming: From theory to practice". In: *The Journal of Logic Programming* 19-20 (1994). Special Issue: Ten Years of Logic Programming, pp. 583–628. URL: http://www.sciencedirect.com/science/article/pii/0743106694900345.

[81] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. "Curry: A truly functional Logic Language". In: *Proceedings ILP Workshop on Visions for the Future of Logic Programming*. Ed. by Leon Sterling. MIT Press, 1995, pp. 95–107.

[82] Robert Harper. *What, If Anything, Is A Declarative Language?* Blog. 2013. URL: http://existentialtype.wordpress.com/2013/07/18/what-if-anything-is-a-declarative-language/.

[83] Jason Hemann and Daniel P. Friedman. "μKanren: A Minimal Functional Core for Relational Programming". In: *Scheme 13*. 2013. URL: http://schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf.

[84] Jason Hemann and Daniel P. Friedman. "A Framework for Extending microKanren with Constraints". In: *Proceedings of Scheme Workshop '15, Northeastern University Technical Report NU-CCIS-2016-001*. 2015. URL: http://hdl.handle.net/2047/D20213213.

[85] Jason Hemann and Daniel P. Friedman. "A Framework for Extending microKanren with Constraints". In: Proceedings 29th and 30th Workshops on *(Constraint) Logic Programming* and 24th International Workshop on *Functional and (Constraint)*

Logic Programming, Dresden and Leipzig, Germany, 22nd September 2015 and 12-14th September 2016. Ed. by Sibylle Schwarz and Janis Voigtländer. Vol. 234. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2017, pp. 135–149.

[86] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. "A Small Embedding of Logic Programming with a Simple Complete Search". In: *Proceedings of DLS '16*. ACM, 2016. URL: http://dx.doi.org/10.1145/2989225.2989230.

[87] Jason Hemann, Cameron Swords, and Lawrence S Moss. "Two Advances in the Implementations of Extended Syllogistic Logics". In: *Joint Proceedings of the 2nd Workshop on Natural Language Processing and Automated Reasoning, and the 2nd International Workshop on Learning*. 2015, p. 1.

[88] Pascal Hentenryck. *Constraint satisfaction in logic programming*. Cambridge, Mass: MIT Press, 1989.

[89] Jacques Herbrand. "Recherches sur la théorie de la démonstration". PhD thesis. Université de Paris, 1930.

[90] Felienne Hermans. *Spreadsheets for Developers*. St. Louis, Missouri, USA. URL: youtube.com/watch?v=0CKru5d4GPk.

[91] Carl Hewitt. "Middle History of Logic Programming Resolution, Planner, Prolog, and the Japanese Fifth Generation Project". In: (2009).

[92] Timothy J. Hickey and Donald A. Smith. "Toward the Partial Evaluation of CLP Languages". In: *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '91. New Haven, Connecticut, USA: ACM, 1991, pp. 43–51. URL: http://doi.acm.org/10.1145/115865.115871.

[93] Ralf Hinze. "Deriving backtracking monad transformers". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.* Ed. by Martin Odersky and Philip Wadler. ACM, 2000, pp. 186–197. URL: https://doi.org/10.1145/351240.351258.

[94] Ralf Hinze. "Prolog's control constructs in a functional setting: Axioms and implementation". In: *International Journal of Foundations of Computer Science* 12.02 (2001), pp. 125–170.

[95] Ralf Hinze. "Prological Features in a Functional Setting: Axioms and Implementation." In: *Fuji International Symposium on Functional and Logic Programming.* Ed. by Masahiko Sato and Yoshihito Toyama. World Scientific, 1998, pp. 98–122. URL: https://doi.org/10.1142/3709.

[96] Wilfrid Hodges. "Logical Features of Horn Clauses". In: *Handbook of Logic in Artificial Intelligence and Logic Programming (Vol. 1).* Ed. by Dov M Gabbay, Christopher John Hogger, and John Alan Robinson. New York, NY, USA: Oxford University Press, Inc., 1993, pp. 449–503. URL: http://dl.acm.org/citation.cfm?id=185728.185756.

[97] Wilfrid Hodges. *Model Theory.* Vol. 42. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1993. URL: https://doi.org/10.1017/cbo9780511551574.

[98] Douglas Hofstadter. *Gödel, Escher, Bach : an eternal golden braid.* New York: Basic Books, 1979.

[99] Christopher John Hogger. *Essentials of Logic Programming.* New York, NY, USA: Oxford University Press, Inc., 1990.

[100] Markus Höhfeld and Gert Smolka. *Definite Relations over Constraint Languages.* Tech. rep. LILOG Report 53. IWBS, 1988.

[101] Alfred Horn. "On sentences which are true of direct unions of algebras". In: *Journal of Symbolic Logic* 16.1 (03/1951), pp. 14–21. URL: https://doi.org/10.2307/2268661.

[102] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. "Datalog and emerging applications: an interactive tutorial". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. Ed. by Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis. ACM, 2011, pp. 1213–1216. URL: https://doi.org/10.1145/1989323.1989456.

[103] R. John Muir Hughes. "A novel representation of lists and its application to the function "reverse"". In: *Information Processing Letters* 22.3 (1986), pp. 141–144. URL: http://www.sciencedirect.com/science/article/pii/0020019086900591.

[104] Serge Le Huitouze. "A new data structure for implementing extensions to Prolog". In: *Programming Language Implementation and Logic Programming*. Springer Science LNCS, 1990, pp. 136–150. URL: http://dx.doi.org/10.1007/bfb0024181.

[105] ISO. "IEC 14882: 2011 Information technology—Programming languages—C++". In: *International Organization for Standardization, Geneva, Switzerland* 27 (2012), p. 59.

[106] Joxan Jaffar and Jean-Louis Lassez. "Constraint Logic Programming". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: ACM, 1987, pp. 111–119. URL: http://doi.acm.org/10.1145/41625.41635.

