DECLARATIVE PROGRAMMING VIA TERM REWRITING

by
Matthew Francis-Landau

A dissertation submitted to The Johns Hopkins University in conformity
with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
February 2024

Abstract

I present a new approach to implementing weighted logic programming languages. I first present a bag-relational algebra that is expressive enough to capture the desired denotational semantics, directly representing the recursive conjunctions, disjunctions, and aggregations that are specified by a source program. For the operational semantics, I develop a term-rewriting system that executes a program by simplifying its corresponding algebraic expression.

I have used this approach to create the first complete implementation of the Dyna programming language. A Dyna program consists of rules that define a potentially infinite and cyclic computation graph, which is queried to answer data-dependent questions. Dyna is a unified declarative framework for machine learning and artificial intelligence researchers that supports dynamic programming, constraint logic programming, reactive programming, and object-oriented programming. I have further modernized Dyna to support functional programming with lambda closures and embedded domain-specific languages.

The implementation includes a front-end that translates Dyna programs to bag-relational expressions, a Python API, hundreds of term rewriting rules, and a procedural engine for determining which rewrite rules to apply. The rewrite rules generalize techniques used in constraint logic programming. In practice, our system is usually able to provide simple answers to queries.

Mixing disparate programming paradigms is not without challenges. We had to rethink the classical techniques used to implement logic programming languages. This includes the development of a novel approach for memoization (dynamic programming) that supports partial memoization of fully or partially simplified algebraic expressions, which may contain delayed, unevaluated constraints. Furthermore, real-world Dyna programs require fast and efficient execution. For this reason, I present a novel approach to just-in-time (JIT) compile sequences of term rewrites using a custom tracing JIT.

Thesis Readers

Dr. Jason Eisner (Primary Advisor)
Professor of Computer Science
Department of Computer Science
Johns Hopkins University

Dr. Michael Hanus
Professor of Computer Science
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Dr. Scott Smith
Professor of Computer Science
Department of Computer Science
Johns Hopkins University

I dedicate this to Angela and our

posse of fluffy pets who

fill every day with joy.

Acknowledgements

This dissertation is the result of several years of work that would have been impossible without the help of several others.

I want to thank my Ph.D. advisor, Jason Eisner. Jason has been a guiding force throughout my entire Ph.D. He has made himself consistently available for discussion around research and pushed to make the work as general as possible. It is undeniable that without Jason's influence, I would not explored the term rewriting formalisms presented in this dissertation.

I also want to thank Tim Vieira and Nathaniel Wes Filardo, who also worked on the Dyna project. Diving into the depths of this research would have been unbearably lonely had it not been for Tim and Wes, who collaborated on these publications and also participated in countless hours of discussion throughout the entire process.

For this dissertation, I would like to thank my committee of Jason Eisner, Michael Hanus, and Scott Smith, who read through the 400 page draft of this dissertation. Their comments have undoubtedly improved the clarity of this document. Any remaining mistakes are my own.

I want thank all of the fellow students in the Argo research and in the broader CLSP community for making Baltimore a welcoming environment for these last few years: Jacob Buckman, Tóngfēi Chén, Ryan Cotterell, Leo Du, Seth Ebner, Nathaniel Wes Filardo, Lisa Li, Chu-Cheng Lin, Brian Lu, Teodor Marinov, Becky Marvin, Arya McCarthy, Hongyuan Mei, Sabrina Mielke, Pamela Shapiro, Tim Vieira, Dingquan Wang, Zach Wood-Doughty, and Patrick Xia.

Finally, I want to thank Greg Durrett, Dan Klein, and John Kubiatowicz, who supervised my research during my undergraduate studies. Without their supervision and inspiration, I would have never started the Ph.D. journey.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The promise of *usable* declarative programming has attracted many to the declarative programming paradigm. **Declarative programming** is a paradigm where the program will specify *what* the result of the program should be without specifying *how* the program should perform the computation [100]. This allows the programmer to focus on the task while leaving maximum flexibility to the implementation of the declarative programming language. As such, the declarative paradigm is successful in a number of different domains. For example: databases with SQL [36, 44], optimization and search problems with SMT/SAT [15, 45], and logic programming languages such as Prolog [35, 38] and Datalog [30, 76].

Our group's research is in the space of Declarative Logic Programming designed for Machine Learning (ML) and Artificial Intelligence (AI) Algorithms. This research is called the *Dyna* programming language project, which investigates how Datalog-inspired programming languages can be used to better encode AI algorithms [1, 59].

This dissertation represents the latest iteration of the Dyna programming language project. The work here is based on the culminated learning from three different implementations of Dyna I created during my Ph.D. This dissertation is the first to fully support the vision of Dyna proposed back in 2011 by Eisner and Filardo [59]. The approach within has the potential to enable a new class of powerful declarative logic programming languages, such as Dyna. Our approach is based on term rewriting on top of a relation algebra we call **R**-exprs (short for *R*elational *expr*ession). We have figured out how to mix a number of language features previously thought incompatible. This includes: Prolog-style backward chaining, Datalog-style forward chaining, mixed-chaining [67], memoization, reactive programming, folding & speculation, constraints satisfaction, functional, weighted terms and aggregation of multiple values for a single term (chapter §2).

Developing a new approach to implement declarative logic programming is not without its challenges. As such, we have investigated ways to make **R**-exprs practical. This includes novel (and more powerful) approaches for memoization, object-oriented programming, implementation of **R**-exprs and their rewrite rules, and compilation of sequences of rewrites.

1.1 Dissertation Outline

This dissertation focuses on the applicability of term rewriting to the implementation of modern logic/weighted declarative programming languages with appli-

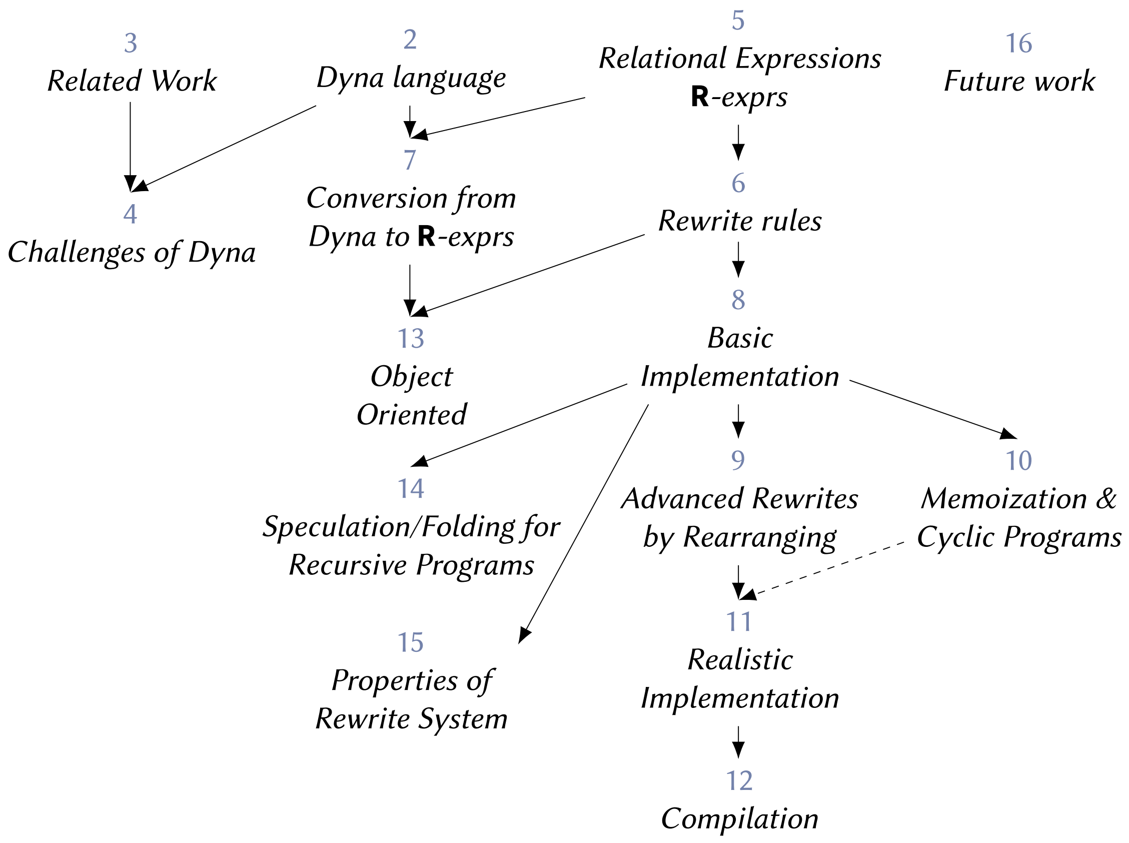


Figure 1-1. Dependency Graph Between Chapters

cations to ML/AI researchers. As such, I believe there are three different kinds of people who might choose to read this dissertation: (1) AI/ML researchers who might be interested in using the Dyna language implemented in this dissertation, (2) logic programming researchers who are interested in understanding how term rewriting can be used, and (3) term rewriting researchers who are looking for interesting applications of term rewriting.

For those interested in using Dyna, chapter §2 is written as a user manual for Dyna and should be the only chapter that one needs to read to use Dyna.

Chapters 5 to 8 introduces our relational algebra, **R**-exprs, how Dyna programs are converted into a **R**-exprs and introduces our how our rewriting process is setup. Chapters 9 to 12 build on chapter §8 to discuss some "more advanced" techniques that I have implemented on top of term rewriting to implement features such as iteration of variable domains, memoization and compilation of multiple steps of rewriting. Chapters 13 to 15 include miscellaneous discussion around Dyna and **R**-expr rewriting. The dependency graph between chapters is approximately as shown in figure 1-1, and readers should feel free to jump around between chapters.

1.2 Brief History of the Dyna Project

The Dyna project was originally started in 2003 as an umbrella project to develop a programming language for ML researchers [60]. Most algorithms that ML researchers implement can be expressed in a few lines of math.[1] In the process of researching new algorithms, researchers often have to iterate many times, refining their algorithms. This means that they first revise the mathematical concept of their algorithm and then recode their program to match. The project Dyna aims to reduce the distance between mathematical concepts and executable code.

This discrepancy between non-executable mathematical notation and executable programs was the central motivation for the Dyna project and led to the devel-

[1]This is still true of large neural networks. However, neural networks research often builds on top of a large library such as PyTorch [111] or TensorFlow [4], which contain many existing neural net modules. Conceptually, similar software libraries could be implemented in Dyna.

opment of Dyna 1.0 [60, 61]. Dyna 1.0 extended Datalog [76][2] by replacing the boolean semiring used in logic programming to allow the use of any semirings. In other words, this meant that Dyna 1.0 was a notation for *dynamic programs*. As such, Dyna 1.0 was successfully used in several research papers: Dreyer and Eisner [54], Dreyer et al. [55], Eisner et al. [62], Eisner and Smith [63], Karakos et al. [95], Schafer [117], Smith and Eisner [121, 122, 123], Smith and Smith [124], Smith and Eisner [125, 126, 127, 128], Smith et al. [129, 130].

On the heels of Dyna 1.0's success, Dyna 2.0 was proposed to rectify many of the limitations of Dyna 1.0 [59]. Dyna 1.0 requires all rules to use the same semiring[3]. Dyna 2.0 removes this restriction. Instead Dyna 2.0's rules define general functions. Dyna 1.0 is a dialect of Datalog, and as such, requires all terms derived using forward chaining to only contain *ground* terms[4]. This allowed the Dyna 1.0 compiler to generate programs that loop over the entire domain of an expression—much like a scan of a database table. Dyna 2.0 has no such restriction. Instead,

[2]Datalog will be explained further in section §3.1.2. In short, Datalog can be thought of as a programming language that operates over boolean values as a dynamic program. All statements in a Datalog program are materialized in *database* tables, which are queried (and joined) with a Prolog-like syntax.

[3] A semiring is an algebraic structure that has an additive and multiplicative operator, additive and multiplicative identity elements. Semirings do not have an additive inverse, like rings. For example, logic programming operates over the boolean semiring, $\langle \text{or}, \text{and}, false, true \rangle$. Dyna 1.0 semirings could include the real numbers $\langle +, *, 0, 1 \rangle$, or the min value (shortest path) $\langle \min, +, \infty, 0 \rangle$, but all rules had to use the same semiring.

[4]Ground terms means that all of the values in the expression are known. For example, the term `foo(1,2,"hello")` is ground, as both integers 1 and 2 and the string "hello" are ground.

Dyna 2.0 allows for variables in the program to remain *free*[5] as in Prolog. Dyna 2.0 performs unification similar to Prolog where expressions like `a(X)=a(Y)` unifies `X` and `Y` together without knowing the value of `X` or `Y`. Dyna 2.0 also supports *lazy* expression allowing for expressions like `X+Y=Z` to remain "*unevaluated*". Dyna 2.0 can also *eagerly* compute and memoize any expression to avoid recomputing the same expression many times. Dyna 2.0 also introduced a prototype-based inheritance mechanism (dynabases), which is useful for building larger programs.

It was shown in Eisner and Filardo [59] that the proposed features of Dyna 2.0 allowed for many ML algorithms to be concisely expressed. However, the unforeseen consequence of Dyna 2.0's design was that existing mainstream techniques for logic programming no longer worked. This spurred the current research phase to develop the necessary theory and techniques to implement Dyna 2.0: Filardo [66], Filardo and Eisner [67, 68], Francis-Landau et al. [70], Vieira [141], Vieira et al. [142]. Our group's research on Dyna also includes two Ph.D. dissertations prior to this one. The first, by Nathaniel W. Filardo, constructed a theoretical foundation for understanding the well-foundedness of Dyna 2.0 programs [66]. The second dissertation, by Tim Vieira, investigated automated program transformations, which can improve the asymptotic runtime of Dyna 1.0 dynamic programs [141].

This dissertation focuses on the implementation of the Dyna 2.0 language. I ac-

[5] A term with a free variable would be like `bar(7,X)` where the variable `X` can represent *any* value.

complish this by developing a novel representation of Dyna programs using **R**-exprs, a relational algebra, and develop an execution strategy using term rewriting. Much of the work presented in this dissertation is *brand new* work on Dyna, while other work in this dissertation will adapt the techniques that were previously developed by Eisner, Filardo, and Vieira to the **R**-expr presentation in this dissertation. Namely, chapter §10 will introduce how memoization works with **R**-exprs, which builds on Filardo's dissertation work, and chapter §14 will discuss how the kinds of program transformations, that are central to Vieira's dissertation work, can be implemented using **R**-exprs.

From this point forward, the name "**Dyna**" will only refer to the Dyna 2.0 version of Dyna defined in chapter §2. The version of Dyna presented in chapter §2 contains some additional features and slight modifications since the original Dyna 2.0 paper of 2011 [59].

Chapter 2

The Dyna Programming Language

In this chapter, I will detail the surface-level syntactic features of Dyna as implemented in this dissertation.[6] This chapter is intended as a *user guide* for a developer who is interested in writing Dyna programs. If you (the reader) are only interested in learning how to use Dyna, and do not care about how Dyna works internally, then this chapter should be the only chapter that you need to read.

The syntax for Dyna was originally proposed in Eisner and Filardo [59] and builds on the earlier Dyna papers (section §1.2). I continue to build on this design and extend it in this dissertation. Dyna started with the syntax of logic programming and deviates in a few ways in an attempt to modernize logic programming and make it more palatable for Machine Learning (ML) and Artificial Intelligence (AI)

[6]The Dyna project has had many implementations and different dialects over the years. Details of these alternate implementations can be found on the project's website `http://dyna.org`.

researchers. Primarily, this means having the ability to perform *weighted* reasoning. For example, a weight can be used to assign a probability to each expression in the program (section §2.2). Furthermore, most ML and AI researchers are primarily writing programs using Python. As such, my Dyna implementation provides an API to be able to easily interact with Python (section §2.3.1).

Our hope for the Dyna project is to enable new approaches to ML and AI to be easily explored, as well as existing approaches to be easily implemented. By this, I mean that we have attempted to incorporate support for commonly used programming techniques as well as techniques that are rarely included in programming languages due to their inherent complexity. This includes: invariance to expression ordering—much like a constraint engine—(section §2.4), fixed-point computation and memoization (sections § 2.5 and 2.7)—which is good for handling cyclic reasoning in graphs—, non-ground reasoning (section §2.6), support for domain-specific languages (section §2.10).

As a new take on logic programming, Dyna also includes a number of niceties that have become commonplace among modern[7] mainstream programming languages. This includes: higher-order functions (section §2.8.1), lambda functions/closures (section §2.8.2), prototype-based inheritance/objects (dynabases) (section §2.9), dependent types (section §2.8.3), and builtin commonly used data structures (such as a hash maps) (section §2.1.1.1).

[7]At least more modern than when Logic Programming and Prolog first appeared in 1972 [38].

Admittedly, all of these features have appeared before in other programming languages. What makes Dyna unique and interesting is that it is the first programming language (to our knowledge) that combines *all* of these features at the same time.

Furthermore, one of the core philosophies of the Dyna research project has been that any program which is "*syntactically valid*" and "*logically consistent*" should generally "*just work*". This means that Dyna tries extremely hard to make programs work even when other logic programming languages would result in an error or non-termination with the same program (chapter §4).

I believe that the combination of features included in Dyna is a sufficiently interesting challenge and demonstrates the main academic contributions of this dissertation (which starts in chapter §5). Additionally, I will discuss in chapter §4 some of the challenges and speculate on the reasons why I believe that Dyna is the only programming language to combine all of these features together at the same time.