[107] Joxan Jaffar and Jean-Louis Lassez. "From Unification to Constraints". In: *Proceedings of the Conference on Logic Programming '87*. Ed. by Koichi Furukawa, Hozumi Tanaka, and Tetsunosuke Fujisaki. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 1–18. URL: https://doi.org/10.1007/3-540-19426-6_1.

[108] Joxan Jaffar, Jean-Louis Lassez, and John Lloyd. "Completeness of the negation as failure rule". In: *Proceedings of the 8th International Joint Conference on Artificial Intelligence IJCAI-83*. Vol. 1. Morgan Kaufmann Publishers Inc. 1983, pp. 500–506.

[109] Joxan Jaffar, Jean-Louis Lassez, and Michael J. Maher. "A Logic Programming Language Scheme". In: Doug DeGroot and Gary Lindstrom. *Logic Programming: Relations, Functions, and Equations*. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1986, pp. 441–467.

[110] Joxan Jaffar and Michael J. Maher. "Constraint logic programming: a survey". In: *The Journal of Logic Programming* 19-20 (05/1994), pp. 503–581. URL: http://dx.doi.org/10.1016/0743-1066(94)90033-7.

[111] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. "The Semantics of Constraint Logic Programs". In: *The Journal of Logic Programming* 37.1-3 (10/1998), pp. 1–46. URL: http://dx.doi.org/10.1016/s0743-1066(98)10002-x.

[112] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and Roland H. C. Yap. "Output in CLP($\mathscr{R}$)". In: *Fifth Generation Computing Systems*. 1992, pp. 987–995.

[113] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. "The CLP($\mathscr{R}$) language and system". In: *ACM Transactions on Programming Languages and Systems* 14.3 (1992), pp. 339–395.

[114] Jean-Pierre Jouannaud and Claude Kirchner. "Solving equations in abstract algebras: A rule-based survey of unification". In: *Computational Logic: Essays in Honor of Alan Robinson*. Ed. by Jean-Louis Lassez and Gordon Plotkin. Cambridge, Mass: The MIT Press, 1991. Chap. 8, pp. 257–321.

[115] Mark H. Karwan, Vahid Lotfi, Stanley Zionts, and Jan Telgen. "An Introduction to Redundancy". In: *Redundancy in Mathematical Programming*. Springer Berlin Heidelberg, 1983, pp. 1–13. URL: https://doi.org/10.1007/978-3-642-45535-3_1.

[116] Andrew W. Keep, Michael D. Adams, Lindsey Kuper, William E. Byrd, and Daniel P. Friedman. "A Pattern-matcher for miniKanren -or- How to Get into Trouble with CPS Macros". In: *Proceedings of Scheme Workshop '09, Cal Poly Technical Report CPSLO-CSC-09-03*. 2009, pp. 37–45.

[117] Claude Kirchner, Hélene Kirchner, and Michaël Rusinowitch. "Deduction with symbolic constraints [Research Report] RR-1358". In: *Revue d'intelligence artificielle* 4.1358 (12/1990). Ed. by Ricardo Caffera, pp. 9–52. URL: https://hal.inria.fr/inria-00077103.

[118] Oleg Kiselyov. *The taste of logic programming*. 2006. URL: http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren.

[119] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. "Pure, declarative, and constructive arithmetic relations (declarative pearl)". In: *Proceedings of the 9th International Symposium on Functional and Logic Programming*. Ed. by Jacques Garrigue and Manuel Hermenegildo. Vol. 4989. LNCS. Springer, 2008, pp. 64–80.

[120] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. "Backtracking, interleaving, and terminating monad transformers: (functional pearl)". In: *Proceedings of ICFP '05*. Vol. 40. 9. ACM, 10/2005, pp. 192–203. URL: http://doi.acm.org/10.1145/1086365.1086390.

[121] Stephen Cole Kleene. *Introduction to Metamathematics*. New York: Van Nostrand, 1952.

[122] Kevin Knight. "Unification: A Multidisciplinary Survey". In: *ACM Computing Surveys (CSUR)* 21.1 (1989), pp. 93–124. URL: https://doi.org/10.1145/62029.62030.

[123] Eugene Edmund Kohlbecker. "Syntactic Extensions in the Programming Language Lisp". PhD thesis. 1986. URL: http://www.cs.indiana.edu/ftp/techreports/TR199.pdf.

[124] H. Jan Komorowski. "QLOG: The programming environment for PROLOG in LISP". In: *Logic Programming*. Ed. by Keith L. Clark and Sten-Åke Tärnlund. Automatic Programming Information Centre (Brighton). Studies in data processing no. 16. Academic Press, 1982, pp. 315–324.

[125] Oliver Kowalke and Nat Goodspeed. *call/cc (call-with-current-continuation): A low-level API for stackful context switching*. C++ Standards Group Paper, Document number: P0534R3. 2017. URL: open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0534r3.pdf.

[126] Robert A. Kowalski. *Logic for problem solving*. Vol. 7. The computer science library: Artificial intelligence series. New York: North-Holland, 1979. URL: http://www.worldcat.org/oclc/05564433.

[127] Robert A. Kowalski. "Predicate Logic as Programming Language". In: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*. Ed. by Jack L. Rosenfeld. Amsterdam: North-Holland, 1974, pp. 569–574.

[128] Robert A. Kowalski, Francesca Toni, and Gerhard Wetzel. "Executing suspended logic programs". In: *Fundamenta Informaticae* 34.3 (1998), pp. 203–224.

[129] Robert Kowalski, Francesca Toni, and Gerhard Wetzel. "Towards a declarative and efficient glass-box CLP language". In: *Workshop Logische Programmierung*. 1994. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.1002&rep=rep1&type=pdf.

[130] Shriram Krishnamurthi. "Teaching Programming Languages in a Post-Linnaean Age". In: *SIGPLAN Notices* 43.11 (2008), pp. 81–83.