2.1 Dyna's Roots in Logic Programming

Dyna builds on the foundation of Prolog [35, 38] and Datalog [30, 76, 82]—with any Datalog or Prolog program also being a semantically valid[8] Dyna program. Dyna

[8]Modulo syntactic differences. A Prolog/Datalog program would require some changes to match Dyna's syntax exactly. The internal representation of Dyna, described in chapter §5 is capable of representing Prolog/Datalog programs.

has a number of syntactic changes compared to Prolog/Datalog in an attempt to modernize logic programming and make it more palatable to developers familiar with scripting languages (such as Python, commonly used in ML applications).

Dyna includes Prolog's and Datalog's notation, which makes it easy to express constraints, don't cares, unions, selections, and joins. For example, we can define a user-defined term 'a' as the join of two relations 'b' and 'c'.

```
a(I,K) :- b(I,J), c(J,K).
```

A **logic program** is a *programming representation* of a set or bag[9] that contains expressions that return *true* according to the program. For example, we might say that a program is a function $\mathcal{P}$ that maps expressions, such as `a(2,5)`, to the value *true* or *false* depending on whether `a(2,5)` is contained in the program. For example, we can say that the program defines the term `a(2,5)` if $\mathcal{P}(\texttt{a(2,5)})$ returns true. Similarly, it is sometimes useful to think of the program as a set/bag containing all terms the program defines as true. In this case, we can write $\texttt{a(2,5)} \in \{t : \mathcal{P}(t)\}$, where the set $\{t : \mathcal{P}(t)\}$ is defined using set-builder notation to contain all terms t where the predicate $\mathcal{P}(t)$ returns true. For convenience, I will blur the distinction between function $\mathcal{P}(\cdot)$ and the set it defines and instead will write $\texttt{a(2,5)} \in \mathcal{P}$.

When working with a logic program, we are not only interested in checking whether a term such as `a(2,5)` is defined according to the program. Instead, we are often interested in asking the program to *fill in* a templated expression that contains

[9]A bag can be thought of as a generalization of a set where an element can appear multiple times. A bag will be defined properly in chapter §5.

variables with *all* assignments to the variables such that the expression returns true under the function $\mathcal{P}$. For example, if we *query* the program with "a(X,Y) $\in \mathcal{P}$", the logic programming system will return a bag of all possible assignments to the variables X and Y. E.g. $\{\langle \texttt{X} = 2, \texttt{Y} = 5\rangle, \langle \texttt{X} = 7, \texttt{Y} = 12\rangle, \ldots\}$

To see how this actually works, let us work through a small Dyna program that has been adapted from commonly used logic programming examples:

```
2  % a number of facts about parent relationships
3  % this is parent(ParentName, ChildName).
4  parent("charles", "james").
5  parent("elizabeth", "james").
6  parent("james", "george").
7  parent("sophia", "george").
8  parent("sophia", "george").
9
10 % rules can combine facts as well as other rules
11 grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
12 greatgrandparent(X, Y) :- grandparent(X, Z), parent(Z, Y).
13 married(X, Y) :- parent(X, Z), parent(Y, Z), X != Y.
14 related(X,X).
15 related(X,Y) :- related(X,Z), parent(Z,Y).
16 related(X,Y) :- parent(X,Z), related(Z,Y).
```

On lines 4 to 7 we define a number of parent-child *facts*. A **fact** is a rule without any side conditions (expressions on the right-hand side of :-). A fact represents a term that is *true*, and can be "*looked up*" in the program to check whether the expression is true. It is appropriate to think of a fact in a logic program as the same as inserting a tuple into a database table.[10]

Rules, such as those on lines 11 to 16, define terms in accordance with expressions

[10]In the nomenclature of a SQL database, the declaration of a fact is equivalent to inserting a tuple into the database. E.g.: `INSERT INTO parent VALUES ('charles', 'james');`

on the right-hand side of the aggregator `:-` which return *true*. For example, the `grandparent` rule on line 11 combines the `parent` facts to define terms such as `grandparent("charles", "george")`. A rule can be seen as similar to performing a join in a database between relations.[11] The expressions which are used to define rules can reference the all user-defined terms, include all facts, other rule defined terms, and even its own term as shown with `related(·,·)` on line 14 depending on itself.

To interact with this program, a user of Dyna will make queries against the program using the Dyna runtime, much like an SQL database. A query ends with a question mark '?' and will return the set of bindings to all variables in the expression. For example, using the program on lines 4 to 16, a user could make the following queries:

```
17 grandparent("charles", X)  ?    % make a query against the Dyna  program
18   ⦃⟨X="george"⟩⦄     % the result is returned as assignments to variables
19 parent("sophia", X)  ?
20   ⦃⟨X="george"⟩⦄                                  % duplicates are combined
```

Note: On line 20 that the duplicate facts from line 7 and 8 is only returned to the user returned once, rather than twice. This differs from what Prolog would do, and the reason for this will be explained in section §2.2.

[11]In SQL, a rule definition would be equivalent to defining a view that will be computed on demand from the defined facts. E.g.: `CREATE VIEW grandparent AS SELECT a.parent, b.child FROM parent a INNER JOIN parent b ON a.child = b.parent;`

2.1.1 Structured Terms

While logic programming using only primitive types such as string and int in the previous section is sufficient to construct simple databases, where logic programming really shines is when constructing more complicated structured terms. In Dyna, we denote these structured terms using square brackets `[]`[12] as: `termName[Argument1, Argument2, ..., ArgumentN]`. Note that this syntax for structured terms is different from what has been used for years in Prolog and Datalog. I will discuss why we have chosen to make these changes in section §2.2.1.

To see how structured terms can be used to revise our previous program, consider the following example:

```
21 % facts represented as structured terms
22 people(person["james", 50, "charles", "elizabeth"]).
23 people(person["george", 10, "james", "sophia"]).
24
25 % extract information from the structured term
26 name(person[Name, Age, Father, Mother], Name).
27 parent(person[Name, Age, Father, Mother], Father).
28 parent(person[Name, Age, Father, Mother], Mother).
29
30 % return the names of all known people
31 names(Name) :- people(Person), name(Person, Name).
```

On lines 22 and 23, we define two people using structured terms. We can get the name of a person using a rule like line 26, which will select the first argument that contains the name of the person. We can further make queries against this

[12]Dyna also allows for an ampersand (&) to be used as a "quote" on a term as an alternate way of writing this expression. E.g. `&termName(Argument1,Argument2, ...,ArgumentN)` $\equiv$ `termName[Argument1,Argument2, ...,ArgumentN]`.

program as follows:

```
32 people(Person), name(Person, "james"), parent(Person, Parent) ?
33   Results returned:
34    ≀⟨Person=person["james",10,"charles","elizabeth"],Parent="charles"⟩,
35     ⟨Person=person["james",10,"charles","elizabeth"],Parent="elizabeth"⟩∫
```

On line 32, we start with all of the people defined on lines 22 and 23, and then we then filter the people, selecting only those with the name `"james"`, using `name(Person, "james")`. Finally, the `parent` rule is used to extract both the father and mother fields from the structured term.

2.1.1.1 Builtin Structured Terms

Dyna additionally includes built-in structured terms for convenience. Following Prolog's design, Dyna includes a linked list that can be constructed using square brackets (`[]`) when there is no term name (e.g. line 36). A vertical bar is used to create a new list where one or more elements have been added to the head of the linked list (e.g. lines 37 and 38). Recursive functions such as `list_length` (line 41) can be used to scan through a linked list by accessing the first elements of the linked list.

```
36 list = [1,2,3].  % Construct a list
37 list_prepend = [4 | list].  % The vertical bar is used to prepend or
38 list_remove = Rest for list = [Head|Rest]. % return the head and tail
39
40 % A recursive predicate defines ⟨list, length⟩ terms
41 list_length([], 0).
42 list_length([Head|Rest], N+1) :- list_length(Rest, N).
```

The equals sign on line 36 shows a major syntactic change from logic programming in that terms have values. Line 36 defines the term `list` as having the value `[1,2,3]`. The body of a rule is defined by an expression that returns a value. For example, on line 38, the expression '`Rest for list = [Head|Rest]`' returns the value assigned to the variable `Rest` when the expression '`list = [Head|Rest]`' returns true.[13]

Furthermore, inspired by modern scripting languages that make extensive use of associative maps, I have extended Dyna to also include support and syntax for a built-in dictionary type[14] (line 43). A dictionary can be constructed using curly braces ({}). Similarly to linked lists, a vertical bar (line 52) is used to both access a key in the dictionary (line 57) and add/remove keys from the dictionary (line 50).

```
43 person_dictionary_map = {   % a dictionary
44   "Name" -> "james",
45   "Age" -> 50,
46   "Father" -> "charles",
47   "Mother" -> "elizabeth",
48 }.
49
50 person_with_address = {                % new dictionary with keys added
51   "Address" -> "123 North st.",
52   | person_dictionary_map     % the vertical bar is used to create a new
53               % dictionary with new elements added or existing elements
54               %   removed, just like the linked list on lines 37 and 38
55 }.
56
57 name({Name | Rest}, Name).         % The variable name can be used to...
58 name_alt({ "Name" -> Name | Rest}, Name).         % ...match a key in the
       dictionary
```

[13]The '**for**' keyword will return the *first* sub-expression's value with the second sub-expression representing a side condition that **must** return the value *true*. The comma '`,`' is conceptually equivalent to '**for**', but it returns the *second* sub-expression's value and the first sub-expression **must** return true. In other words, the expression '`A for B`' is equivalent to the expression '`B, A`'.

[14]Future work may consider adding in syntax for other types, such as sets. However, sets can be emulated using a map by using only the Key and setting a dummy value for all pairs, like: `{"X"->0, "Y"->0}`.

```
59 get_item_by_key({Key -> Value | Rest}, Key, Value).
```

The '`Key -> Value`'[15] pair in the dictionary can use any values in the Herbrand universe. When adding a key to a dictionary, as on line 50 the key *must not* be contained in the dictionary, otherwise the result of this expression will be `null`. The reason is that when pulling a key out of the dictionary, as on line 57, will result in that key not being contained in the dictionary represented by `Rest`, and we require that all uses of an operator, such as the vertical bar, behave *identically* regardless of how it is used (in this case to add or remove a key).

2.2 Weighted Rules

As a Dyna extension to traditional logic programming, Dyna associates a value with each term in the program. As such, we call rules *Horn equations* rather than *Horn clauses* as in Prolog and Datalog. A **Horn equation** returns the value computed from the expression in the equation (which appears on the left-hand side of the aggregator). Returning a value from terms moves Dyna from a logic programming (which only returns *true* or `null` (undefined/false)) towards a functional programming language. A single term in Dyna can have values contributed either from a single rule or from a collection of different rules. To handle this, we *combine* the contributed values using an aggregator. An **aggregator** is an associative and commutative binary operator written between the head of the rule on the left-hand

[15]The notation '`->` ' was chosen to avoid conflicting with other syntax in the Dyna language.

side and the body of the rule on the right-hand side. For example, the aggregator '`+=`' sum of all contributions, '`max=`' selects the maximum contribution, and the aggregator '`:-`' that we have already seen checks that there is some expression that returns *true*.

Using the '`+=`' aggregator, we can write a matrix multiplication between two functors '`b`' and '`c`', which represent matrices as follows:

```
a(I,K) += b(I,J) * c(J,K).
```

Here, if we have the term `b(2,8) = 3.` and `c(8,5) = 7.`, then this will define `a(2,5) = 21`. One major difference between Dyna and other logic programming languages is that aggregators such as '`+=`' combine *all* possible values—whereas boolean *logical* aggregators such as '`:-`'[16] can stop once it has found a proof that a term is true. To illustrate this point, let us consider what happens if we add rules `b(2,9) = 11.` and `c(9,5) = 4` to our program. In this case, we will have that `a(2,5) = 65.` which is the sum of 21 and 44. Conversely, if this was a logic program as in section §2.1, then the value of `a(2,5)` already has been proven true by `b(2,8)` and `c(8,5)`, and defining new values for the terms `b(2,9)` and `c(9,5)` does not change the *truthfulness* of `a(2,5)`.

In addition to the sum aggregator, Dyna also includes other aggregators, which are commonly found in AI and ML applications. This includes '`min=`' and '`max=`' which find the min or max value respectively. '`*=`' computes the product of the

[16]Prolog and Datalog are "allowed" to stop early, but the exact circumstances under which this happens can differ. See the discussion in the related work chapter for more information. §3

contributions much like '`+=`' computes the sum. '`|=`' and '`&=`' compute the boolean logical OR and logical AND between contributions. Dyna also includes a special aggregator '`:=`', which selects the value from the rule defined on the "last line" that returns a non-`null` value. This allows us to define rules that *override* the value returned by previously defined rules for a term.[17] Finally, we have the aggregator '`=`' which ensures there *only one* contributed value (otherwise it will error), and the aggregator '`?=`' which is allowed to arbitrarily pick *any* of the contributed values.[18] [19]

To see how aggregation in logic programming can make expressing AI programs easy, let us consider the task of computing the shortest path in a graph. On lines 62 and 63, we define a recursive rule to compute the path in a graph. The rule will look for the *minimum* value associated with each node. We use the keyword **`arg`**[20] to track the argument associated with the minimum value at each node.[21] This, in

[17]The '`:=`' aggregator accepts a special value of `$null` that can be returned by an expression to make the result of '`:=`' `null`, overriding other non-`null` contributions from earlier lines.

[18]Dyna is allowed to stop computation for a term's value early once it has determined that the value will not change further. This happens in the case of `|=, &=, :-` which have saturating values of true, false, true. Aggregators like `:=` can stop the computation of earlier lines in the program if there is a value contributed from a line that occurs later. Similarly, `?=` can stop all other computations once it has found something.

[19]Each *term* in the program can only use a single aggregator otherwise the value is defined to be an error. However, rules that define terms that share the same functor name can be defined using different aggregators. For example, we can write '`a(X) += 2 for g(X).`' and '`a(X) *= 7 for q(X).`' provided that `g(X)` and `q(X)` are non-overlapping.

[20]Previous Dyna papers have used the keyword **`with_key`** for this instead of the keyword **`arg`**. The keyword **`with_key`** is still supported by the Dyna front-end parser.

[21]Internally, Dyna tracks this using a pair of values, which includes the argument and the value returned. The Dyna implementation automatically references the value field by default or will reference the argument field using the macro **`$arg`**`(·)` which suppresses the automatic access of the value field.

essence, tracks *backpointers* at every node in the graph with **$arg**(path(·)) being equivalent to a mathematical expression using argmin.

```
% Horn equation defines how to compute path and tracks back edges
path(start) min= 0                       arg [start].
path(X)     min= edge(X,Y) + path(Y)     arg [X | $arg(path(Y))].

% weighted facts define the edges in a graph
edge("bal", "nyc") = 200.
edge("dc" , "bal") = 20.
edge("dc" , "nyc") = 300.
start = "nyc".

% assert[22] checks the expression is true conditioned on the lines above
assert path("dc") = 220.
assert $arg(path("dc")) = ["dc", "bal", "nyc"].
```

2.2.1 Evaluation by Default

One of the major syntactic differences between Dyna and other logic programming languages is that Dyna *evaluates* an expression in place by default. The reason for this change is that most terms *have*[23] a meaningful value, much like how a function returns a value in a functional programming language. Conversely, in logic programming languages such as Prolog or Datalog, terms only "return" the value of *true*.

[22]A Dyna *program* is declaritive in that defined *rules* can be rearranged and evaluated out of order. A Dyna *script* may contain procedural statements such as **assert** and **print**, which evaluate an expression with respect to the rules defined *before* the statement.

[23]There is a slight technicality here between "having" a value and "returning" a value. In Dyna, the *term* is an identifier, and the *Dyna program* is the function that takes the *term* as an argument and *returns* a value. In other words, in a functional language, you might have "`return_value = named_function(Arg1, Arg2)`" whereas in Dyna we have "`return_value = dyna_program_function(named_term_identifier[Arg1, Arg2])`" with the program itself being the function.

To see how this manifests, let us consider the program that computes $\sin(x)^2 + \cos(x)^2$ written in Dyna vs Prolog:

```
% Dyna rule for computing the trig function
trig(X) = sin(X)**2 + cos(X)**2.

% Prolog rule for computing the trig function
trig(X,Result) :- sin(X,S), pow(S,2,SS), cos(X,C), pow(C,2,CC),
                  Result $= SS + CC.[24]
```

In Prolog and other *logic* programming languages, whenever we want to get the *value* from a term, we have to represent the returned value using an extra argument such as `S` and `C`. The reason is that terms like `sin(X,S)` only "return" *true*. Hence, it would be meaningless to write an expression like `pow(sin(X),2)` in Prolog with evaluation by default, as it would be equivalent to `pow(true,2)`. Instead, Prolog chooses to use this syntax to represent structured terms (section §2.1.1), with the expression `pow(sin(X),2)` being interpreted as `pow(sin[X],2)` which is at least potentially meaningful as long as '`pow`' is able to do something with the structured term `sin[X]`.

[24]The $= is necessary in Prolog to represent a floating point addition constraint between the floats `SS`, `CC`, and the `Result` variable. If this was written with an equals sign only and without the dollar sign, it would end up assigning the term `'+'[SS,CC]` (not the numerical value) to the variable `Result`.
Alternately, Prolog has **`is`** which can be used to evaluate numerical expressions and could be used as `Result **is** SS + CC`, but **`is`** will only evaluate the expression on the right-hand side and assign it to the variable on the left-hand side. In other words, in Prolog the following **`is`** expression would result in an error: `5 **is** 2 + X`.