[131] Frank Kriwaczek. "An Introduction to Constraint Logic Programming". In: *Advanced Topics in Artificial Intelligence*. Springer, 1992, pp. 82–94.

[132] Kenneth Kunen. "Negation in logic programming". In: *The Journal of Logic Programming* 4.4 (12/1987), pp. 289–308. URL: https://doi.org/10.1016/0743-1066(87)90007-0.

[133] Kenneth Kunen. "Signed Data Dependencies in Logic Programs". In: *The Journal of Logic Programming* 7.3 (1989), pp. 231–245. URL: http://www.sciencedirect.com/science/article/pii/0743106689900228.

[134] Catherine Lassez. "Constraint Logic Programming". In: *Byte* 12.9 (08/1987), pp. 171–176.

[135] Jean-Louis Lassez. "From LP to LP: Programming with Constraints". In: *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings*. Ed. by Takayasu Ito and Albert R. Meyer. Vol. 526. Lecture Notes in Computer Science. Springer, 1991, pp. 420–446. URL: https://doi.org/10.1007/3-540-54415-1_57.

[136] Jean-Louis Lassez. "Querying Constraints". In: *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*. Ed. by Daniel J. Rosenkrantz and Yehoshua Sagiv. ACM Press, 1990, pp. 288–298. URL: https://doi.org/10.1145/298514.298581.

[137] Jean-Louis Lassez, Michael J. Maher, and Kim Marriott. "Unification Revisited". In: *Foundations of Logic and Functional Programming*. Springer Berlin Heidelberg, 1988, pp. 67–113. URL: https://doi.org/10.1007/3-540-19129-1_4.

[138] Jean-Louis Lassez and Ken McAloon. "A Constraint Sequent Calculus". In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society Press, 1990, pp. 52–61. URL: https://doi.org/10.1109/LICS.1990.113733.

[139] Jean-Louis Lassez and Ken McAloon. "Independence of Negative Constraints". In: *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'89)*. Ed. by Josep Díaz and Fernando Orejas. Vol. 351. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1989, pp. 19–27. URL: https://doi.org/10.1007/3-540-50939-9_122.

[140] Thierry Le Provost and Mark Wallace. "Generalized constraint propagation over the CLP scheme". In: *The Journal of Logic Programming* 16.3 (1993), pp. 319–359.

[141] Pierre Lim and Peter J. Stuckey. "A constraint logic programming shell". In: *Programming Language Implementation and Logic Programming*. Springer. 1990, pp. 75–88.

[142] John Wylie Lloyd. *Declarative programming in Escher*. Tech. rep. CSTR-95-013. Department of Computer Science, University of Bristol, 1995.

[143] John Wylie Lloyd. *Foundations of Logic Programming*. 1st. Springer, 1984.

[144] John Wylie Lloyd. *Foundations of Logic Programming*. 2nd. Berlin, Heidelberg: Springer-Verlag, 1987.

[145] John Wylie Lloyd and John C. Shepherdson. "Partial evaluation in logic programming". In: *The Journal of Logic Programming* 11.3-4 (10/1991), pp. 217–242. URL: http://dx.doi.org/10.1016/0743-1066(91)90027-m.

[146] Kuang-Chen Lu, Weixi Ma, and Daniel P Friedman. "Towards a miniKanren with fair search strategies". In: *Proceedings of the 2019 miniKanren and Relational Programming Workshop*. TR-02-19. Cambridge, Massachusetts, 2019, pp. 1–15. URL: dash.harvard.edu/bitstream/handle/1/41307116/tr-02-19.pdf.

[147] Michael J. Maher. "A Logic Programming View of CLP". In: *Logic Programming, Proceedings of the Tenth International Conference on Logic Programming, Budapest, Hungary, June 21-25, 1993*. Ed. by David Scott Warren. The MIT Press, 1993, pp. 737–753. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.6946.

[148] Michael J. Maher. "Adding Constraints to Logic-based Formalisms". In: *The Logic Programming Paradigm: A 25-Year Perspective*. Ed. by Krzysztof R. Apt, Victor W. Marek, Mirek Truszczynski, and David S. Warren. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 313–331. URL: https://doi.org/10.1007/978-3-642-60085-2_13.

[149] Michael J. Maher. “Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees”. In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 1988, pp. 348–357. URL: https://doi.org/10.1109/LICS.1988.5132.

[150] Johann A. Makowsky. “Why Horn Formulas Matter in Computer Science: Initial Structures and Generic Examples”. In: *Proceedings Of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) Berlin, March 25-29, 1985 on Mathematical Foundations of Software Development, Vol. 1: Colloquium on Trees in Algebra and Programming (CAAP'85)*. CAAP '85. Berlin, Germany: Springer-Verlag, 1985, pp. 374–387. URL: http://dl.acm.org/citation.cfm?id=21855.21878.

[151] Johann A. Makowsky. “Why Horn Formulas Matter in Computer Science: Initial Structures and Generic Examples”. In: *Journal of Computer and System Sciences* 34.2-3 (1987), pp. 266–292. URL: http://www.sciencedirect.com/science/article/pii/0022000087900274.

[152] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. Cambridge, Mass: The MIT Press, 1998.

[153] Alberto Martelli and Ugo Montanari. “An Efficient Unification Algorithm”. In: *ACM Transactions on Programming Languages and Systems* 4.2 (04/1982), pp. 258–282. URL: https://doi.org/10.1145/357162.357169.

[154] Gianfranco Mascari and Antonio Vincenzi. “Model-theoretic Specifications and Back-and-forth Equivalences”. In: *Recent Trends in Data Type Specification*. Ed. by H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel. Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 166–184. URL: https://doi.org/10.1007/3-540-54496-8_9.

[155] Elliott Mendelson. *Introduction to Mathematical Logic*. 6th. Boca Raton: CRC Press/Taylor & Francis Group, 06/2015.

[156] Pedro Meseguer. “Interleaved Depth-First Search”. In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*. Morgan Kaufmann, 1997, pp. 1382–1387. URL: http://ijcai.org/Proceedings/97-2/Papers/085.pdf.

[157] Juan Jose Moreno-Navarro and Mario Rodriguez-Artalejo. “Logic Programming with Functions and Predicates: The Language Babel”. In: *The Journal of Logic Programming* 12.3 (1992), pp. 191–223. URL: http://www.sciencedirect.com/science/article/pii/074310669290024W.

[158] Lawrence S. Moss. “Logics for Two Fragments beyond the Syllogistic Boundary”. In: *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*. Ed. by Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 538–564. URL: https://doi.org/10.1007/978-3-642-15025-8_27.

[159] Lee Naish. “A three-valued semantics for logic programmers”. In: *TPLP* 6.5 (2006), pp. 509–538. URL: https://doi.org/10.1017/S1471068406002742.

[160] Lee Naish. “Adding equations to NU-Prolog”. In: *Programming Language Implementation and Logic Programming*. Ed. by Jan Maluszyński and Martin Wirsing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 15–26.

[161] Lee Naish. “Prolog Control Rules”. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*. Ed. by Aravind K. Joshi. Morgan Kaufmann, 1985, pp. 720–722. URL: http://ijcai.org/Proceedings/85-2/Papers/006.pdf.

[162] Lee Naish. *Pruning in logic programming*. Tech. rep. Technical Report 95/16. Melbourne, Australia: Department of Computer Science, University of Melbourne, 06/1995.

[163] Lee Naish and Harald Søndergaard. “Truth versus information in logic programming”. In: *TPLP* 14.6 (2014), pp. 803–840. URL: https://doi.org/10.1017/S1471068413000069.

[164] Joseph P. Near, William E. Byrd, and Daniel P. Friedman. "αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic". In: *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*. Ed. by Maria Garcia de la Banda and Enrico Pontelli. Vol. 5366. Lecture Notes in Computer Science. Springer, 2008, pp. 238–252. URL: https://doi.org/10.1007/978-3-540-89982-2_26.

[165] Martin Nilsson. "The World's Shortest Prolog Interpreter?" In: *Implementations of Prolog*. Ed. by John A. Campbell. Chichster, England: Ellis Horwood, 1984, pp. 87–92.

[166] Pilar Nivela and Fernando Orejas. "Initial Behaviour Semantics for Algebraic Specifications". In: *Recent Trends in Data Type Specification, 5th Workshop on Abstract Data Types, Gullane, Scotland, UK, September 1-4, 1987, Selected Papers*. Ed. by Donald Sannella and Andrzej Tarlecki. Vol. 332. Lecture Notes in Computer Science. Springer, 1987, pp. 184–207. URL: https://doi.org/10.1007/3-540-50325-0_10.

[167] Erik Palmgren. "Denotational Semantics of Constraint Logic Programming - A Nonstandard Approach". In: *Constraint Programming, Proceedings of the NATO Advanced Study Institute on Constraint Programming, Parnu, Estonia, August 13-24, 1993*. Ed. by Brian H. Mayoh, Enn Tyugu, and Jaan Penjam. Vol. 131. NATO ASI Series. Springer, 1993, pp. 261–288. URL: https://doi.org/10.1007/978-3-642-85983-0_10.

[168] Erik Palmgren and Viggo Stoltenberg-Hansen. "Logically Presented Domains". In: *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26-29, 1995*. IEEE. IEEE Computer Society, 1995, pp. 455–463. URL: https://doi.org/10.1109/LICS.1995.523279.

[169] Aiqin Pan and Barrett R Bryant. "Logic programming implementation of functional programming languages". eng. In: *TENCON '89. Fourth IEEE Region 10 International Conference. INFORMATION TECHNOLOGIES FOR THE 90's".*

E^2C^2; ENERGY, ELECTRONICS, COMPUTERS, COMMUNICATIONS. Nov 22-24, 1989 Bombay, India. IEEE. IEEE, 1989, pp. 174–178. URL: http://xplorestaging.ieee.org/ielx2/843/4471/00176865.pdf?arnumber=176865.

[170] Michael S. Paterson and Mark N. Wegman. "Linear unification". In: *Journal of Computer and System Sciences* 16.2 (04/1978), pp. 158–167. URL: https://doi.org/10.1016/0022-0000(78)90043-0.

[171] Francis Jeffry Pelletier and Allen P. Hazen. "A History of Natural Deduction". In: *Handbook of the History of Logic*. Elsevier, 2012, pp. 341–414. URL: https://doi.org/10.1016/b978-0-444-52937-4.50007-1.

[172] Alan J. Perlis. "Epigrams on Programming". In: *SIgPLAN Notices* 17.9 (1982), pp. 7–13.

[173] Dag Prawitz. "An improved proof procedure 1". In: *Theoria* 26.2 (1960), pp. 102–139.

[174] Eric S Raymond. *The New Hacker's Dictionary*. 3rd. Cambridge, Massachusetts, USA: MIT Press, 1996.

[175] Raymond Reiter. "On Closed World Data Bases". In: *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*. Ed. by Hervé Gallaire and Jack Minker. Advances in Data Base Theory. New York: Plemum Press, 1977, pp. 55–76. URL: https://doi.org/10.1007/978-1-4684-3384-5_3.

[176] Graem A. Ringwood. "SLD: a folk acronym?" In: *ACM SIGPLAN Notices* 24.5 (05/1989), pp. 71–75. URL: https://doi.org/10.1145/66068.66074.

[177] John Alan Robinson. "A machine-oriented logic based on the resolution principle". In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41.