2.3 A User's Interaction With Dyna

Dyna's intended audience is ML and AI researchers. This means that Dyna is more focused on expressing mathematical models *quickly* and getting them to work. Dyna also encourages *interactive experimentation*, which is useful when debugging a ML model. Dyna also hopes to be a tool which *complements* existing tooling and infrastructure used by ML researchers—it is unreasonable to expect Dyna to have a thriving ecosystem on its own. As such, Dyna is designed to be used from a "*driver*" program written in another language.[25] In this way, Dyna is similar to a database where a program submits *queries* and *updates* to run against the Dyna program. As such, Dyna provides a Read-Eval-Print-Loop (REPL) for interacting with the Dyna system, as well as a Python and Java API.

2.3.1 Python API

Most ML and AI research is currently being done using Python. As such, we expect that the Python API will be the way most users will interact with Dyna. The Java API also provides the same interface as the Python API, so we will omit detailing the Java API here.

Interacting with the Dyna system is similar to interacting with an in-memory database (such as a SQL database). A new Dyna runtime instance is created using

[25]Prior work did include the idea that there would be a driver program querying and updating the Dyna program [61]. However, the design of the interface detailed here is a contribution of this dissertation.

the dyna.Dyna() method (line 82). Programs can be loaded into the Dyna runtime using the run method (lines 84 and 92), passing either a string of code or a file containing Dyna code. The defined rules and facts will persist between calls to the run function, just like a database.

```
80  from dyna import Dyna
81
82  dyna_runtime = Dyna()        # create a new instance of the Dyna runtime
83
84  dyna_runtime.run("""
85  factorial(N) := factorial(N-1) * N.
86  factorial(0) := 1.
87
88  print factorial(5).  % prints 120
89  """)
90
91  # load rules from a file
92  dyna_runtime.run(open('dyna_program.dyna'))
```

The query method (line 93) is used to return the values that are calculated by the Dyna program. We can return primitive types, such as numbers and strings, as well as more complicated types, such as lists, dictionaries, and dynabases (objects inside of Dyna, which will discussed further in chapter §13):

```
93  result = dyna_runtime.query("""
94  factorial(10)?       % The first query
95  factorial(11)?       % The second query
96
97  factorial_up_to(N) := [factorial(N) | factorial_up_to(N-1)].
98  factorial_up_to(0) := [].
99
100 factorial_up_to(5)?  % Third query returns a list
101 """)
102
103 assert result[0] == 3628800        # the results of queries are returned
104 assert result[1] == 39916800
```

```
assert result[2] == [120, 24, 6, 2, 1]
```

Like SQL database APIs, Dyna supports *query parameters* for passing values into Dyna without having to encode the value as a string. The query parameters are denoted using a dollar sign followed by a number: $0, $1, $2 . . . $n. We can pass any value into the Dyna runtime. When a value can be cast into Dyna's Herbrand universe then it is automatically cast and usable inside of Dyna. If a value cannot be cast to Dyna, then it is passed around as an opaque pointer. This allows Dyna to integrate with other Python libraries without having to support everything itself.

```
result = dyna_runtime.query("""
factorial($0)?
""", 5)

assert result[0] == 120

class MyClass: pass

dyna_runtime.run("""
class_reference = $0.  % save a reference to the class
""", MyClass()).

result = dyna_runtime.query("""
class_reference?
""")

assert isinstance(result[0], MyClass)
```

Dyna's Python API also supports defining external functions that can be called from the Dyna program. This allows Dyna to leverage existing functions and libraries without requiring all features to be fully reimplemented in Dyna. Unlike with Dyna terms, *invoking* an externally defined function requires that all arguments have known values.[26]

```
@dyna_runtime.define_function('my_function')
def my_function(a,b,c):
    return a*3 + b*5 + c*11

dyna_runtime.run("""
assert my_function(1,2,3) == 1*3 + 2*5 + 3*11.
""")
```

As a more useful example of Dyna interacting with external datatypes, we can leverage PyTorch's [111] GPU tensors. Operations such as tensor multiplication can be performed by calling back to external functions:

```
import torch

gpu_tensor = torch.tensor(...).cuda()

@dyna_runtime.define_function('tensor_multiply')
def tensor_multiply(A, B):
    return A @ B

dyna_runtime.run("""
pytorch_tensor = $0.

tensor_squared = tensor_multiply(pytorch_tensor, pytorch_tensor).
""", gpu_tensor)
```

Currently, Dyna does not have built-in support for GPUs, so this approach can at least serve as a "stop gap" for working with GPUs, which are critical for making modern (neural) ML models efficient. In section §16.7 I will discuss potential future work to add native GPU support to Dyna.

[26]Externally defined functions must have unique names and cannot be combined with terms that are defined by the Dyna program.

2.3.2 Multi-file Programs

Just like other programming languages, Dyna has the ability to write a program using multiple files and import the definitions across files.[27]

Terms in Dyna are scoped to the file in which they were defined. This means that a term `foo(X)` defined in file `a.dyna`, will be different from `foo(X)` in file `b.dyna`. Terms can be imported from other files using `import` at the top of the file as follows:

```
:- from "a.dyna" import foo/1.
```

Dyna also supports importing default terms that were declared as exported. This way, they do not have to be explicitly listed when importing terms into another file.

```
% in file a.dyna
foo(X) = 123.
:- export foo/1.

% in file do_import.dyna
:- import "a.dyna".
assert foo(1) = 123.  % use foo defined in file a.dyna
```

2.4 Invariance to Expression Order

One of the core principles of Dyna is that the way that a program is written should have as *little* impact as possible on how the program executes. The Dyna runtime is required to find *some* solution to the equations/constraints written, but otherwise, it is flexible in how it executes a program. We hope that this approach will make it easier for developers to write Dyna programs as they can focus on the "business

[27] This is a contribution of this dissertation.

logic"[28] of their ML/AI model while not having to worry about the internal state of the Dyna program.

One way in which Dyna manifests this property is that Dyna is invariant to the order in which expressions and most rules are defined in the program. In this way, Dyna is closer to a *constraint satisfaction engine* than other logic programming languages, which have a fixed execution order (section §3.1.1). For example, a well-studied example Prolog program is computing the permutations of a list (lines 151 to 154). Following the mathematical definition of a permutation, given a list l and the permutation p of list l, will have that l is also a permutation of the list p. E.g. the list [1,3,2] is a permutation of [1,2,3] *and vice-versa.*

```
151 deleteone([X|Xs], Xs, X).
152 deleteone([X|Xs], [X|Ys], Z) :- deleteone(Xs, Ys, Z).
153 permute([], []).
154 permute(As, [Z|Bs]) :- deleteone(As, Rs, Z), permute(Rs, Bs).
155
156 permute([1,2,3], Out)?              % works under backchaining (Prolog)
157 permute(Out, [1,2,3])?              %  does not work under backchaining
```

This program, when executed under Prolog's fixed execution order, only line 156 will work, while line 157 will cause Prolog to get stuck in an infinite recursion. The reason is that Prolog executes in a fixed *left-to-right*, *top-to-bottom* rule ordering. As a result of this fixed order, when Prolog executes line 154, the call to deleteone will be evaluated before the recursive call to permute is evaluated. This means that the variables As and Rs will both be *un-ground*. This causes deleteone to enumerate

[28]Business logic sometimes called domain-specific logic is the part of the program which encodes rules which are relevant to the real world problem being solved.

the *infinite* pairs of all lists where the list `Rs` will contain one less element that is equal to `Z`.

Conversely, when this program is executed under Dyna, we do not commit to execute the rules in any particular order. So, while Dyna runtime is allowed to *explore* the `deleteone` function before `As` and `Rs` are grounded to specific values, it does not commit to running the `deleteone` function before the `permute` function.

2.5 Fixed-Point Computations

Dyna is designed as a superset of Prolog and Datalog styles of logic programming. Datalog's execution strategy is notably different from Prolog's. A Prolog program is executed using *backward chaining*. This means that Prolog starts with the expression that it is trying to prove to be true and performs a *depth-first* search "backward" looking for other rules and facts that support the expression it is trying to prove. In contrast, Datalog operates via *forward chaining*. When Datalog starts, it looks at all of the ground facts that are currently asserted by the program. Datalog will then *deduce* new facts using the rules in the program.

```
158 a :- b.   % a is true if b is true
159 b :- a.   % b is true if a is true
160 c :- d.   % c is true if d is true
161 a.        % assert that a is true regardless
```

On line 161, we define the fact that the term a is true. Datalog using forward-chaining will use line 159 to deduce that b is also true. Datalog will then stop processing as there are no new rules which can be deduced as true.[29] In this way, Datalog has reached a *fixed-point* of the program where: $\mathcal{P}_T = \textsc{DeduceAllTrue}(\mathcal{P}_T)$. Here, we can think of the function DeduceAllTrue as taking the set of all terms in the program $\mathcal{P}_t$ at time t, and computing a new set of terms $\mathcal{P}_{t+1}$ at time $t+1$ which includes the new terms which have also been deduced. Once no additional terms are proven as true, the fixed-point condition is satisfied $\mathcal{P}_T = \textsc{DeduceAllTrue}(\mathcal{P}_T)$.

As mentioned above, Dyna is a superset of Datalog's fixed-point computation. As such, Dyna also includes support for fixed-pointing weighted expressions (whereas Datalog can only fix-point boolean values that are proven true). For example, we can use a fixed point to solve the expression $e = 1 + e/2$ by writing lines 162 to 163.

```
162 e += 1.
163 e += e/2.
```

Dyna will solve this program by first assigning the value of 1 to the term 1, and then it will iterate line line 163 until it reaches numerical convergence.

The place where having fixed-point semantics is important is in the case of *cyclic programs*, such as when representing a cycle on a graph. For example, when computing the shortest path in a graph in section §2.2, we were only interested in finding the minimum length to a given node in the graph. If we had a large set of edges, it is likely it would have contained cycles and many alternate paths. Furthermore, the fewest number of steps might not have been the cheapest path in

[29]Note: the program on lines 158 to 161 will not execute under Prolog, as Prolog will get stuck performing a depth-first search on lines 158 and 159.

the graph. Hence, the value assigned to a node in the graph might change multiple times during the program's run.

Note: A fixed-point might not exist for a program, or there might exist multiple different fixed-points for a single program. For example, if we define that a term is true if it is false, then the Dyna runtime can cycle forever, proving that ne is true, and then switching back to ne is false:

```
164 ne :- not ne.  % ne is true if ne is false
```

Datalog is able to reject programs like this due to stratification [30, 76], which limits the kinds of programs supported by Datalog. Dyna does not limit programs with stratification. Furthermore, Dyna programs are Turning complete (section §15.2) so proving that a general program terminates or cycles is impossible.[30] Furthermore, we can write a Dyna program which *never* repeats a state by writing a non-terminating program that counts to infinity (lines 165 and 166), or has a negative-weighted-cycle in the shortest path graph (e.g example line 62).

```
165 count_to_infinity += 1.
166 count_to_infinity += count_to_infinity.
```

As for programs that have multiple fixed-points, Dyna is only looking for an assignment to all terms in the program that are consistent with all of the rules that are defined.[31] This means that it is possible to define a term that will take on

[30]This is the halting problem, which states that there exists a program for which we can not prove or disprove that it terminates. This property is true for all Turing complete languages.

[31]When there are multiple solutions, Dyna is only required to find *a* solution. We make no guarantees about *which* solution is returned. This is in contrast to Datalog, which guarantees that the *minimal* solution is returned (section §3.1.2), and answer set programming, which returns *all* solutions (section §3.1.6).

the value true or false, or even a term that can take on any value in the Herbrand universe.

```
true_or_false :- true_or_false.
any_value = any_value.
```

In practice, if either of these terms are queried, they will return the result `null` or undefined as there is no base case to their recursion. However, this behavior should not be depended on as what value is returned by these kinds of expressions should be treated as *purposely underspecified behavior*.

2.6 Non-ground Reasoning

As mentioned a few times already, Dyna is a superset of both Prolog and Datalog; this means that it supports features of both languages at the same time. One such feature is *non-ground* reasoning. Non-ground reasoning means that the value assigned to a variable is not known at the time of execution. This technique is most commonly associated with Prolog-style systems and is not supported at all by Datalog. An example of non-ground reasoning would be an expression like:

```
a :- X > 5, X < 3.
```

Here, the value of a will be `null`, as there is no assignment to the variable `X`, which is both greater than 5 and less than 3 at the same time. This reasoning can also be applied across rules boundaries. For example, the following will still deduce that the term a is false.

```
b(foo[Z]) :- Z > 5.
```

```
171 a :- b(foo[Y]), Y < 3.
```

What makes Dyna's approach to non-ground reasoning more powerful than Prolog is that it is able to run non-ground reasoning while simultaneously performing Datalog-style fixed-pointing. To see an example of what this means, let us modify our shortest path program from line 62 to remove the hard-coded starting node and turn it into the *all-pairs* shortest path program:

```
172 path(Start,X) min= edge(X,Y) + path(Start,Y).
173 path(Start,Start) min= 0.[32]
174
175 edge("bal", "nyc") = 200.
176 edge("nyc", "bal") = 180.
177 edge("dc" , "bal") = 20.
178 edge("dc" , "nyc") = 300.
```

Observe this program cannot be executed under Prolog-style backward-chaining or Datalog-style forward-chaining. The reason that Prolog fails to execute this program is that there exists a cycle in the edge graph (`"bal"`→`"nyc"`→`"bal"`); hence depth-first backwards-chaining would get stuck in a cycle. While Datalog can handle the cycles in the program, it cannot support line 173, as Datalog does not handle non-ground expressions like this.

[32]This example, and specifically line 173, is discussed further in section §4.2.1.

2.7 Memoization, Dynamic Programming, and Reactive Programming

One central feature in Dyna is support for *memoization*. **Memoization** essentially is the act of replacing a compute operation in a program with a "recall from memory" operation. For certain kinds of programs, this can have a significant impact on the runtime of a program. For example, the Fibonacci sequence (line 179), will take exponential time without memoization but can run in linear time when memoization is enabled:

```
179 fib(0) += 0.
180 fib(1) += 1.
181 fib(N) += fib(N-1) for N > 1 ;     % semicolon defines two rules at once
182          fib(N-2) for N > 1. % when the head and aggregator are the same
183 
184 print fib(10).  % run in 177 operations without memoization
185 
186 $memo(fib[N:$ground]) = "unk".  % enable memoization for fib where N[33]
187                                 % must be a ground value (such as 0,1,2 etc)
188 print fib(10).  % run in 10 operations with memoization
```

Memoization traditionally requires that the memoized result of a function does not change. However, Dyna's memoization implementation is *reactive*. **Reactive** means that Dyna automatically tracks dependencies between anything that a memoized value depends on, and that the memoized value will be recomputed or updated if anything upstream is changed. The value assigned to a term by the Dyna program can change as a result of the system *converging* towards final values

[33]The $memo control mechanism for memoization is a contribution of this dissertation.

or as the result of the program being modified through the addition of new rules or facts (e.g. line 97 in the Python API example, section §2.3.1).

This means that if some input to a currently memoized value changes, then any dependent memoized values will also be updated. For example, if we modify the definition of `fib(3)` by defining a new rule, this will cause all of the values of the Fibonacci terms to change.

```
print fib(10).  % print 55 using the existing memoized value
fib(3) += 1.    % modify the definition of fib
print fib(10).  % print 76 using the new modified version of fib
```

2.7.1 Prioritization of Updates

In a Dyna program, there are often many terms in the program that are memoized. As a result, there can be many different terms that have pending *updates* waiting to be processed at any given point in time. However, the Dyna runtime cannot process all updates simultaneously. Additionally, the order in which these updates are processed can significantly impact the program's overall runtime. To handle these cases, we allow the user to specify a prioritization function that controls the order in which updates can be processed. The definition of the prioritization function does not impact the correctness of the program; however, a badly defined prioritization function can cause the program to exhibit much worse runtime—for example, a badly defined prioritization function will cause the Fibonacci example (line 179) to run in exponential time instead of linear.

A priority function is defined by defining a `$priority` term just like the `$memo` show above.

```
$priority(fib[N]) = -N.      % prioritize small values first (0,1,2 etc)
                            % this is the optimal prioritization function
                                          % for fibonacci and runs in O(n)
$priority(fib[N]) = N.      % prioritize large values first (10,9,8 etc)
```

The priority function will map a term to a number that is used as the priority. Higher priorities will be run first.[34] The priority function is computed only when the pending update is created.[35] This means that `$priority` should be considered a meta-user-definable term used only to control the inner operations of the system and is not like the other Dyna terms in the language, which can be updated reactively.

2.7.2 Memoization with non-ground variables

Memoization in a logic programming language, such as Dyna, is not as simple as memoizing the returned value. Instead, terms in logic programming languages support being called in different *modes*. A **mode** is the signature which a term supports being "called" with in terms of which variables *must* be known *ground* values in the Herbrand universe, and which variables are *allowed* to be *free* and have an unknown, yet to be determined, value. We have already seen an example

[34]The system internally uses the priority of 10^{16} for internal work, which should be run as soon as possible, and -10^{16} for user updates when no `$priority` is set. The system does not prevent you from setting a priority higher or lower than $\pm 10^{16}$, but be aware of these internal values as going higher or lower than $\pm 10^{16}$ can have unforeseen consequences.