[178] John Alan Robinson. "Beyond LOGLISP: Combining Functional and Relational Programming in a Reduction Setting". In: *Machine Intelligence 11*. Ed. by J. E. Hayes, D. Michie, and J. Richards. New York, NY, USA: Oxford University Press, Inc., 1988, pp. 57–68. URL: http://dl.acm.org/citation.cfm?id=60769.60772.

[179] John Alan Robinson. “Computational logic: Memories of the past and challenges for the future”. In: *Computational Logic—CL 2000*. Springer, 2000, pp. 1–24.

[180] John Alan Robinson. “Logic programming —Past, present and future—”. In: *New Generation Computing* 1.2 (06/1983), pp. 107–124. URL: https://doi.org/10.1007/bf03037419.

[181] John Alan Robinson and Ernest E. Silbert. *Logic Programming in LISP*. Tech. rep. RADC-TR-80-379-VOL-1. Rome Air Development Center / Syracuse University, School of Computer & Information Science, 01/1981. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a096042.pdf.

[182] John Alan Robinson and Ernest E. Silbert. “LOGLISP: an alternative to PROLOG”. In: *Machine Intelligence 10*. Ed. by J. E. Hayes, Donald Michie, and Y.-H. Pao. Ellis Horwood, 1982. Chap. 20, pp. 399–419.

[183] Gregory Rosenblatt, Lisa Zhang, William E Byrd, and Matthew Might. “First-order miniKanren representation: Great for tooling and search”. In: *Proceedings of the 2019 miniKanren and Relational Programming Workshop*. TR-02-19. Cambridge, Massachusetts, 2019, pp. 20–34. URL: dash.harvard.edu/bitstream/handle/1/41307116/tr-02-19.pdf.

[184] Francesca Rossi. “Constraint (Logic) Programming: A Survey on Research and Applications”. In: *New Trends in Contraints, Joint ERCIM/Compulog Net Workshop, Paphos, Cyprus, October 25-27, 1999, Selected Papers*. Ed. by Krzysztof R. Apt, Antonis C. Kakas, Eric Monfroy, and Francesca Rossi. Vol. 1865. Lecture Notes in Computer Science. Springer, 1999, pp. 40–74. URL: https://doi.org/10.1007/3-540-44654-0_3.

[185] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. “Certified Semantics for miniKanren”. In: *Proceedings of the 2019 miniKanren and Relational Programming Workshop*. TR-02-19. Cambridge, Massachusetts, 2019, pp. 80–98. URL: dash.harvard.edu/bitstream/handle/1/41307116/tr-02-19.pdf.

[186] Amr Sabry. "Declarative Programming Across the Undergraduate Curriculum". In: Technical Report 99-346. Section 3 of [58]. 08/1999. URL: http://www.ccs.neu.edu/home/matthias/FDPE99/.

[187] Taisuke Sato. "Completed logic programs and their consistency". In: *The Journal of Logic Programming* 9.1 (1990), pp. 33–44. URL: http://www.sciencedirect.com/science/article/pii/074310669090032Z.

[188] Tom Schrijvers, Peter J. Stuckey, and Philip Wadler. "Monadic constraint programming". In: *Journal of Functional Programming* 19.06 (08/2009), p. 663. URL: http://dx.doi.org/10.1017/s0956796809990086.

[189] Ryan Senior. "Practical core.logic". In: *Clojure/West*. San Jose, Califoria, 03/2012. URL: infoq.com/presentations/core-logic.

[190] Silvija Seres. "The algebra of logic programming". PhD thesis. University of Oxford, UK, 2001. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.365466.

[191] Silvija Seres, J. Michael Spivey, and C. A. R. Hoare. "Algebra of Logic Programming". In: *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*. Ed. by Danny De Schreye. MIT Press, 1999, pp. 184–199.

[192] Peter Sestoft. *The Structure of a Self-applicable Partial Evaluator*. Tech. rep. DIKU Report 85-11. Institute of Datalogy, University of Copenhagen. URL: http://www.itu.dk/~sestoft/papers/Sestoft-DIKU-report-85-11.pdf.

[193] John C Shepherdson. "Negation as Failure II". In: *The Journal of Logic Programming* 2.3 (1985), pp. 185–202. URL: http://www.sciencedirect.com/science/article/pii/0743106685900184.

[194] John C Shepherdson. "Negation as failure, completion and stratification". In: *Logic Programming*. Ed. by John A Robinson Dov M Gabbay CJ Hogger. Vol. 5. Handbook of Logic in Artificial Intelligence and Logic Programming. 1998, pp. 355–419.

[195] John C Shepherdson. "Negation in logic programming". In: *Foundations of deductive databases and logic programming*. Ed. by Jack Minker. Los Altos, California: Elsevier, 1988, pp. 19–88.

[196] Olin Shivers. *List Library. Scheme Request for Implementation. SRFI-1*. 1999. URL: http://srfi.schemers.org/srfi-1/srfi-1.html.

[197] Jörg H. Siekmann, ed. *Computational Logic*. Vol. 9. Handbook of the History of Logic Edited by Dov M. Gabbay, Jörg H. Siekmann, and John Woods. Amsterdam, Boston: North-Holland, 2014. URL: https://doi.org/10.1016/c2009-0-16676-x.

[198] Jörg H. Siekmann. "Unification theory". In: *Journal of Symbolic Computation* 7.3-4 (03/1989), pp. 207–274. URL: https://doi.org/10.1016/s0747-7171(89)80012-4.

[199] Ben A. Sijtsma. "On the Productivity of Recursive List Definitions". In: *ACM Transactions on Programming Languages and Systems* 11.4 (10/1989), pp. 633–649. URL: http://doi.acm.org/10.1145/69558.69563.

[200] Donald A Smith. "Constraint Operations for CLP($\mathscr{F}\mathscr{T}$)." In: *Proceedings of ICLP '91*. 1991, pp. 760–774.

[201] Donald A. Smith and Timothy J. Hickey. "Partial Evaluation of a CLP Language". In: *Logic Programming, Proceedings of the 1990 North American Conference, Austin, Texas, USA, October 29 - November 1, 1990*. Ed. by Saumya K. Debray and Manuel V. Hermenegildo. MIT Press, 1990, pp. 119–138.

[202] J. Michael Spivey and Silvija Seres. "Embedding Prolog in Haskell". In: *Proceedings of Haskell Workshop '99, Utrecht University Technical Report UU-CS-1999-28*. Ed. by E. Meier. Vol. 99. 1999. URL: http://www.cs.uu.nl/research/techreps/repo/CS-1999/1999-28.pdf.

[203] Guy Lewis Steele Jr. "It's Time for a New Old Language". In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*. Ed. by Vivek Sarkar and Lawrence Rauchwerger. ACM, 2017, p. 1. URL: http://dl.acm.org/citation.cfm?id=3018773.

[204] Guy Lewis Steele Jr. "The definition and implementation of a computer programming language based on constraints". PhD thesis. Massachusetts Institute of Technology, 1980.

[205] Ivan E. Sutherland. "Sketch-Pad: A Man-machine Graphical Communication System". In: *Proceedings of the SHARE Design Automation Workshop*. DAC '64. New York, NY, USA: ACM, 1964, pp. 6.329–6.346. URL: http://doi.acm.org/10.1145/800265.810742.

[206] Bruce A. Tate, Ian Dees, Frederic Daoud, and Jack Moffitt. *Seven More Languages in Seven Weeks: Languages That Are Shaping the Future*. Pragmatic Bookshelf, 2014.

[207] Emina Torlak and Rastislav Bodik. "Growing solver-aided languages with Rosette". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software - Onward! '13*. ACM. 2013, pp. 135–152. URL: https://doi.org/10.1145/2509578.2509586.

[208] Sauro Tulipani. "Decidability of the existential theory of infinite terms with subterm relation". In: *Information and Computation* 108.1 (01/1994), pp. 1–33. URL: https://doi.org/10.1006/inco.1994.1001.

[209] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. "Nominal unification". In: *Theoretical Computer Science* 323.1-3 (09/2004), pp. 473–497. URL: https://doi.org/10.1016%5C%2Fj.tcs.2004.06.016.

[210] Alasdair Urquhart. "Emil Post. Logic from Russell to Church". In: *The Handbook of the History of Logic*. Ed. by Dov M. Gabbay and John Woods. Vol. 5. Elsevier, 2009, pp. 5–617.

[211] Maarten H. van Emden and Robert A. Kowalski. "The Semantics of Predicate Logic as a Programming Language". In: *Journal of the ACM (JACM)* 23.4 (1976), pp. 733–742. URL: https://doi.org/10.1145/321978.321991.

[212] Pascal Van Hentenryck. "Constraint logic programming". In: *The Knowledge Engineering Review* 6.3 (09/1991), pp. 151–194. URL: https://doi.org/10.1017/s0269888900005798.

[213] Pascal Van Hentenryck and Viswanath Ramachandran. "Backtracking Without Trailing in CLP($\mathscr{R}_{lin}$)". In: *ACM Transactions on Programming Languages and Systems* 17.4 (07/1995), pp. 635–671. URL: http://doi.acm.org/10.1145/210184.210192.

[214] Thomas Vasak. "A survey of control facilities in logic programming". In: *Australian Computer Journal* 18.3 (1986), pp. 136–145.

[215] Marcel Lodewijk Johanna van de Vel. "Theories with the Independence Property". In: *Studia Logica* 95.3 (), pp. 379–405. URL: https://doi.org/10.1007/s11225-010-9263-5.

[216] K. N. Venkataraman. "Decidability of the Purely Existential Fragment of the Theory of Term Algebras". In: *Journal of the ACM* 34.2 (04/1987), pp. 492–510. URL: http://doi.acm.org/10.1145/23005.24037.

[217] Hugo Volger. *On Theories Which Admit Initial Structures*. Tech. rep. Universität Passau, 1987.

[218] Philiip Wadler, Walid Taha, and David MacQueen. "How to add laziness to a strict language without even being odd". In: *SML'98, Workshop on Standard ML*. Baltimore, 09/26/1998.

[219] Mark Wallace. "Constraint Logic Programming". In: *Computational Logic: Logic Programming and Beyond*. Springer Science LNCS, 2002, pp. 512–532. URL: http://dx.doi.org/10.1007/3-540-45628-7_19.

[220] Mark Wallace. "Tight, consistent, and computable completions for unrestricted logic programs". In: *The Journal of Logic Programming* 15.3 (1993), pp. 243–273. URL: http://www.sciencedirect.com/science/article/pii/074310669390041E.

[221] Richard S. Wallace. "An easy implementation of PiL (Prolog in Lisp)". In: *SIGART Newsletter* 85 (07/1983), pp. 29–32. URL: https://doi.org/10.1145/1056635.1056638.

[222] Mitchell Wand. *A semantic algebra for logic programming*. Tech. rep. 148. Department of Computer Science, 08/1983.

[223] Mitchell Wand and Dale Vaillancourt. “Relating Models of Backtracking”. In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’04. Snow Bird, UT, USA: ACM, 2004, pp. 54–65. URL: http://doi.acm.org/10.1145/1016850.1016861.

[224] Martin P Ward. “Language-oriented programming”. In: *Software-Concepts and Tools* 15.4 (1994), pp. 147–161.