[35]Future work may wish to add the ability for `$priority` to handle updates.

of *ground* and *free* variables in the matrix multiplication example repeated here.

```
196 a(I,K) += b(I,J) * c(J,K).
```

Observe that the only constraints on the variable `J` are the terms `b(I,J)` and `c(J,K)`. This means that either '`b`' or '`c`' must allow for the variable `J` to be *free*. In which case, that functor will be able to enumerate an upper bound on the domain of the possible values for the variable `J`. And by extension, any memoized version of that functor will also have to allow for the variable `J` to be *free*.

To declare a policy for memoization, we use the declarations `$free`, `$ground` and the meta-term `$memo`. When a variable is marked as `$ground` in a `$memo` policy, this means that the value assigned to the variable must be known *before* we can even *check* if there are any relevant memoized terms in the memo table. Similarly, `$free` specifies that the variable's value is allowed to be unknown and allows us to check for matching terms in the memo table. Unlike with `$ground`, when a variable is marked with `$free` the memo table will *guarantee* that all relevant terms will be memoized.

For example, we can write a memo policy for the matrix example on line 196 as follows:

```
197 $memo(b[I:$ground,J:$free]) = "null".
```

Here on line 197, we are specifying that the first variable `I` *must* be known before we can check the memo table where we will find *all* of the relevant terms for different values of `J`. The `$memo(·)` returns the *string* `"null"` to indicate what is the *default* value for the memoized `b(·,·)` terms. By stating `"null"`, we declare that any term *not* found in the memo table has the associated value `null`, and is therefore undefined. Alternately, the `$memo` function can return `"unk"`[36] to indicate that the value for a term *unknown* and must be computed and should be saved into the memo table, or it can return `"none"` to indicate that there is no value in the memo table for a term and the value should *not* be saved into the memo table.[37]

2.8 Modern PL Constructions

Dyna's syntax was primarily inspired by logic programming languages which date back to the 1970s. Since then, more modern programming languages have been adding syntactic sugar to make common operations easier to write. Dyna has followed suit and includes higher-order functions,[38] lambda functions, and type annotation.[39]

[36]The terminology of `"null"` vs. `"unk"` memos was adopted from Filardo and Eisner [67]—an earlier research paper published by our group.

[37]An `"unk"` memo is shorthand for specifying the case where all arguments *must* be ground. For example, the following policies are equivalent:
`$memo(foo[X,Y:$ground,Z:$free]) = "unk".` ≡
`$memo(foo[X:$ground,Y:$ground,Z:$ground]) = "null".` This is discussed in greater detail in section §10.8.3.

[38]Admittedly, Prolog included `call/N`, which supports higher-order functions, which have existed since early version of Prolog. However, earlier versions and proposals of Dyna were Datalog-inspired and did not include higher-order functions. Hence, the contribution of higher-order functions is specifically to *Dyna* and its syntax and not a contribution to logic programming in general.

[39]The modern PL constructions implementation and their syntax are all contributions of this dissertation work to the Dyna language.

2.8.1 Higher-Order Functions

Dyna allows for "function pointers" to be passed around the runtime and for those pointers to be used via indirect calls. This enables Dyna to support higher-order functions. A function pointer in Dyna is represented using the same structured term (section §2.1.1) that we have seen already. When an indirect call is performed, Dyna looks up the user-defined term by name and passes any additional parameters. In Dyna, user-defined terms are scoped to the file in which they were defined. This means that there can potentially be many user-defined terms that share the same name but are otherwise unrelated. To handle this, structured terms track which file they were constructed in via additional hidden metadata. This metadata is only used when an indirect call is performed. A structured term can also be scoped to a dynabase (an object), allowing indirect calls to be performed against terms defined on a dynabase object.

To make an indirect call in a higher-order function, we can simply use a variable in place of the functor's name as on line 198.[40]

```
198 perform_call(Calling) = Calling(1,2,3).
199 called(X,Y,Z) = X*Y + Z.
200 called(X,Y,Z,W) = X*Z + Y+W.
201 
202 assert perform_call(called[]) = 1*2 + 3.
203 assert perform_call(called[4]) = 4*2 + 1*3.  % curried functions can
204                                               % include extra parameters
```

[40]The indirect call can also use a *expression*, such as a call to another user-defined term. In this case, extra parenthesis are needed around the expression returning the function pointer. E.g. `(foo(1,2))(4,5)`

More usefully, this can allow us to define higher-order functions such as map, which applies a function to all elements in a list line 205.

```
205 map(Func, []) = [].
206 map(Func, [X|Rest]) = [Func(X) | map(Func, Rest)].
207
208 assert map('*'[2], [1,2,3]) = [2,4,6].
```

2.8.2 Lambda functions

Dyna includes syntax for creating lambda functions. Lambda functions are given a generated name and converted into a standard function like we have been writing already. Given that lambda functions are the same as regular Dyna rules, this means that we also define an aggregator and perform aggregation inside of the lambda function. For example, if we want to create a function that takes the max of two arguments, we can write: `((X, Y) max= X ; Y)`. The semicolon splits between different body expressions, as was done on line 181. The outer parentheses encapsulate the entire expression, with the first inner parentheses denoting additional arguments that can be passed to the lambda expression. If no arguments are included, then the lambda is immediately evaluated in place. For example, we can write `X=5, Y = 7, 7 = (max= X; Y)` where the variables `X` and `Y` are captured in a closure, and then the expression is immediately evaluated using the `max=` aggregator.

Together this can be combined with the higher-order-functions to write the following expressions, which shows how these concepts can be syntactically combined:

```
209 assert Seven=7, map(((Arg1) max= Seven; Arg1), [1,5,10]) = [7,7,10].
```

2.8.3 Type Declarations

Dyna includes support for adding type restrictions onto any expression in the language. Type annotation is denoted using a colon followed by a term. For example, the expression `X:int`, denotes that the variable `X` is of type `int`. The syntax `X:int` adds the constraint `int(X)` into the body, where `int(X)` the same kind of call to the `int` term that we have seen already. The difference with the colon expression is that instead of returning the value of the `int(X)` call, this expression returns the value of `X`. This means that we can easily annotate variables with types throughout the program (e.g. line 213) and can define our own types (e.g. line 210).

```
210 my_custom_type(foo[X,Y:float,Z:int]) :- Y > Z  % define a custom type
211 my_custom_type(bar[X:string]).                 % with side conditions
212
213 my_function(X: my_custom_type) = ....  % check the type matches
```

Furthermore, we can use this approach to construct "*templated*" types. This is accomplished by passing arguments to the term that appears after the colon. The type-constrained variable is appended as the last argument. For example, a templated list type can be defined as on line 214:

```
214 list(Type, []).
215 list(Type, [Head | Rest]) :- Type(Head), list(Type, Rest).
216
217 % Check the argument L matches a list of a particular type
218 list_of_my_type(L : list(my_custom_type[])) = ....
219 list_of_greater_than_5(L : list(((X) :- X > 5))) = ....
```

The ‘`Type(Head)`’ expression turns the `list(·,·)` term into a higher order function which indirectly calls the term referenced by the variable `Type` on line 215

Union types can also be written inline through the use of a vertical bar like `X:int|string`. Conceptually, this is the same as defining a new term that attempts to match against `int(X)` or `string(X)` using the `:-` aggregator.[41]

2.8.3.1 Type Checking and Type Errors

In Dyna, “types” are not distinct from “normal” function calls. This is not a significant issue, though, as in Dyna, the Dyna system is allowed to evaluate any expression at any point. This means that the Dyna runtime is allowed to evaluate expressions and function calls at “compile-time” if there is sufficient information to deduce the resulting value of the expression. In other words, if we can statically deduce the types, then it is possible to avoid the runtime overhead of performing type checks. However, in the cases where there is a *complicated* type, the Dyna runtime is allowed to perform runtime type checks as needed.

A consequence of this design is that Dyna does not have “type errors” that can be concisely reported to the user. Instead, Dyna can check and warn about *dead code* which will never be evaluated under any circumstances. For example, the expression `5 < 3` is always returns false. Similarly, the expression `int(X), string(X)`, which

[41]This is equivalent to defining a new `int_or_string(X)` term using two rules to create a union type:

```
int_or_string(X) :- int(X).
int_or_string(X) :- string(X).
```

places two incompatible type constraints on the variable `X`, hence this expression will always return *false* and can be reported as a warning.[42]

2.9 Object-Oriented Programming (OOP) via Dynabases

When building a larger, more complicated AI and ML system, it is necessary to have some way to make smaller units that can be composed together. In most programming languages, this is done via object-oriented programming (OOP) or via modules (section §2.3.2). Dyna includes a prototype-based approach to object-oriented programming we call *dynabases*. Dynabases are designed to resemble more typical approaches to OOP, as in a procedural programming language. Dynabases are denoted using curly braces {} with Dyna rules inside. The optional keyword **new** can be used to syntactically distinguish between dictionaries (section §2.1.1.1) and dynabases when needed. For example, we can define a simple Dynabase on line 220.[43]

```
220 e = { count += 1.
221       count += 2. }.
222 assert e.count = 3.
```

We can also modify a dynabase after it has been created:

[42]Warnings for dead code can only happen if Dyna can identify the dead code. In general, this is tricky as identifying dead code is equivalent to just *running* the program. Furthermore, some code might only be *temporarily* dead and become used when other rules are added to the program.

[43]There is currently disagreement among the Dyna team about the design of dynabases. It is possible that a future implementation of Dyna will have dynabases that behave differently. The design presented here is my own design and corresponds with how dynabases work in my implementation.

```
223 e.count += 3.
224 assert e.count = 6.
```

Dynabases can be duplicated as well as extended with additional rules by using the keyword **new**. Rules defined in a dynabase can reference the dynabase itself using the keyword **$self**.

```
225 copy_e = new e.             % create a copy of a dynabase
226 extend_e = new e {          % extend with additional rules
227   % $self[44] is used to reference fields and functions on the dynabase
228   bar(X) = $self.foo * X.
229 }.
```

The extended and copied dynabases are distinct from their parent and can be modified individually. However, changes to the parent dynabase are visible in the children even after the children have been "created." The reason for this is that Dyna is declarative. Hence the order in which rules are defined does not impact the expressions in the program. For example, we can define a new rule `foo` on the `e` dynabase on line 230. This rule will be visible on all descendants dynabases, and it is even possible that rules defined before `foo` was defined (line 228), can still reference the definition of `foo`.

```
230 e.foo = 123.                % define a new field foo on the parent
231 copy_e.count += 4.
232 assert copy_e.foo = 123.    % the parent changes are visible in children
233 assert copy_e.count = 10.   % existing rules are extended
234 assert extend_e.bar(2) = 123*2.
235 assert e.count = 6.         % the parent is not impacted by children
```

[44]**$self**'s behavior is similar to that of **self** in Python. **$self** is not required on the functor name of defined rules (bar on line 228), but it is required inside of the expression that defines the rule when referencing other terms defined on the dynabase.

2.9.1 Dynabases vs Procedural Programming OOP

As mentioned above, dynabases are a bit different from classes in a procedural programming language since Dyna is a *declarative* programming language that can be evaluated out-of-order. Another difference between Dyna's dynabases and procedural programming's OOP is that a dynabase might not *actually* be "created." To see what I mean by this, consider for a moment that due to aggregation, we can compute the aggregated result of multiple contributions at the same time. In fact, we can even combine an *infinite* number of expressions. As such, it is possible to write a program where we create an infinite number of dynabases (e.g. lines 236 to 237).

```
236 return_dynabase(X,Y,Z) = { value = ... . }.
237 result min= return_dynabase(X,Y,Z).value.
```

We are able to handle programs with an "infinite" number of dynabases as long as we are able to deduce some representation for the `value` term, which can be used for solving the `min=` aggregator on line 237.

2.10 Embedded Domain Specific Languages

Support for Domain-Specific Languages (DSLs) is a useful feature for programming languages to have as it enables library developers to support a wide variety of different problem domains. Early languages such as LISP were popular among AI researchers in part for this reason. These days, ML researchers are clearly demonstrating that they want access to DSLs despite the fact that Python (the language that is often used for ML) does not easily support macros. This has led libraries such as JAX [23] and PyTorch Script [2], which use Python's reflection to access and manipulate Python ASTs for a function. Given that Dyna is targeting the same kinds of researchers, we believe having the ability to write macros to transform Dyna's AST before it is loaded and also use multiline strings as embedded languages will allow for other languages to be embedded in a Dyna program.[45]

2.10.1 String DSLs

A multiline string in Dyna is escaped between the symbol `'{` and `}`. The string can contain `{` and `}` as long as they are balanced. Furthermore, Dyna-style comments, a percent sign `%` followed by a new line, are also stripped from the string. This is done to encourage `'{}` to be used to embed DSLs that have similar behavior to the Dyna language. Furthermore, Dyna includes special syntactic handling inspired by languages such as Ruby and Lua for passing a block of `{}` to a function. When `{}` appears after a function call in any form (string, dictionary, or dynabase), it will be passed as the last argument to the function. Together, this allows us to write a program as follows for embedding a context-free grammar (line 238) and a linear program (line 257):

[45]There are have other proposals from Dyna team about how DSLs could be implemented in Dyna. The design here is my own design and a contribution of this dissertation.

```
238 my_grammar = grammar'{
239   S -> NP VP
240   NP -> Det N
241   NP -> NP PP
242   VP -> V NP
243   VP -> VP PP    % inline dyna comments are removed
244   PP -> P NP     % when converted to a string
245
246   NP -> Papa   {} % balanced {} can be included in the string
247   N -> caviar
248   N -> spoon
249   V -> spoon
250   V -> ate
251   P -> with
252   Det -> the
253   Det -> a
254 }.
255
256 my_lp = solution[X,Y,Z,Objective]
257   for Objective = linear_program({X,Y,Z}) '{
258 Maximize                     % embed a linear program using the
259     obj: X - 2.3Y + 0.5Z     % LP file format
260 Subject
261     c1: X - Y + S <= 10.75
262         -Z + 2X - S >= -100
263 }.
```

For grammar defined on line 238, the entire grammar from lines 238 to 253 is passed to the `grammar(·)` term as its only argument. The `grammar` term can create dynabase to represent the grammar. For the linear programming example, the first argument to `linear_program(·,·)` is the dictionary `{"X"->X, "Y"->Y, "Z"->Z}` and the second argument is the linear program represented as a string. The implementation of `linear_program(·,·)` is responsible for matching the linear program's variable names with the dictionary's variable names.

2.10.2 Macros

Macros have access to the AST of the Dyna program. The Dyna AST is represented using the same structured-term object that is passed around in Dyna programs. This means that we can use the same pattern-matching tools that we presented in section §2.1.1. Additionally, we have backtick `` `() ``, which is used to escape the AST and embed variables used to match against part of the AST. In the following example, on line 266, we match `_ + 2` and replace it with `_ + 3`:[46]

```
264 :- macro my_macro/1.
265 my_macro(X) := X.                % no change to AST if it does not match
266 my_macro(`(`X + 2)) := `(`X + 3).                          % change AST
267 
268 f1(W) = my_macro(W + 1).
269 f2(W) = my_macro(W + 2).
270 
271 assert f1(3) = 4.
272 assert f2(3) = 6.
```

More practically, macros could be used to implement symbolic auto differentiation (just like JAX [23]), against the AST of a Dyna program. Line 273 shows a conceptual example of how an automatic differentiated neural network could be used in Dyna.[47]

[46]It is helpful to observe the Dyna AST when developing macros by running in the REPL: `print $ast'{ my_program = 1. }`.

[47]The macro `auto_differentiate` is not implemented at this time. I suggest in section §16.2.1 that future work should consider implementing libraries like this in Dyna.

```
273 network(InputMatrix) = auto_differentiate {
274   relu(X) max= X.            % define a neural net unit using a dynabase
275   relu(X) max= 0.
276 
277   layer1(I, J) = InputMatrix.elem(I, J).
278   layer2_input(I, J) += $self.layer1(I,K) * $self.weight(K, J).
279   layer2(I, J) = $self.relu($self.layer2_input(I,J)).
280 }.
281 
282 gradient = network(...).gradient(...).  % gradient term added via macro
```

The '.gradient' term is added to the 'network' dynabase by the auto_differentiate macro called on line 273.

Chapter 3

Related Work

In this chapter, I will cover some related projects and programming languages and how their implementations work. With the Dyna project, it is a bit difficult to make direct comparisons. The reason is that the Dyna language contains combinations of features that are not entirely supported by other languages. A lot of the challenges with Dyna have been a result of how features *interact* with each other, rather than the addition of any individual features—this will be discussed further in chapter §4.

Further complicating this chapter, some of the systems/languages that I compare against are not specific implementations but rather a large class of related systems—for example, I will discuss *SQL databases* and how they relate to Dyna.

I note this chapter is focused on the *implementation* of similar declarative and logic programming systems. This chapter is not intended as a tutorial on the features or syntax of other systems. The reason for this is that, except for chapter §2, this dissertation is focused on implementation and not the Dyna language itself.

I have attempted to write the rest of this dissertation starting at chapter §5 with few dependencies on this chapter, so this chapter can be skipped for those who are only interested in the academic contributions of this dissertation.

3.1 Logic Programming Languages

We start with logic programming as exemplified by languages such as Prolog [9, 35, 38, 52, 101, 108, 132, 144, 147] and Datalog [30, 76, 119, 140].

Logic programming is commonly associated with the declarative programming paradigm, where the programmer should only have to specify *what*, rather than *how*, to perform a computation [100]. Unfortunately, in my opinion, logic programming languages fail to deliver on being "truly" declarative, as I will show when discussing how these languages work.