Jason Hemann

E-Mail: jhemann@iu.edu Website: hemann.pl

EDUCATION

2020 **PhD in Computer Science**
Indiana University (Bloomington, IN, USA)
Minor: Logic, *Certificate in Logic*
Advisor: Dan Friedman

2012 **MS in Computer Science**
Indiana University (Bloomington, IN, USA)
Advisor: Dan Friedman

2007 **BS in Computer Science, Philosophy**, *cum laude*
Trinity University (San Antonio, TX, USA)
Advisor: Paul Myers

2007 **BA in History**, *cum laude*
Trinity University (San Antonio, TX, USA)

TEACHING EXPERIENCE

2018–2019 **Clinical Instructor**, College of Comp. & Info. Science
Northeastern University (Boston, MA, USA)
Programming Langs, *CS II*, *Software Dev*

2017–2018 **Visiting Faculty**, Department of CS & SE
Rose-Hulman Inst. Tech. (Terre Haute, IN, USA)
Programming Langs

2012–2017 **Course Admin, Instructor** Department of Computer Science
Indiana University (Bloomington, IN, USA)
Programming Langs, *Programming Langs (Graduate)*

June 2014 **Instructor**, uCombinator Lab
University of Utah (Salt Lake, UT, USA)
miniKanren Summer School

Summer 2013 **Instructor**, Foundations in Science and Mathematics
Indiana University (Bloomington, IN, USA)
Intro to CS

2010–2012 **Lab Instructor**, Department of Computer Science
Indiana University (Bloomington, IN, USA)
Discrete Math for CS, *Theory of Comp*, *CS for Non-Majors*

PROFESSIONAL SERVICE

2019	miniKanren Workshop (Program Committee)
2019	Scheme Workshop (Program Committee)
2016	Scheme Workshop (Bursar)
2015	ICFP (Student Volunteer)
2014	Scheme Workshop (General Chair, Publicity Chair)

Reviewing

ICFP, CPP, miniKanren Workshop, Scheme Workshop, ML Workshop

STEM Education Outreach

2013–2017 **Foundations in Science and Mathematics (FSM)**
Indiana University (Bloomington, IN, USA)

- Grant Writing Team (2015–2017)
- Program Administration (2015–2017)
- Computer Science Program Lead (2014–2017)
- Computer Science Course Instructor (2013)

AWARDS & RECOGNITIONS

Travel Awards

2017	IJCAI-17 Travel Grant
2017	CP/ICLP/SAT DP Travel Award
2016	PLMW Travel Award for SPLASH 2016
2016	SIGPLAN PAC Travel Award for SPLASH 2016
2016	Scheme 2016 Travel Scholarship
2016	SIGPLAN PAC Travel Award for ICFP 2016
2015	SIGPLAN PAC Travel Award for ICFP 2015
2015	NSF Travel Award for ECOOP 2015
2014	FLoC Travel Award for Vienna Summer of Logic
2014	ICLP Summer School Travel Award for Vienna Summer of Logic
2013	Scheme 2013 Travel Scholarship
2013	SIGPLAN PAC Travel Award for ICFP 2013
2012	NASSLLI 2012 Travel Award
2012	OPLSS Housing Grant
2012	NECSS 2012 Student Sponsorship

Other Awards

2017	Women's Philanthropy Leadership Council Award (FSM Team)
2015	SOIC Associate Instructor of the Year

PUBLICATIONS

Books & Dissertations

[BD1] **Jason Hemann**. "Constraint microKanren in the CLP Scheme". PhD thesis. Indiana University, Bloomington, 01/2020.

[BD2] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and **Jason Hemann**. *The Reasoned Schemer, 2nd Edition*. The MIT Press, 01/2018. URL: mitpress.mit.edu/books/reasoned-schemer-0.

Journal & Selective Conference Papers

[JC1] Daniel Schwab, Logan Cole, Karna Desai, **Jason Hemann**, Kate Hummels, and Adam Maltese. "A Summer Stem Outreach Program Run By Graduate Students: Successes, Challenges, And Recommendations For Implementation". In: *Journal of Research in STEM Education* 4 (2 12/2018), pp. 117–129.

[JC2] **Jason Hemann**, Daniel P. Friedman, William E. Byrd, and Matthew Might. "A Small Embedding of Logic Programming with a Simple Complete Search". In: *Proc. of DLS'16*. Amsterdam, Netherlands: ACM, 11/2016. URL: dx.doi.org/10.1145/2989225.2989230.

Workshop Papers & Technical Reports

[W1] **Jason Hemann** and Daniel P. Friedman. "A Framework for Extending microKanren with Constraints". In: *Joint Proc of WLP'15/'16/WFLP'16 29th*. Ed. by Sibylle Schwarz and Janis Voigtländer. Vol. 234. EPTCS. Open Publishing Association, 01/2017, pp. 135–149. URL: eptcs.web.cse.unsw.edu.au/content.cgi?WFLP2016.

[W2] **Jason Hemann** and Daniel P. Friedman. "Deriving Pure, Functional One-Pass Operations for Processing Tail-Aligned Lists". In: *Proc. of Scheme '16*. Nara, Japan, 09/2016. URL: scheme2016.snow-fort.org/static/scheme16-paper6.pdf.

[W3] **Jason Hemann** and John Clements, eds. *Proceedings of the 2014 Workshop on Scheme and Functional Programming, Indiana University Technical Report TR718*. Washington, D.C., USA: Computer Science Department, Indiana University, 09/2015. URL: cs.indiana.edu/pub/techreports/TR718.pdf.

[W4] **Jason Hemann** and Daniel P. Friedman. "A Framework for Extending microKanren with Constraints". In: *Proc. of Scheme '15, Northeastern University Technical Report NU-CCIS-2016-001*. Ed. by Andrew W. Keep and Ryan Culpepper. 09/2015. URL: http://hdl.handle.net/2047/D20213213.

[W5] **Jason Hemann**, Cameron Swords, and Lawrence S Moss. "Two Advances in the Implementations of Extended Syllogistic Logics". In: *Joint Proc. of NLPAR'15/LNMR'15*. Ed. by Marcello Balduccini, Alessandra Mileo, Ekaterina Ovchinnikova, Alessandra Russo, and Peter Schüller. Lexington, Kentucky, USA, 09/2015, pp. 1–15. URL: peterschueller.com/pub/2015/nlpar2015-proceedings.pdf.