The way logic programming pursues its declarative goal is by defining a program via logical clauses instead of procedural operations. The usual syntax is similar to the syntax of Dyna that we saw in section §2.1. The only "aggregator" that logic programming languages support is the `:-` aggregator, which is intended to look like a backward implication symbol ($\Longleftarrow$). The **head** of the expression, or the part to the left of the `:-`, is the expression that we are trying to prove by depending on the predicates that appear on the right-hand side of the `:-`.

```
a(X) :- b(X,Y), c(Y).
a(X) :- d(X).
```

Variables represented placeholder values, just like with Dyna.

The way in which logic programming languages "execute" differs greatly depending on the kind of logic programming language used. That said, one of our goals with Dyna is to unify the two major approaches for executing logic programs. We hope that our execution approach will simultaneously benefit from the advantages both approaches have to offer while minimizing their respective weaknesses.

3.1.1 Prolog Language

The oldest and most iconic family of logic programming languages is Prolog. For reference, the first appearances of Prolog happened in 1972, the same year the C programming language was developed [5, 35, 38, 144]. Over the years, there have been countless implementations and extensions of the Prolog language [9, 21, 43, 52, 137, 144, 147, 152]. The diversity in Prolog implementations is not surprising when one considers that a simple Prolog can be implemented in less than 200 lines of code in a procedural programming language [137].

Prolog's approach to execution certainly shows some taint of the limited memory environment in which it was invented. In fact, writing a large Prolog program often requires extensive knowledge of *how* the Prolog system executes a program.

Prolog searches for the *existence* of a proof, utilizing a greedy backtracking-based search method. The way this works is that Prolog searches through declared predicates in a top-to-bottom, left-to-right order, using unification [102] to check if the current assignment to variables is consistent. As long as the current partial assignment is consistent, Prolog continues to expand the program. When Prolog eventually reaches the "end" of unification, it returns the value "*true*" (typically printed as "*yes*") to the user as well as the assignment to variables.

To make this description more concrete, consider the program in figure 3-1.

```
285 a(X) :- X = 0.
286 a(1). % conceptually equivalent to line 285, instead with 1 instead of 0
287 a(2).
288 a(3).
289 a(4).
290 a(5).
291 a(6).
292 a(7).
293 a(8).
294 a(9).
295 b(A,B,C,D,E,F,G,H,I,J) :- a(A), a(B), a(C), a(D), a(E),
296                           a(F), a(G), a(H), a(I), a(J).
```

Figure 3-1. Simple Prolog/Datalog Program used to illustrate the differences in language execution. This program defines `a/1`[48] as *true* for the integer values of 0-9, lines 285 to 294, and `b/10` as *true* for all terms between 0000000000 and 9999999999.

If we query `b(A,B,C,D,E,F,G,H,I,J)` on line 295, then the Prolog engine returns a *lazy* stream of *bindings* to the variables `A,B,C,D,E,F,G,H,I,J`. A **binding** is the current assignment to the variable. A binding can be either a ground value (such as the number `0`, 1, 2 etc.), another variable, or a structure (such as in section §2.1.1). The Prolog engine searches for a binding to all variables, which is consistent

[48]Prolog systems will usually refer to rules using their name and arity (number of arguments). In this case, `a/1` refers to the 'a' rule defined on lines 285 to 294.

with the rules of the program. It does this by expanding rules in the standardized *left-to-right*, *top-to-bottom* evaluation order. In the case of a/1 and b/10, it starts by expanding b/10, where it encounters the first clause a(A). Then it will unify the variable A with the variable X on line 285. At this point, neither the variable A nor X has any *ground* value assigned to it. The Prolog engine continues to evaluate the program in its left-to-right order and encounters X=0 on line 285. This causes the Prolog engine to assign the value 0 to X, which causes the value of A to also be set to 0 as a result of the previous unification between A and X. Because there are no more clauses on line 285, the Prolog engine returns control flow back to line 295 where a(B) will then be evaluated, and so on. However, before this return of control flow occurs, the Prolog engine marks a point in its execution, which is used for backtracking.

Once the Prolog engine has reached the end of line 296, it will return the current binding to the caller of b/10. In this case, it was called from the top-level user query, so the assignment A=0,B=0,C=0,D=0,E=0,F=0,G=0,H=0,I=0,J=0 is returned to the user. If we ask the Prolog engine for the next assignment, then it will go to its stack of backtracking locations and pop the most recent location. In this case, it would be where J is assigned the value 0. The engine then advances to the next disjunctive branch of a/1, in this case line 286, assigning the value 1 to J. Again, there are no more conditions on line 286 or line 296 which need to be handled, so the new binding is returned to the user, which is now `A=0,B=0,C=0,D=0,E=0,F=0,G=0,H=0,I=0,J=1`. This process will continue until `A=9,B=9,C=9,D=9,E=9,F=9,G=9,H=9,I=9,J=9`, at which point there are no more possible bindings to consider for the variables `A–J`.

This procedure is conceptually quite simple and has the advantage of only using a limited amount of memory—it only requires storage for the current assignments of the variables and records of where branching decisions were made so it can backtrack. Unfortunately, writing good Prolog programs requires that the programmer think about *how* the Prolog engine works, and therefore is not entirely "declarative".

For example, suppose that we want to use `b/10` to query if there is an integer where the second digit is larger than the first digit, e.g., a number like 0100000000. This can be done using something like line 297:

```
297 c :- b(A,B,C,D,E,F,G,H,I,J), A < B.
```

Figure 3-2. Checking if there is a number where the second digit is greater than the first digit. For example, something like 0100000000.

Now, line 297 does "*work*"; however, it is extremely inefficient. The Prolog engine always evaluates left-to-right, which means that it creates a complete assignment to the variables `A–J` before it checks `A < B`. This means that Prolog loops through 10^8 combinations before finding an assignment that satisfies `A < B`.

Ideally, we would like to *intermix* the checking of `A < B` with the assignment to variables A and B. Thankfully, modern Prolog implementations have a solution to this. Prolog (with a CLP extension, section §3.1.5) allows tracking of *delayed constraints*, which are constraints (like A < B) that cannot be immediately evaluated. These constraints are conceptually similar to the A=X constraint that was caused by the unification on line 285. However, these constraints are "more powerful" in that instead of depending on manipulation of pointers in internal data structures (see Martelli and Montanari [102]), these constraints are allowed to define their own *arbitrary code* for handling when a variable's state is modified.

Now, when using delayed constraints, we must still remember that Prolog evaluates left-to-right, as adding the delayed constraint after b/10 does not help:

```
d :- A #< B, b(A,B,C,D,E,F,G,H,I,J).
```

Figure 3-3. Delayed Constraint in Prolog. Delayed constraints are annotated using a hash symbol #. [9, 147]

The way that A #< B works is that it is saved as a delayed constraint into the constraint store, or program's state (which previously only held variable binding), and will run the #< code anytime that A or B is "*modified*". This means that when B is assigned the value 0, and A already has the value 0 assigned, it will quickly fail the check of A #< B, causing the Prolog engine to immediately backtrack.[49]

Now, this is great! But there is still some behavior that is a bit "annoying" (and

[49]I should note that unification is usually presented as working with structured terms (as in section §2.1.1) instead of delayed constraints on numerical values.

also differs from Dyna's approach). If we query for 'd', the Prolog engine will quickly prove it is true but returns a lazy stream that keeps returning solutions. In other words, rather than finding one solution to prove 'd', the Prolog engine enumerates *all* $45 * 10^8$ proofs of 'd'. This happens because every time that we ask for the next item in the lazy stream of solutions, Prolog simply backtracks to the previous variable, even if the returned solution (that 'd' is true) is *identical* to the previously returned solution. This differs from Dyna, which would only return 'd' is true once due to the aggregation performed by `:-` over all of the answers.

To fix this, we can use a *cut* in Prolog, which prevents the Prolog engine from backtracking over different assignments to variables. A cut in Prolog is annotated with an exclamation mark '!'.

```
e :- A #< B, b(A,B,C,D,E,F,G,H,I,J), !.
```

Figure 3-4. Adding a cut to avoid backtracking through all possible assignments.

Now the 'e' rule will only find one solution and return *true* one time.

This again shows how Prolog programmers need to be aware of how the Prolog engine works internally to make their programs work well. Conversely, Dyna's is designed to not need operations such as cut. We believe that aggregators such as `:-` and `?=` provide sufficient mechanisms for handling these cases.

3.1.1.1 Infinite Relations in Prolog

Prolog treats unification between variables as a first-class built-in operation. This means that expressions such as `X=Y` are handled without backtracking over possible assignments to `X` or `Y`. This is achieved by tracking which variables are unified together by essentially updating the internal pointers backing `X` and `Y` so that the space in memory reserved for holding the value of `X` and `Y` become the same space [102]. This also extends to structural terms. So unification like `X=g(Y,7)`[50] are handled in a similar way.

This approach lets the Prolog engine work even when there are infinite relations by tracking "constraints" and leaving the exact value of some variable as *unknown*.

Therefore, the extension of modern Prolog with delayed constraints (as in figure 3-3) can be seen as an extension of unification from tracking only variable unification via pointer reassignments to allowing arbitrary code to run when a variable's "unification status" is changed.

```
f(X,Y) :- Y=g(X,7,V), V #= X + 3.
```

Figure 3-5. An example infinite relation in Prolog. `f(X,Y)` does not require any backtracking to evaluate. It will define an infinite number of terms like `f(1,g(1,7,4)`.

[50]The notation `Y=g(X,X)` is how Prolog does structural terms (section §2.1.1). Prolog does *not* have automatic evaluation, as we have in Dyna. Hence, Prolog uses parenthesis `()` for both calls and unification with structures.

3.1.2 Datalog Language

Datalog is the other major approach to logic programming [30, 76] and was the primary inspiration for Dyna 1.0 [60, 61]. A short description of how Datalog works is that it uses the same high-level syntax as Prolog for dynamic programming [18] over a boolean semiring.[51] Let us break this down a little. Datalog's approach can be summarized as, the Datalog engine stores *all* true facts. A Datalog **fact** is a statement that is known to be true and is represented as an term (without any variables). For example, `a(1)` is a fact, but `a(X)` is not a fact as it contains the variable `X`.

Datalog uses the rules in the program and all the stored facts to deduce new facts—storing those as well. A Datalog engine finishes running when the deduced facts are "stabled"—meaning that it has reached a fixed point, just like Dyna section §2.5. Because everything is stored, this creates different kinds of opportunities for how to execute a Datalog program.

First, because everything is stored, this means that even simple programs can cause a Datalog system to be quite inefficient. For example, figure 3-1 would end up storing 10^{10} facts in memory to represent `b/10`. The simplicity of Datalog means it does not have a mechanism to efficiently represent this program.

Conversely, having a simple "store everything" approach does have some ad-

[51]See footnote 3 for an explanation Semirings.

vantages. For example, Datalog systems will often employ a host of efficient join techniques as in a database system (e.g. [106]). Datalog can also employ brute force strategies such as “loop over everything” and be *guaranteed* that these strategies terminate. The reason is that a Datalog program can only represent finite relations.[52] Being finite is both part of Datalog’s design but also follows as a result of Datalog storing *all* facts, and computers have a finite amount of storage. Hence, there can only be a finite amount of facts stored.

3.1.3 Datalog is Breath First, Prolog is Depth First

Another advantage of Datalog’s approach is that it avoids Prolog’s greedy search behavior. For example, in the following program:

```
301 h :- h.
302 h :- true.
```

Figure 3-6. Datalog program which deduces that ‘h’ is true due to line 302.

Datalog easily deduces that ‘h’ is true due to line line 302. Once ‘h’ is stored in memory, the Datalog system does not get stuck handling line 301 and can easily solve this program. Whereas, Prolog gets stuck on this program. The reason is that Prolog would backtrack through line 301 and reencounter ‘h’ endlessly, without ever attempting to run line 302, due to Prolog’s top-to-bottom, left-to-right evaluation order.

[52]This means that there are no infinite relations like in Prolog.

In this way, we can roughly think of Datalog's approach as *breath-first* evaluating of logic programs while Prolog is *depth-first* evaluation. Both of these languages exhibit the relative advantages and disadvantages of these breath vs depth first search: memory overhead, robustness to search order, complexity of internal state (with unified variables in Prolog), etc. In Dyna, we are essentially looking to combine the best of both Datalog and Prolog. We want to avoid cases where a bad execution order causes the system to not terminate or be inefficient. We also want to be able to store deduced facts (like Datalog), while still representing infinite relations (like Prolog).

3.1.4 Aggregation in Logic Programming

In Dyna, a central feature is aggregation, which appears everywhere throughout a Dyna program and is built into the Dyna syntax (section §2.2). Both Datalog and Prolog "support" aggregation, though their implementations are less flexible than Dyna's aggregation and it is essentially bolted on. The way aggregation has been implemented is by adding "meta predicates" that internally perform the operation of aggregation. In Prolog, this might look something like figure 3-7:

```
303 j(X,AggResult) :- bagof(InputToAggregator,
304                          body_getting_aggregated(X,InputToAggregator),
305                          AggList),
306                    sumlist(AggList,AggResult).
307 body_getting_aggregated(1,1).
308 body_getting_aggregated(1,2).
309 body_getting_aggregated(1,1).
```

Figure 3-7. Aggregation supported by Prolog[53]

Here bagof/3 will collect all of the assignments to InputToAggregator into a list AggList.

Evaluation of bagof/3 requires that *all* possible assignments to body_getting_aggregated/2 are completed *before* bagof/3 "returns". This prevents any opportunity to make the aggregator more efficient, as we will see later in section §6.5 with Dyna.

Note that line 307 and line 309 both assert that $1,1$ is true. As a result, the aggregation from Prolog will give a different result than Datalog. In Prolog, it will count the number of times that something has been deduced. In this case with Prolog, we will have 'AggList$=[1,2,1]$'. However, with Datalog, we get 'AggList$=[1,2]$', since the second number 1 was already deduced and not counted a second time.

[53] Documentation about Prolog's meta predicates that enable aggregation: https://www.eclipseclp.org/doc/bips/kernel/allsols/bagof-3.html https://www.swi-prolog.org/pldoc/man?predicate=bagof/3 https://eclipseclp.org/doc/bips/lib/fd_global/sumlist-2.html
Documentation for aggregation in Datalog: https://souffle-lang.github.io/aggregates

3.1.5 Constraint Logic Programming

Constraint Logic Programming (CLP) is a derivative of Prolog-style logic programming. As we already saw in figure 3-3, a delayed constraint is an expression that cannot be immediately evaluated. These constraints are stored in a *constraint store*, alongside unifications between variables. Constraints are re-evaluated when variables are assigned. Constraints can also interact with other constraints to infer new constraints through *propagation* [72].

For example, if we have ‘j’ defined on line 310:

```
310 j :- A #> 5, B #= A - 2, B #< 0.
```

Figure 3-8. The variable A, must be some value greater than 5, and B must be less than 0. Additionally, we have that B = A-2, hence there is no valid assignment to both A and B.

Then there is no possible way to assign A and B such that ‘j’ is *true*. This program can be handled by CLP using interval constraints. First, `A #> 5` will track that A *must* have a value greater than 5 (the interval $(5,\infty)$). Next, the constraint `B #= A - 2` identifies that B is two less than A, therefore, it will have an interval of greater than 3 associated with it (the interval $(3,\infty)$). Finally, the constraint `B #< 0` will associate an interval of less than 0, which does not overlap with the greater than 3 interval ($(-\infty,0)\cap(3,\infty)=\emptyset$). Hence, the CLP engine initiates Prolog-style backtracking, removing constraints and unifications from the constraint store.

To make CLP work, a CLP system will have hundreds of different propagation and simplification rules. These rules are sometimes called Constraint Handling Rules, usually referred to as CHR [72]. Development of new CHRs and ways to implement CHRs has been an area of research over the years [72, 105, 118, 131]. Some of the rewrite rules presented in chapter §6 are similar to the rules found in CHR [72].

3.1.5.1 MiniKanren

MiniKanren [27, 71] was designed as a small implementation of logic programming that can be implemented in less than 200 lines of Scheme. As a result of MiniKanren's small core, it has been the basis for much research on logic programming over the years [28, 91]. MiniKanren's approach to logic programming differs from Prolog in that it does not mutate any global data structures state[54] but instead returns lazy streams of binding states. Each operation is implemented as a function that modifies the maps from a stream of bindings to a new binding, returning no elements in the event that there is an inconsistency between the variable bindings [26].

The approach taken by miniKanren is similar to our approach in that we are going to avoid mutating global state (like Prolog); however, we do not depend on passing the host's language functions around (like miniKanren) and instead will develop an explicit *relational expression* representation (**R**-exprs in chapter §5). This

[54]On the contrary, in Prolog the unifications between variables are tracked globally and maintained using stack discipline when Prolog backtracks in the case of failure.

will allow us to be more flexible with the kinds of mutations and expressions that we can represent and to have more flexibility in picking the order of evaluation.

3.1.6 Constraint Satisfaction Programming

Constraint satisfaction-based solving techniques, such as SMT solvers [15, 16, 47, 48, 50, 107] or answer set programming [64, 79] work by representing a problem as variables and constraints between variables. These systems will often have some ability to handle "function calls" by expanding the function up to some depth.[55] The representation of a problem is then translated into a mathematical theory that can be solved. For example, the SMT formalism to multiple calls to a modified SAT solver [16]. These approaches have the advantage of residing on the foundation of a sound mathematical theory. It is possible that we could solve Dyna programs the same way. However, we have opted not to take this approach. The reason is that we do not believe that it will work well with the kinds of problems we are interested in. The kinds of Dyna programs we expect will realistically have many more variables than can be realistically solved using these kinds of approaches.[56] Instead, we have opted for a *fixed-point* based solving technique, which will iterate until a solution is found (section §2.5).

[55]When the expanded version of the program is greater than the max depth needed to represent all function calls in the program, then the solver is able to represent all intermediate values that the program would compute inside of the solver.

[56]As approaches that are built on SAT solving can take an exponential amount of time to solve.

3.2 Probabilistic Programming

Dyna is *technically not* a probabilistic programming language, though it is frequently compared to probabilistic programming languages.

A probabilistic program assigns *probabilities* to different assignments of the variables [29, 37, 65, 81, 112, 138, 139]. In some ways, we can think of this as a generalization of logic programming, which only assigns *true* or *false* with being able to assign factional chances of being true. At a high level, the probability is similar to Dyna's ability to weigh the result from a rule and combine them with an aggregator (section §2.2). However, the difference is that Dyna allows for *general* weights, which might not be a probability, and probabilistic programming *requires* that the weights be probabilities. Because probabilistic programming works with a more restricted problem, they will often provide features that are useful for modeling probabilities. For example, probabilistic programming systems have the ability to fit parameters to observed data.[57]

Like Dyna, probabilistic programming systems are usually not complete languages but are either embedded as a DSL or used as a library from a driver program like Dyna (as in section §2.3.1).

[57] It is conceptually possible that Dyna could add these features in the future as DSL in Dyna (e.g. section §2.10).

3.3 Relational Algebra

In chapter §5, I will go into detail about how we use a relational algebra we call **R**-exprs to model and implement the Dyna programming language. Chapter §5 will provide an introduction to relational algebra, so here, I will instead focus on surveying related work that uses a relational algebra.

The most iconic systems that are built on a relational algebra are SQL[58] database systems [36, 44]. A SQL database consists of database tables and is queried using SQL queries. *Relations* in a database are represented by database tables. Tables are combined (intersected or unioned) with other tables, filtered for particular values, and projected to select for a subset of the database table. The result is represented as another database relation. The relation can be either returned to the user (as in the case of a query), saved back into the database as a new relation, or further extended and used as a component of a larger query. An example SQL query is shown in figure 3-9:

[58]SQL: Structured Query Language

```
SELECT column_a, column_b, column_c            ▷Select columns using projection
FROM   table_1
JOIN   table_2 on table_1.column_a = table_2.column_f     ▷Intersect relations
WHERE  column_d = 'identifier_1' and table_2.column_e > 11          ▷Filter
```

(a) SQL

```
result(A,B,C) :- table_1(A,B,C,'identifier_1'),      % filter with value
                 table_2(E,A),             % intersect by reuse of var A
                 E > 11.                  % filter as external constraint
```

(b) Equivalent Dyna

Figure 3-9. Example SQL query

Internally, SQL databases work over finite materialized relations. The relations provide internal APIs to access the underlying data. This can include filtering using a particular key or looping over all tuples in the relation using an iterator [134]. Database systems include *query optimizers* that rearrange operations in the query to automatically figure out the most efficient way to run queries.[59]

Just like Dyna, SQL supports aggregation and grouping [46, 83, 98]. In both SQL and Dyna, aggregation is represented as reducing a relation into another equivalent relation with the result of aggregation. An example is shown in figure 3-10.

[59]Dyna can also reorder operations, like a database (section §2.4). At this time, Dyna does not include an *optimizer* that finds the best way to arrange computation, but this is proposed as future work (section §16.4).

```
SELECT   column_a, sum(column_b)
FROM     table_1
GROUP BY column_a
```

(a) SQL

```
result(A) += B for table_1(A,B,C,D).
```

(b) Equivalent Dyna

Figure 3-10. Example SQL with aggregation and `GROUP BY`.

3.4 Term rewriting

Term rewriting [10, 12, 13, 17, 32–34, 51, 77, 78, 93, 94, 114, 143] is a central idea used in this dissertation (chapters 5 and 6). Essentially, given an expression or *term*, it is rewritten into an *semantically equivalent* representation. This rewriting process corresponds with *execution* of the program. For example, if we have the term $2+3$ it can be rewritten as 5, which is semantically equivalent and represents the evaluation of the plus sign.

The abstraction of term rewriting is too broad to completely cover, so I will instead focus on a few subtopics that are related to the ideas explored in this dissertation.

3.4.1 Implementation of Term Rewriting

A term rewriting system is defined using both the structure being rewritten and the rewrite rules performed against the structure. In this dissertation, I will create our own implementation of term rewriting based on the rewrite rules that we defined (chapters 8 and 11). While creating a term rewriting system from scratch is entirely reasonable, there are frameworks such as maude [33, 34] or k-framework [114] that create an implementation from the definition of rewrite rules only. As will be discussed in chapter §8, the reason that we choose to create our own "rewriting framework" for Dyna, rather than using an existing framework, is that the way in which we apply rewrite rules differs from what existing frameworks are designed to support. The existing frameworks are designed to support procedural languages, where the order of execution is deterministic. Therefore, they only have to match their rewrite rules against the next instruction to execute. Conversely, in Dyna, our execution is *non-deterministic*, and we implement this by allowing any applicable rewrite rule to match any part of the term.

3.4.2 Term Rewriting a Relational Algebra for Logic Programming

Term rewriting on top of a relational algebra for the implementation of logic programming has been experimented with before: Arias et al. [10], Bellia and Occhiuto [17], Gallego Arias et al. [77, 78]. Bellia and Occhiuto [17] was the first that we are aware of to make the connection in 1993, they created a "*variable free*"[60] representation of logic programming they called *c-expressions*. They also identified that their relational algebra representation can be manipulated with rewrite rules, as we will do in chapter §6. Arias et al. [10], Gallego Arias et al. [77, 78] is an ongoing project

[60]Their "variable free" representation the integer index in a tuple to replace the variable *name*. We choose to use a *name-based* representation as we believe it is cleaner, section §5.1.1.

which also uses a term rewriting-based formalism to implement logic programming. Like [17], their systems is focused on logic programming and does not support aggregation as we do with Dyna.

3.4.3 Functional Logic Programming

Dyna started as an extension of Datalog and has added features of Prolog and eventually added enough features to (in my opinion) be comparable to functional programming.[61] As such, I think that *functional logic* programming languages and the version of Dyna in this dissertation share a number of similarities.

The space of functional logic programming has been around for several years, with a journal running from 1995 to 2008 [3] and at least two other programming language research projects in this space: the Curry language (Antoy [6], Antoy et al. [7], Antoy and Hanus [8], Braßel et al. [24], Hanus [84, 85], Hanus et al. [86, 87], Hanus and Prehofer [88], Hanus and Sadre [89]) and Verse language being developed at Epic games (Augustsson et al. [11]). Both Curry and Verse are Haskell-esque languages with the addition of logic programming features. The major addition to functional programming is the ability to represent non-deterministic results from functions and assignments to variables. For example, a function can be called when its arguments' values are unknown, and only the return value is

[61]The version of Dyna in this dissertation supports higher-order functions, lambdas, closures, lazy evaluation, immutable data structures such as hash maps, and the standard logic programming lists and structured terms. User-defined rules are side-effect-free and have a functional dependency between the arguments to the rule and the value returned from the aggregator.

known. This is just like logic programming, which allows arguments to functions to be variables with unknown values.

The formalism for these languages is done using term rewriting, just like I do in this dissertation. There are two approaches to handling the non-determinism: *narrowing* and *delayed evaluation*[62] (as we see later in this dissertation).

The term *Narrowing* refers to the idea of representing a program as a set of equations with variables whose value is unknown and then solving those equations. Narrowing does not refer to a specific strategy for solving the equations, but there has been work on developing narrowing strategies for different classes of equations and for more efficient search strategies [6, 7, 85, 88].

3.5 Memoization & Reactive

A long review of *what memoization is* can be found in section §10.1.

Memoization is the programming technique of saving the result of some computation and reusing it later rather than recomputing it every time [104]. The simplistic view of memoization requires that memoized computation is *functional* and *unchanging* so that the stored result does not have to be invalidated later. This can be fixed by making a computation *reactive*, upstream dependencies for any memoized value are tracked, and whenever there is a change, all downstream dependents are recomputed [14]. (The exact details of how reactive programming

[62]Also called *residuation* in Antoy [6].

is implemented differ greatly between implementations.)

In the context of logic programming, languages such as Datalog (section §3.1.2) are entirely based on memoization, with everything stored instead of "calling a function to perform computation". XSB is an extension of Prolog which adds memoization [135, 145, 151]. XSB works by placing markers on terms that are expanded during backtracking. If a cycle is detected during backtracking, it memoizes that the term where the cycle is detected is memoized as *false*. If something is later deduced as true later in the computation, it triggers recomputation.

Prior work from the Dyna project has also focused on developing a formalism for memoization and reactive programming: Filardo [66], Filardo and Eisner [67]. This work sought to formalize programs as "*computational circuits*" where the memoized values are vertices in a graph and edges in the graph track computation dependencies. A vocabulary of different message types that can be passed along the edges was developed that corresponds to different kinds of implementations of reactive programming. For example, there are different options around how the recomputation is scheduled and when changes are visible to downstream dependents.

3.6 Tracing JIT Compilation

In chapter §12, I will discuss my efforts to compile Dyna to make it run faster. Our compilation method is inspired by *traced-based JIT compilation* [31, 74, 75] and will also make use of partial evaluation [42, 57, 73, 116, 149]. Trace-based JITting is a very flexible technique for implementing a JIT compiler has been previously used on the first JavaScript JIT compilers [74], the PyPy Python JIT [20], LuaJIT [109], the TorchScript for neural PyTorch models [2], and even Prolog [21].[63] Tracing differs from *method at a time* compilation in that instead of compiling methods that appear in the program, it compiles a sequence of steps performed when executing the program. When there is a conditional branch in the program, the tracing compiler inserts a *check* that the conditional branches the same way each time. The JIT compiler will insert a stub that is used to resume the trace, and only when the branch is actually taken does it generate the code.

To see how trace-based JIT compilation works, let us work through an example of compiling the program presented in figure 3-11 using tracing. This program starts with ExampleFunction and uses the `MyPrint` function to indirect to the built-in Print call.

[63]Prolog has also been compiled using method-based compilation with research projects such as the YAP compiler [43] and the Mercury project [39].

```
function ExampleFunction($n$)
    for $i \in [0, n)$ :
        if $i < 5$ :
            MyPrint(“hello”)
        else if $i > 10000000000$ :
            MyPrint(“never”)
        else
            MyPrint(“world”)
    MyPrint(“done”)
    return
function MyPrint(x)
    Print(“something ” + x)
```

Figure 3-11. Example function getting traced

When we generate a trace of figure 3-11 in figure 3-12, observe that we have *not* generated all of the code from figure 3-11. Instead, we have some lines that are marked “*not yet generated*”. These lines of code contain sufficient metadata and jump statements to get back into the JIT compiler to resume tracing.

```
1:  $i \leftarrow 0$
2:  if $i \geq n$ :                              ▷ Condition to check
3:      not yet generated
4:  if $i \geq 5$ :                              ▷ Condition to check
5:      not yet generated
6:  if $i > 10000000000$ :
7:      not yet generated
8:  Print(“something hello”)                     ▷ The MyPrint function is embedded
9:  $i \leftarrow i + 1$
10: goto 2                                       ▷ Return to top of loop
```

Figure 3-12. First generated version of the function ExampleFunction from figure 3-11 using tracing.

Observe that the trace has stopped with line 10. The reason is that when tracing, the control flow has jumped back to a location in the code that has already been generated. Therefore, the tracer compiler will generate the equivalent jump statement to the previously generated statement. Furthermore, observe that the function MyPrint does not appear in the generated output of figure 3-12. The reason is that the trace only contains useful operations, such as incrementing the variable *i* and calling the built-in Print, but the user's function MyPrint simply gets "*absorbed*" in by the processes of tracing.[64]

As the program continues to run, it will eventually hit one of the "*not yet generated*" branches, and it will update jump statements and generate additional code as in figure 3-13.

[64] In a dynamic language like Javascript or Python, the trace may include a check that the MyPrint function was not redefined.

```
 1: $i \leftarrow 0$
 2: **if** $i \geq n$ :                                  ▷ *Condition to check*
 3:     **not yet generated**
 4: **if** $i \geq 5$ :              ▷ *Updated to branch to newly generated code*
 5:     **goto** 13
 6: **if** $i > 10000000000$ :
 7:     **not yet generated**
 8: PRINT(“something hello”)
 9: $i \leftarrow i + 1$
10: **goto** 2
11:
12: ▷ *Additional compilation to fill in code from line 5*
13: PRINT(“something world”)
14: $i \leftarrow i + 1$
15: **goto** 2
```

Figure 3-13. After the first “not yet generated” branch has been hit and additional code has been added.

This process of running the generated code and replacing the “*not yet generated*” branches with generated code will continue. Eventually, the program is done and will hit the return statement, as in figure 3-14. Not all parts of the program have been executed, so there are parts of the generated code that contain “*not yet generated*” on some branches.

```
 1: $i \leftarrow 0$
 2: **if** $i \geq n$ :
 3:     **goto** 18
 4: **if** $i \geq 5$ :
 5:     **goto** 13
 6: **if** $i > 10000000000$ :
 7:     **not yet generated**
 8: PRINT(“something hello”)
 9: $i \leftarrow i + 1$
10: **goto** 2
11:
12: ▷ *Additional compilation to fill in code from line 5*
13: PRINT(“something world”)
14: $i \leftarrow i + 1$
15: **goto** 2
16:
17: ▷ *Additional compilation to fill in code from line 3*
18: PRINT(“something done”)
19: **return**                                ▷ *Generate return to caller code*
```

Figure 3-14. The program has ended as it hits the *return* statement. Not all branches of the code have been hit, so there can still be un-generated parts of the compiled code.

Chapter 4

Challenges in Dyna

Usually the *design* and *implementation* of a programming language are done at the same time. This means that difficult-to-implement features and combinations of features are frequently left out of programming languages. Dyna did not have this luxury. Dyna's design was proposed in 2011 by Eisner and Filardo [59] without an implementation.[65] Furthermore, the long-term vision for the Dyna project is that all programs that are "*conceptually sound*" (as read by a human, not a computer) and describe the programmer's intent (in a declarative way) should "just work". A more formal way of stating this vision would be to say, "all syntactically correct programs which pass some elementary level of type checking should 'work' and return some 'useful' result to the user when queried".

Now, this vision is an amazing pitch. Nevertheless, it certainly complicates Dyna's implementation. For instance, this vision does not provide us with any

[65]There have been some syntactic changes to Dyna since 2011. None of those changes has made the language *simpler*.

insight into the methods to *use* for Dyna's implementation. However, we can *reject* many previous approaches to logic programming implementation, as they are incapable of supporting Dyna's *vision*.

My intention in this chapter is to provide you, the reader, with an understanding of our motivation *before* I subject you to hundreds of pages about implementation. Note: I am *not* selling the *vision* of Dyna—this dissertation is about *implementation*. I consider the vision of Dyna *prior work* published in 2011. I encourage anyone interested in more details on this vision to read the original 2011 paper by Eisner and Filardo [59]. Additionally, I will state that the implementation presented in this dissertation does not completely deliver on Dyna's vision—there exist syntactically correct and conceptually sound programs that we currently cannot run. However, it is our belief that the techniques presented in this dissertation provide a sufficient *foundation* for Dyna, such that the vision of Dyna can be approached "*in the limit*" with additional work and features being added.

4.1 Features in Dyna

We start by looking at the features that Dyna supports and how this compares to other declarative frameworks, as shown in table 4-I.

A high-level outline of features supported by Dyna, as compared to other declarative frameworks, is shown in table 4-I. A high-level description for each feature is as follows:

Feature	SQL §3.3	Datalog §3.1.2	Prolog §3.1.1	CLP §3.1.5	Probabilistic Programming §3.2	SMT §3.1.6	Dyna 1.0 [60]	**Dyna 2.0 (This dissertation) §2**
Finite	✓	✓	✓	✓	✓	✓	✓	✓
Deductive	✗	✓	✓	✓	≈	≈	✓	✓
Updatable	✓	✓	✓	✓		✓	✓	✓
Semiring Weighted	✗	✗	✗	✗	≈	✗	✓	✓
General Weighted	✗	✗	✗	✗	✗	✗	✗	✓
Aggregation	✓	✗ §3.1.4	✗ §3.1.4	✗	✓	✗ [50]	✓	✓
Memoization	✓	✓	✗ [135]	✗	✗	✗	✓	✓
Turing Complete	✗	✗	✓	✓	✗	≈	✓	✓
Unconstrained Execution Order	✓	✓	✗	≈	✗	✓	✓	✓
Constraints	✗	✗	✗	✓	✗	✓	✗	✓
Object Oriented	✗	✗	✗ [49, 146]	✗	✗	✗	✗	✓
Syntactic Sugar		✗	≈				✗	✓

Table 4-I. Different Declarative Programming Languages (Paradigms) compared by feature. The definitions for the features are given on the next page. This table represents what is *most commonly* associated with a particular programming paradigm. References are included in the table when some further research has been done to add a particular feature.

- **Finite** — All systems are able to define finite relations. By finite, we mean that the set of all declared entries is finite $|\{\cdots\}| < \infty$. For example, the integers between 0 and 10 is a finite set $|\{1,2,3,4,5,6,7,8,9\}| < \infty$. Conversely, an example of a *non-finite* relation would be all natural numbers as $|\mathbb{N}| \not< \infty$.