[W6] Daniel Brady, **Jason Hemann**, and Daniel P. Friedman. "Little Languages for Relational Programming". In: *Proc of Scheme '14, Indiana University Technical Report TR718*. Washington, D.C., USA, 09/2015, pp. 54–64. URL: cs.indiana.edu/pub/techreports/TR718.pdf.

[W7] **Jason Hemann** and Daniel P. Friedman. "μKanren: A Minimal Functional Core for Relational Programming". In: *Proc. of Scheme '13*. Digital. Alexandria, Virginia, USA, 11/2013. URL: schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf.

[W8] **Jason Hemann** and Daniel P. Friedman. "λ^*: Beyond Currying". In: *Proc. of Scheme '13*. Digital. Alexandria, Virginia, USA, 11/2013. URL: schemeworkshop.org/2013/papers/HemannCurrying2013.pdf.

[W9] **Jason Hemann** and Eric Holk. "Visualizing the Turing Tarpit". In: *Proc. of FARM '13*. FARM '13. Boston, Massachusetts, USA: ACM, 2013, pp. 71–76. URL: doi.acm.org/10.1145/2505341.2505348.

[W10] **Jason Hemann**, Fatma Mili, and Paul Myers. "Synchronized Energy Efficient Clustering of Wireless Sensor Networks". In: *Proc. of NCUR 2007*. San Rafael, California, 04/2007. URL: ncurproceedings.org/ojs/.

Presentations & Demonstrations

[PD1] Daniel P. Friedman and **Jason Hemann**. "Implementing a microKanren". In: *CodeMesh 2016*. London, England, 11/2016. URL: http://youtube.com/watch?v=0FwIwewHC3o.

[PD2] Daniel P. Friedman and **Jason Hemann**. "From Functions To Relations in miniKanren". In: *Øredev 2015*. Malmö, Sweden, 11/2015. URL: vimeo.com/144710533.

[PD3] Daniel P. Friedman and **Jason Hemann**. "Generating a Quine". In: *Midwest PL Summit '15*. West Lafayette, Indiana, USA, 12/2015.

[PD4] Daniel P. Friedman and **Jason Hemann**. "How to be a Good Host: miniKanren as a Case Study". In: *Curry On 2015*. Prague, Czech Republic, 07/2015. URL: youtube.com/watch?v=b9C3r3dQnNY.

[PD5] Daniel P. Friedman and **Jason Hemann**. "Rapidly Rolling a Relational DSL". In: *Øredev 2015*. Malmö, Sweden, 11/2015. URL: vimeo.com/144988186.

[PD6] Daniel P. Friedman and **Jason Hemann**. "Roll Your Own Relational DSL: A Logic Programming Language in Less than 40 Lines". In: *Lambda Jam 2014*. Chicago, Illinois, USA, 07/2014.

[PD7] Daniel P. Friedman and **Jason Hemann**. "Write the Other Half of Your Program: From Functional to Logic". In: *Strange Loop 2014*. St. Louis, Missouri, USA, 09/2014. URL: youtube.com/watch?v=RG9fBbQrVOM.

[PD8] Daniel P. Friedman and **Jason Hemann**. "It's Only Quine Time". In: *Programming Languages Fest*. Bloomington, Indiana, USA, 10/2013. URL: web.archive.org/web/20140113225905/lambda.soic.indiana.edu/programming-languages-fest.

[PD9] Daniel P. Friedman and **Jason Hemann**. "The Art of Several Interpreters, Quickly". In: *Lambda Jam 2013*. Chicago, Illinois, USA, 07/2013.

[PD10] **Jason Hemann**. "A Typed Trivalent Logic to Resolve Category Mistakes". In: *North Georgia Student Philosophy Conference*. Kennesaw, Georgia, USA, 04/2007.

Doctoral Consortia

[DC1] **Jason Hemann**, Daniel P. Friedman, William E. Byrd, and Matt Might. "A Simple Complete Search for Logic Programming". In: *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*. Ed. by Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei. Vol. 58. Melbourne, Australia: OASIcs, 2018, 14:1–14:8. URL: dagstuhl.de/dagpub/978-3-95977-058-3.

Posters

[Po1] **Jason Hemann**, Daniel P. Friedman, William E. Byrd, and Matt Might. *A Small Embedding of Logic Programming with a Simple Complete Search*. Poster presented at SPLASH '16, Nov. 2, 2016, Amsterdam, The Netherlands. 2016.

[Po2] Karna Desai, Jing Yang, and **Jason Hemann**. *Foundations in Science and Mathematics Program for Middle School and High School Students*. Poster presented at AAS Meeting #227, Jan. 4-8, 2016, Kissimmee, Florida, USA. 2016. URL: adsabs.harvard.edu/abs/2016AAS...22724612D.

Panels

[Pa1] Deyaaeldeen Almahallawi, **Jason Hemann**, Rin Metcalf, and Fatemeh Sharifi. *Associate Instructor Panel*. Panel Discussion IU SOIC Graduate Recruiting, Feb. 24, 2017, Bloomington, Indiana, USA. 2017.

[Pa2] Charles Pope and **Jason Hemann**. *Associate Instructor Panel*. Panel Discussion at IU SOIC Associate Instructor Training, Sept. 11, 2015, Bloomington, Indiana, USA. 2015.

Interviews

[I1] **Jason Hemann**. *CodeMesh 2016 Talk Interview*. Interview with Eric Normand of PurelyFunctional, Oct. 12, 2016. 2016. URL: purelyfunctional.tv/speaker-interview/jason-hemann-code-mesh-2016-interview/.

[I2] **Jason Hemann**. *Pre-conj Scheme '14 Interview*. Interview with Eric Normand of LispCast, Nov. 17, 2014. 2014. URL: lispcast.com/pre-conj-scheme-workshop.