  Only supporting finite means that *simple* strategies that *enumerate everything* are workable. The reason is that *brute force* strategies are *guaranteed* to terminate. Because of the termination guarantee, it makes it easier to experiment with different execution orders in finite systems without worrying about secondary non-termination issues.

- **Deductive** — Deductive means that the system deduces new facts in the language. In logic programming, this is represented as:

  ```
  deduced :- requirement_1, requirement_2.
  ```

  Logic programming languages, such as Prolog and Datalog, are both deductive, whereas languages such as SQL are generally not considered deductive. The SQL language does not automatically perform inferences for new entries in the database tables, though this can be added manually through the use of database triggers.

- **Updatable** — If the program or data can be changed after the program has started running. Most of the systems in table 4-1 provide some kind of REPL where they can be started and maintain state between interactions.

- **Weighted** — Every expression in the system has a *weight*. Logic programming languages only have the single weight of *true*. Some probabilistic programming languages associate a probability with every expression in the system in the case that the language is semiring based (e.g. [65]). Other probabilistic programming languages will randomly sample results from the program and compute the probability for each result depending on how probabilistic factors were used in the execution of the program (e.g. [29, 81, 138]). This allows a probabilistic programming language to represent a result as being 50%

  Dyna allows for a value to be associated with every expression. The value is not required to be a numerical value or even a probability.

- **Aggregation** — Dyna and SQL build aggregation into the language as a central feature. Logic programming with Prolog or Datalog does *technically* support aggregation as shown in section §3.1.4, but the syntax is generally quite awkward and there are many *footguns*[66], in that using the aggregation feature can result in unexpected behavior, as previously described.

- **Memoization** — If there is a mechanism to avoid performing the same computation multiple times. This is admittedly more of an *implementation* feature than a language feature. However, the design of the language does (usually) influence the implementation.

[66]A footgun is a bad feature which is easy to *misuse* and very likely to give surprising results in "*common*" use cases.

Datalog and SQL materialize the result of all intermediate computations and hence are entirely memoized. Prolog generally does not perform memoization at all, but there are some extensions that add memoization (see [135, 145, 151]). The Dyna implementation supports memoization as a central feature, section §2.7 and chapter §10.

- **Turing Complete** — Many declarative programming frameworks are not Turing-complete, such as SQL and Datalog. The reason for this is that these systems instead prioritize termination guarantees, making it easier to rearrange operations/optimize database queries.

  Some SMT solvers are also Turing-complete despite the fact that they internally reduce their computation to SAT solving, which is not Turing-complete. The reason is that SMT solvers attempt many reductions, increasing the size without bounds until a solution is found.

- **Unconstrained Execution Order** — Reordering expressions in a language provides optimization opportunities. SQL is well known for having a query optimizer that attempts to figure out the best order to run operations and perform joins between database tables. Advanced Datalog systems will also reorder operations for better performance. As stated before, Prolog does not allow for automatic reordering of the expressions (section §3.1.1). SMT achieves out-of-order execution and Turing completeness by reducing to a SAT solver, which internally can perform the out-of-order evaluation.

- **Constraints** — Constraints are logical expressions that can be combined together. SQL, Datalog, and Prolog are generally not considered to have constraints, the logic expressions are evaluated immediately when encountered, though SQL and Datalog do allow query optimizers to reorder expressions *before* execution starts. In the previous chapter (section §3.1.5), we mostly focused on constraint logic programming (CLP), which is the extension to Prolog that allows for delayed constraints.

- **Object Oriented** — Most declarative frameworks do not support object-oriented programming. This is a feature that has been added to Dyna (section §2.9). The complication here is that the standard approaches to object-oriented programming are procedural (such as set field on an object, and call function which mutates fields on the object) and do not work well in a declarative paradigm.

- **Syntactic Sugar** — Dyna aims to be a more modern programming language with the addition of syntactic sugar. For example, we have built-in notation for aggregators, dynabases, lambda functions, etc. The other languages in this table are either Prolog-style or SQL, which do not have this level of syntactic sugar. Admittedly, this does not increase the complexity of implementation too much as this is entirely handled by the front-end parser, which is scarcely mentioned in this dissertation.

4.2 Examples of Difficult Programs

To illustrate some of these features of Dyna and how their combination can make executing programs difficult, let us take a look at a few example programs.

4.2.1 All Pairs Shortest Path

The first program is the *all-pairs shortest path in a directed graph* program, which previously appeared in section §2.6. Each edge of the graph is defined using an `edge` rule, whose values (weight) is the length of the edge. By construction, we have for any `Start` and `End`, such that there exists a path between `Start` and `End`, the value of `path(Start,End)` is the length of the shortest path.

```
323 path(Start,Start) min= 0.                                   % base case
324 path(Start,End) min= path(Start,Mid) + edge(Mid,End).  % recursive case
325 edge("baltimore", "washington dc") = 38.      % example rules for edges
326 edge("baltimore", "new york") = 195.
327 ............                              % many other edge rules omitted
```

This program is *not* a valid Datalog program and is also likely to not terminate when evaluated under Prolog. The reason why this is not Datalog is that line 323 defines an infinite relation, which is not allowed under Datalog. For example, the distance between `path("does not exist", "does not exist")` is 0 due to line 323. This line can be modified to work with Datalog by changing it to '`path(Start,Start) min= 0 for city(Start).`' in which case '`city(Start)`' ensures that the set of starting locations are all valid city names. However, this does change the semantics of the program as `path("does not exist", "does not exist")` changes from the value of 0 to no longer being defined or *null*.

A Prolog-style backtracking solver allows `Start` to be an unknown value and would attempt to answer queries using its depth-first backward chaining strategy. Unfortunately, Prolog's strategy would recurse forever due to line 324 even on the query `path("atlantis","baltimore")`. The reason is that `path("atlantis", "baltimore")` would recursively depend on `path("atlantis",Y)`, which would recurse to itself. Prolog essentially gets stuck and never evaluates `edge(Mid,End)`.

This example perfectly demonstrates the limitations of Datalog and Prolog, as the query `path("atlantis","baltimore")?` can be easily recognized by a human as the shortest path in a graph between two cities. Even the query `path("atlantis",Y)?` can be recognized as a single-source shortest path problem and can be solved using Dijkstra's algorithm [53]. A human programmer can also identify that the query `path(X,Y)?` is solvable using Bellman-Ford [19, 69] and augmenting the returned pairs of city distance with `path(Start,Start) min= 0`.

Our challenge with Dyna is to design a system that is able to handle the fact that `Start` is a variable while still avoiding the complications of cycles. Furthermore, we would like to recover the efficient strategies of Bellman-Ford or Dijkstra's when evaluating `path(X,Y)`.[67]

4.2.2 “Infinite” Neural Network

Our second example program was written on a whim for our 2017 paper [142] as a *natural* example Dyna program, and we did not even realize that it is a difficult program to execute until we analyzed this program at a later date. This program (lines 328 to 337), defines a two-dimensional convolutional neural network.

```
328 σ(X) = 1/(1+exp(-X)).              % define sigmoid function for all X
329 in(J) += out(I) * edge(I,J).                     % vector-matrix product
330 out(J) += σ(in(J)).                               % activation of node J
331 out(input[X,Y]) += pixel_brightness(X,Y).     % activation for input nodes
332 loss += (out(J) - target(J))**2.            % L2 loss of the predictions
333 edge(input[X,Y],hidden[X+DX,Y+DY]) = weight_conv(DX,DY).       % layer 1
334 edge(hidden[XX,YY],output[Prop])=weight_output(XX,YY,Prop).    % layer 2
335 weight_conv(DX,DY) := random(*,-1,1)                    % init with random
336         for range(DX,-4,4), range(DY,-4,4).
337 weight_output(XX,YY,Property) := random(*,-1,1).
```

The input to the neural network is `pixel_brightness(X,Y)` on line 331, and could be defined as a real number between 0 and 1, for example. Lines 328 to 330 define a feed-forward neural network where each neuron in the neural network is defined by a ground value. For example, we can have neurons named `input[12,34]`, `hidden[14,31]` or even `output["kitten"]`. The output activation, `out(J)`, of neuron ‘`J`’ is calculated as a sigmoid linear combination of the activations of other neurons

[67] One approach for solving the shortest path would be to wait until the value of `Start` is known, and then start forward chaining the values for the paths. This corresponds to using the memoization policy of
`$memo(path[Start:$ground,X:$free]) = "null".`
We can further implement Dijkstra’s by ordering the shortest distances first using
`$priority(path[Start,X]) = -path(Start,X).`

'`I`'. An interesting facet of this example is that lines 328 to 330 *does not* define the topology of the neural network. Instead, the topology is entirely determined by `edge(X,Y)`, which is defined on lines 333 to 337. Specifically, in this example, `hidden(XX,YY)` is activated by the 9×9 square of neurons centered at `input(XX,YY)`. Then, for each image property, `Prop`, all hidden units are pooled to activate the output unit, `output(Prop)`, whose activation represents the degree to which the image is predicted to have that property.

For a given finite image input, this program defines a neural network with finitely many edges. Nevertheless, it is difficult to solve this system of equations using standard strategies. The difficulty arises from line 333. Using a Datalog's forward-chaining strategy, the value for `weight_conv(2,-3)` would forward-chain through line 333, which establishes values for infinitely many edges of the form `edge(input(X,Y),hidden(XX,YY))` where $\texttt{XX} = \texttt{X} + 2$ and $\texttt{YY} = \texttt{Y} - 3$, despite the fact that only a finite number of edges are used for a given finite image. Conversely, using a Prolog style backwards-chaining with queries like `out(output["kitten"])` would lead to an internal query like `edge(I,hidden[XX,YY])` with free variables `XX` and `YY`. That query must return the entire infinite set of input-to-hidden edges, even though for a given finite image, only finitely many of these edges will touch input units that actually have values. This does not work.

To handle this program, we need to represent the infinite relation of `edge(X,Y)` without having to enumerate all of the edges with a lazy iterator or having to materialize all of the edges in a table.

4.2.3 “Infinitely” Many Dynabases

This was already mentioned in section §2.9.1, but Dyna’s object-oriented programming cannot directly adapt the existing techniques. The reason is that we can easily write a program where there is an *infinite* number of objects. For example, lines 338 to 341 defines a program that computes $\underset{x}{\operatorname{argmin}}\, x^2$.

```
338 a(X: float) = {
339   value = X**2.
340 }.
341 best_X = $arg((min= a(X).value arg X)).
```

Line 338 defines a dynabase to represent an input to the function getting optimized. In the “constructor” to the dynabase, it takes the current location where the function is being evaluated. Conceptually, line 338 defines an *infinite* number of dynabases, as it is defined for all *real* numbers.

If we have a system that *must* fully evaluate a subset of the program at a time, then this is impossible to solve. To handle this program, we must represent the *entire* program symbolically at the same time. In this case, we can recognize the dynabase as a layer of abstraction and can solve the expression $\operatorname{argmin}_x x^2$ using any number of optimization techniques.

4.2.4 “Infinite” Identity Matrix

Another common issue with Dyna programs is that the *details* that are useful for *running* the program are not included in the source code. Recall, at the start of this chapter, Dyna’s stated goal is that a program that is understandable by *humans* (not *computers*) should “*just work*”.

A small example of this is an “*infinite*” identity matrix:

```
342 identity_matrix = new matrix {
343   elem(X,Y) := 0.
344   elem(X,X) := 1.
345 }.
```

Here, we are defining that any element in the matrix such that both of its first and second argument are the same will have a value 1 (line 344), and 0 otherwise (line 343).

This matrix could be used with other matrices. For example, we can define a matrix multiplication on line 353:

```
346 matrix = new {
347   row(X) :- $self.elem(X,_).
348   rows += 1 for $self.row(_).
349   col(Y) :- $self.elem(_,Y).
350   cols += 1 for $self.col(_).
351 }.
352
353 multiply_matrices(A,B) = new matrix {
354   elem(I,K) += A.elem(I,J) * B.elem(J,K).
355 }.
356
357 result = multiply_matrices(identity_matrix, some_other_matrix).
```

This matrix representation is understandable by humans and even has some *nice* properties. For example, we are *not* required to use numerical integers as the identifies for the rows and columns. For example, we could have a row identified by the structured term `noun["kitten"]`, and the code continues to work.

However, this code has some difficulties when trying to design execution strategies. For starters, we do not know the number of rows and columns of the identity matrix. In this case, the programmer forgot to inform us that identity matrices are square matrices. As such, if the query '`result.rows?`' is attempted, we can not use any information available from the `some_other_matrix` on line 357 to compute this query. Now, as long as the Dyna program does not *need* to know the value of '`result.rows`', then this is not a problem.

4.3 A Common Theme

All of these hard examples that can work share a common theme. There exists a subset of the program that is insufficient to run it. A naive representation—such as Datalog's materialize everything or Prolog lazy bindings to variables—is insufficient when trying to "*greedily*" evaluate the program, one subset at a time. We need to be able to represent a subset of the program, even when it cannot be evaluated. Once there is enough information, there needs to be some approach to apply the relevant strategies to run the program.

We will develop the necessary techniques in chapters 5 and 6.

Chapter 5

Relational Expressions for Logic programming

In this chapter, I introduce our *relational expression* notation, which we shorten to **R**-exprs. This chapter is one of the central contributions of this dissertation and was originally introduced in Francis-Landau et al. [70]. **R**-exprs is our *internal representation*, and is used to represent and execute Dyna programs. **R**-exprs represent both functions—much like how bytecode is the *internal representation* in procedural programming languages—and bag relations—much like a database table. In essence, we will use **R**-exprs to represent both *code* and *data* (e.g. section §5.1.1).

As we will see throughout this dissertation, the homogeneity of **R**-exprs allows the Dyna runtime to be flexible, as yet-to-be-evaluated "*delayed*" code is stored alongside data. This turns out to be necessary to handle a number of tricky programs (chapter §4).

In the remainder of this chapter, I will define the semantic interpretation of **R**-exprs. In chapter §7, I will show how Dyna programs can be converted into the **R**-expr representation. Then in chapters 6 and 8, I will discuss how **R**-exprs are used to execute programs using term rewriting. In chapters 10 to 12, I will address some of the challenges of using a term rewriting system to implement a programming language.

5.1 Representing Programs Using Bags

Before discussing our bag-relational algebra in the next section, let us review bag algebras and look at a few high-level examples.

A **bag** is a generalization of a set with elements in the bag being counted one or more times. Bags are written using $\wr \cdots \int$. We refer to the number of times an element appears in a bag as **multiplicity**, and we will denote it using an at-sign @. As an example, a bag containing four, two-element tuples, where only two of which are distinct, looks like:

$$\wr \langle 1,2\rangle @1, \langle 3,4\rangle @3 \int \tag{5.1}$$

where $\langle 1,2\rangle$ is contained once, and $\langle 3,4\rangle$ is contained three times.

5.1.1 Bags of Named Tuples

When representing programs, we often have dozens of variables in an expression. Hence, the above notation with values in a tuple marked by their index becomes cumbersome. Instead, from this point forward, we will use bags over *named-tuples* with bindings to variables. For example:

$$\wr \langle \mathsf{X}=1, \mathsf{Y}=2\rangle @1, \langle \mathsf{X}=3, \mathsf{Y}=4\rangle @3 \int \tag{5.2}$$

is the same as the expression before, but we have now denoted the first variable as X and the second variable as Y.

For our purposes, we say that bags represent relations over *all* variables, not just the variables explicitly mentioned. Any variable that is not explicitly mentioned is considered to as taking on “any/all” values:

$$\wr \langle \mathsf{X}=1\rangle @1 \int \equiv \wr \langle \mathsf{X}=1, \mathsf{Y}\rangle @1 : \mathsf{Y} \in \mathcal{G} \int \tag{5.3}$$

At first, this might appear a bit awkward, however, this is useful when we are combining multiple bags together using intersection $\Cap$ and union $\uplus$. For example, consider the intersection between two bags with different variables:

$$\wr \langle \mathsf{X}=1\rangle @2 \int \Cap \wr \langle \mathsf{Y}=4\rangle @3 \int \equiv \wr \langle \mathsf{X}=1, \mathsf{Y}=4\rangle @6 \int \tag{5.4}$$

Observe, with bag intersection $\Cap$, the resulting bag is both the intersection of the bags and the product of the multiplicities. Note, we do not have a *min* multiplicity **R**-expr.[68] The reason is that we want to count the number of ways a tuple is contained in the bag. The counts of the number of times a tuple is in the bag is used by the aggregators (section §5.2.2.10).

[68] Min multiplicity is sometimes used by other bag algebras for intersection.

In example 5.4, the variables were different, so the intersection only appears as a tuple that contains the assignments to both X and Y. However, the power of bag intersection is that it checks the consistency between tuples contained in the bag. For example, in example 5.5, we have that the variable X appears in both bags that we are intersecting:

$$\wr \langle \mathsf{X} = 1 \rangle @7, \langle \mathsf{X} = 3 \rangle @5 \int \Cap \wr \langle \mathsf{X} = 3, \mathsf{Y} = 11 \rangle @2 \int \equiv \wr \langle \mathsf{X} = 3, \mathsf{Y} = 11 \rangle @10 \int \quad (5.5)$$

The assignment $\langle \mathsf{X} = 1 \rangle$ is incompatible with the second bag, hence it is eliminated by the bag intersection.

Bag union $\uplus$ behaves similarly to bag intersection. Tuples that are compatible with each other have their multiplicities added together. Incompatible tuples behave as independent elements, appearing as their own elements in the resulting bag.

$$\begin{aligned} &\wr \langle \mathsf{X} = 1 \rangle @1, \langle \mathsf{X} = 2 \rangle @2, \int \uplus \wr \langle \mathsf{X} = 1 \rangle @3, \langle \mathsf{Y} = 2 \rangle @9 \int \\ &\equiv \wr \langle \mathsf{X} = 1 \rangle @4, \langle \mathsf{X} = 2 \rangle @2, \langle \mathsf{Y} = 2 \rangle @9 \int \end{aligned} \quad (5.6)$$

5.1.2 Representations of Constraints in Bags

To make bags useful for representing computation, we extend our bag notation with a "bag-builder" notation. This bag-builder notation is conceptually similar to set-builder[69] notation where a set is defined using a boolean predicate. However, bags associate a positive *multiplicity* with each element in the bag instead of the boolean *true*. This means that our bag-builder notation will return a nonnegative integer which represents the number of times in which a named-tuple is contained in the bag.

As an example, let us define the bag that contains all *values* assigned to the variable X that are less than the number 5:

$$\wr \langle \mathtt{X} \rangle @c : c = \begin{cases} 1 & \textbf{if } \mathtt{X} < 5 \\ 0 & \textbf{otherwise} \end{cases} \int \tag{5.7}$$

Here, c is the variable that corresponds with the multiplicity of the tuple $\langle \mathtt{X} \rangle$ for some value of X. The multiplicity is computed using the expression $\begin{cases} 1 & \textbf{if } \mathtt{X} < 5 \\ 0 & \textbf{otherwise} \end{cases}$ which returns 1 if it is contained in the bag, and 0 otherwise.

As in section §5.1.1, we can intersect bags to create composite expressions. For example, we can represent bag of integers between $[0,5)$ by intersecting the

[69] For example, in set builder notation, the set of even integers could be written as $\{x \in \mathbb{Z} : \exists y \in \mathbb{Z}, y * 2 = x\}$, where the boolean predicate $\exists y \in \mathbb{Z}, y * 2 = x$ is only true for even integers. Note, when creating sets using an enumerable expression (iterable) in a program, this is called *set comprehension*. However, in mathematics (as we are doing here) this is called set-builder notation. The predicate that defines the set is not required to be computable. For example, we can use set-builder notation to define the set of all people you have met and *will ever* meet in your life. This set is well defined, but not something that we can express or construct in an executable program without also inventing a time machine.

constraints for $\mathtt{X} \in \mathbb{Z}, \mathtt{X} \geq 0, \mathtt{X} < 5$:

$$
\begin{aligned}
&\Lbag \langle \mathtt{X} \rangle @ c : c = \begin{cases} 1 & \textbf{if } \mathtt{X} \in \mathbb{Z} \\ 0 & \textbf{otherwise} \end{cases} \Rbag \\
\bowtie &\Lbag \langle \mathtt{X} \rangle @ c : c = \begin{cases} 1 & \textbf{if } \mathtt{X} \geq 0 \\ 0 & \textbf{otherwise} \end{cases} \Rbag \\
\bowtie &\Lbag \langle \mathtt{X} \rangle @ c : c = \begin{cases} 1 & \textbf{if } \mathtt{X} < 5 \\ 0 & \textbf{otherwise} \end{cases} \Rbag
\end{aligned}
\tag{5.8}
$$

5.1.3 A First Step Towards Computation with Bags

Equation (5.8) above has 3 different constraints represented with bag-builder notation. This expression is perfectly acceptable and is something that we are capable of handling and representing in our Dyna implementation. We may even return an expression like this to the user via the REPL in some cases.

However, our goal is to use expressions like example 5.8 to perform computation. This means that we need some way to manipulate expressions like example 5.8, and our manipulations should correspond with computation. The way we accomplish this is to *rewrite* an expression into a "*simpler*", *semantically equivalent* expression. I will formally define our rewrites in chapter §6, and what simpler means in chapter §15.

In the case of eq. (5.8), it can be rewritten as a bag which enumerates the possible values of $\texttt{X}$:

$$\left(\begin{array}{r}\lbag \langle \texttt{X} \rangle @ c : c = \begin{cases} 1 & \textbf{if } \texttt{X} \in \mathbb{Z} \\ 0 & \textbf{otherwise} \end{cases} \rbag \\ \mathbin{\times\!\!\!\cap} \lbag \langle \texttt{X} \rangle @ c : c = \begin{cases} 1 & \textbf{if } \texttt{X} \geq 0 \\ 0 & \textbf{otherwise} \end{cases} \rbag \\ \mathbin{\times\!\!\!\cap} \lbag \langle \texttt{X} \rangle @ c : c = \begin{cases} 1 & \textbf{if } \texttt{X} < 5 \\ 0 & \textbf{otherwise} \end{cases} \rbag \end{array}\right) \to \left\lbag \begin{array}{l} \langle \texttt{X} = 0 \rangle @ 1, \\ \langle \texttt{X} = 1 \rangle @ 1, \\ \langle \texttt{X} = 2 \rangle @ 1, \\ \langle \texttt{X} = 3 \rangle @ 1, \\ \langle \texttt{X} = 4 \rangle @ 1 \end{array} \right\rbag \tag{5.9}$$

Here, we have *rewritten* from example 5.8 into example 5.9. In this example, and throughout this dissertation, rewrites are denoted using the right arrow $\to$. I claim that the rewritten bag in example 5.9 is *simpler* than example 5.8, as we can directly read the values $0, 1, 2, 3, 4$ (the integers between $[0, 5)$) from the bag.[70] For now, an intuitive definition of simpler is that a simpler **R**-expr requires less computation to get the "final" answer. For example, the value -1 is simpler than $e^{i\pi}$, which requires evaluating the exponential to determine the value of $e^{i\pi}$. I will give a formal definition of *simpler* in section §15.3.

5.1.4 A More Convenient Notation

The notation that I have used so far is admittedly very verbose and tedious to write. As such, we have designed a much more concise notation for **R**-exprs.

First, observe that all bag expressions are over *all* variables. Hence, naming variables in the $\langle \cdots \rangle$ tuple is redundant. Second, we will say that all **R**-exprs

[70] Even if represented the set of integers between $[0, 10^{100})$ as a bag of 10^{100} integers, we still consider the bag of 10^{100} integers as *simpler* despite its enormous size, as it requires less "computation". This is explained in detail in section §15.3.

"return" a multiplicity. Hence, it is unnecessary to write the **if**-case expression as in examples 5.7 and 5.8. Third, we are going to represent operations using term names instead of mathematical symbols. This means that an expression like $X \in \mathbb{Z}$ is instead written as `int(X)`. Additionally, we will write $X \geq 0$ as `lessthaneq(0,X)` and $X < 5$ as `lessthan(X,5)`. Variables, such as `X`, follow the same convention as Dyna code, they start with a capital letter and are colored Green in this dissertation. Finally, for intersection and union, we will write `*` and `+` in monospaced typewriter font instead of $\Cap$ and $\uplus$.

All together, this will allow us to write example 5.8 as the **R**-expr:

```
int(X)*lessthaneq(0,X)*lessthan(X,5)
```

5.2 Semantics of R-exprs

As noted in section §5.1.3, when rewriting an **R**-expr to perform "computation", we are rewriting the **R**-expr into another simpler **R**-expr which is semantically equivalent. As such, we need to define semantic equivalence for **R**-exprs.

5.2.1 Ground Values

Let us first start by defining the set of **ground terms** $\mathcal{G}$ built from a ranked set of functors $\mathcal{F}$ and primitive values such as integers, floats, and strings. For example, $\mathcal{G}$ includes 123, 3.14, `"hello world"` and structured terms like `foo[123, "hello",` `bar[4,5,6]]` (as introduced in section §2.1.1).

We further define $\mathcal{M} = \mathbb{N} \cup \{\infty\}$ as the set of multiplicities. Therefore, a bag containing an n-arity tuple of ground terms ($\mathcal{G}^n$), can be represented as a function that map from $\mathcal{G}^n$ to the multiplicity `M` $\in \mathcal{M}$ of that tuple contained in the bag: $\mathcal{G}^n \mapsto \mathcal{M}$.

That said, we do not use the integer position in an n-arity tuple as it is inconvenient. Instead, we use a map from variable names to ground terms. Let $\tilde{\mathcal{V}}$ be an infinite set of unique **variable** names. A **environment** $E(\cdot)$ is, a function that maps from a finite set of variables to ground terms. More specifically, for any $\mathcal{U} \subset \tilde{\mathcal{V}}$, the $\mathcal{U}$-**tuples** are maps $E(\cdot) : \mathcal{U} \mapsto \mathcal{G}$. Using this representation, a bag over $\mathcal{U}$-tuples can be represented by a function $F(\cdot)$ where $F(\cdot) : (\mathcal{U} \mapsto \mathcal{G}) \mapsto \mathcal{M}$.[71]

For convenience, we define the set of **value types** as $\mathcal{V} = \tilde{\mathcal{V}} \cup \mathcal{G}$. We extend the $E(\cdot)$ function to $\mathcal{V}$ by mapping any ground value $g \in \mathcal{G}$ to itself. With this, we can think of the $E(\cdot)$ function as a kind of "get value" function.[72]

5.2.2 Inductive Definition of **R**-exprs Semantics

In the remainder of this section, I will give the definitions of **R**-exprs and their semantics. **R**-exprs are terms that are defined recursively in terms of sub-**R**-exprs

[71]For this chapter, $E(\cdot)$ will only be used when it contains the necessary variables. Hence, we do not define what happens if a variable is not contained in $E(\cdot)$. In chapter §8, when discussing the implementation of **R**-exprs and the rewrite rules, I will discuss what happens when a variable is not contained in $E(\cdot)$.

[72]Note that the variables and ground terms are disjoint ($\mathcal{G} \cap \tilde{\mathcal{V}} = \emptyset$).

1. $[\![\texttt{U=V}]\!]_E = $ **if** $E(\texttt{U}) = E(\texttt{V})$ **then** 1 **else** 0, where $\texttt{U},\texttt{V} \in \mathcal{V}$

2. $[\![\texttt{T=f(X}_1,\cdots,\texttt{X}_n\texttt{)}]\!]_E = $ **if** $E(\texttt{T}) = f[E(\texttt{X}_1),\cdots,E(\texttt{X}_n)]$ **then** 1 **else** 0,
where $\texttt{T},\texttt{X}_1,\cdots,\texttt{X}_n \in \mathcal{V}$ and $\texttt{f} \in \mathcal{F}$

3. $[\![\texttt{plus(I,J,K)}]\!]_E = $ **if** $E(\texttt{I}) + E(\texttt{J}) = E(\texttt{K})$ **then** 1 **else** 0, where $\texttt{I},\texttt{J},\texttt{K} \in \mathcal{V}$

4. $[\![\texttt{R+S}]\!]_E = [\![\texttt{R}]\!]_E + [\![\texttt{S}]\!]_E$, where $\texttt{R},\texttt{S} \in \mathcal{R}$

5. $[\![\texttt{R*S}]\!]_E = [\![\texttt{R}]\!]_E \cdot [\![\texttt{S}]\!]_E$, where $\texttt{R},\texttt{S} \in \mathcal{R}$

6. $[\![\texttt{M}]\!]_E = \texttt{M}$, where $\texttt{M} \in \mathcal{M}$

7. $[\![\texttt{if(Q,R,S)}]\!]_E = $ **if** $[\![\texttt{Q}]\!]_E > 0$ **then** $[\![\texttt{R}]\!]_E$ **else** $[\![\texttt{S}]\!]_E$, where $\texttt{Q},\texttt{R},\texttt{S} \in \mathcal{R}$

8. $[\![\texttt{proj(X,R)}]\!]_E = \sum_{x\in\mathcal{G}} [\![\texttt{R}]\!]_{E[\texttt{X}=x]}$, where $\texttt{X} \in \mathcal{V}, \texttt{R} \in \mathcal{R}$

9. $[\![\texttt{A=sum(X,R)}]\!]_E = $ **if** $E(\texttt{A}) = \sum_{x\in\mathcal{G}}(x * [\![\texttt{R}]\!]_{E[\texttt{X}=x]})$ **then** 1 **else** 0,
where $\texttt{A},\texttt{X} \in \mathcal{V}, \texttt{R} \in \mathcal{R}$.

10. $[\![\texttt{f(T}_1,\ldots,\texttt{T}_n\texttt{)}]\!]_E = [\![\texttt{R}_\texttt{f}\{\texttt{X}_1 \mapsto \texttt{T}_1,\ldots,\texttt{X}_n \mapsto \texttt{T}_n\}]\!]_E$, where $\texttt{T}_1,\ldots,\texttt{T}_n \in \mathcal{V}$

Figure 5-1. Semantic definitions of all **R**-expr kinds. The semantics $[\![\texttt{R}]\!]_E$ of an **R**-expr R is defined as the multiplicity ($\mathcal{M} = \mathbb{N} \cup \{\infty\}$) of E in the bag defined by R. E is a tuple of named variables and their ground assignments (section §5.1.1). Additional details about the **R**-exprs' definitions are provided in section §5.2.2.

and value types $\mathcal{V}$. As we will see shortly, the semantics of **R**-exprs is defined as a system of equations where the semantic interpretation each individual **R**-expr is defined as a *multiplicity* given environment $E(\cdot)$.

To make this more formal, let $\mathcal{R}$ denote the set of all **R**-exprs.

Each **R**-expr $\mathsf{R} \in \mathcal{R}$ has a finite set of *free variables* denoted by $\text{vars}(\mathsf{R})$. A **free variable** is a variable in an **R**-expr that is not bound by another **R**-expr. Note, $\text{vars}(\mathsf{R}) \subseteq \tilde{\mathcal{V}}$ as $\text{vars}(\mathsf{R})$ returns a set of named *variables*, not *value types*.

Next, for each environment E, the **denotation function** $[\![\cdot]\!]_E : \mathcal{R} \mapsto \mathcal{M}$ interprets an **R**-expr in the environment given by the named tuple E. It defines a multiplicity $[\![\mathsf{R}]\!]_E$ for any **R**-expr R whose $\text{vars}(\mathsf{R}) \subseteq \text{domain}(E)$.[73]

We will now define and explain $[\![\mathsf{R}]\!]_E$ and $\text{vars}(\mathsf{R})$ for the different kinds of **R**-exprs in our system, which are also shown in figure 5-1.

5.2.2.1 Equality Constraints

First, we define **equality constraints** between two value types $\mathsf{U}, \mathsf{V} \in \mathcal{V}$. An equality constraint is *true* in an environment when the ground value assigned to U and V is equal. This is represented using multiplicity 1, which indicates that the current environment is consistent with this **R**-expr and is therefore contained in the bag represented by this **R**-expr. False is represented with multiplicity 0.[74]

[73] $[\![\cdot]\!]_E$ is undefined in the case that $\text{vars}(\mathsf{R}) \not\subseteq \text{domain}(E)$.

[74] Recall that a multiplicity is defined as $\mathcal{M} = \mathbb{N} \cup \{\infty\}$. Hence, 0 and 1 are the identity and annihilator elements of the multiplicities.

1. $[\![\texttt{U=V}]\!]_E = $ **if** $E(\texttt{U}) = E(\texttt{V})$ **then** 1 **else** 0, where $\texttt{U}, \texttt{V} \in \mathcal{V}$

Notice that we did not write $[\![\texttt{U}]\!]_E = [\![\texttt{V}]\!]_E$ but $E(\texttt{U}) = E(\texttt{V})$—our denotation function $[\![\cdot]\!]_E$ maps **R**-exprs to multiplicities, and $E(\cdot)$ maps variables and ground terms in $\mathcal{V}$ to ground terms in $\mathcal{G}$.

5.2.2.2 Structured Term Equality Constraints

Next, we have **equality constraints with a structured term** with $\texttt{T}, \texttt{X}_1, \cdots, \texttt{X}_n \in \mathcal{V}$ and $\texttt{f} \in \mathcal{F}$ being the name of the structured term (previously introduced in section §2.1.1).

2. $[\![\texttt{T=f[X}_1, \cdots, \texttt{X}_n\texttt{]}]\!]_E = $ **if** $E(\texttt{T}) = \texttt{f}[E(\texttt{X}_1), \cdots, E(\texttt{X}_n)]$ **then** 1 **else** 0,

   where $\texttt{T}, \texttt{X}_1, \cdots, \texttt{X}_n \in \mathcal{V}$ and $\texttt{f} \in \mathcal{F}$

5.2.2.3 Builtin R-exprs Constraints

We also have a number of **built-in R**-expr kinds. These **R**-exprs correspond to "low-level" operations on primitive ground terms, such as addition between numbers or concatenating strings. Built-ins are defined as `1` when the assignment to their arguments is *consistent* with their definition. For example, the built-in addition, which henceforth will be referred to as `plus` for clarity, is defined as follows:

3. $[\![\texttt{plus(I,J,K)}]\!]_E = $ **if** $E(\texttt{I}) + E(\texttt{J}) = E(\texttt{K})$ **then** 1 **else** 0, where $\texttt{I}, \texttt{J}, \texttt{K} \in \mathcal{V}$

Note, in this definition, the $+$ sign corresponds with addition between the values of $E(\texttt{I})$ and $E(\texttt{J})$, which are both in the $\mathcal{G}$ that is the Herbrand universe of values of the *Dyna* language.

5.2.2.4 Constraints

The **R**-exprs we have so far are constraints. A **constraint** is any **R**-expr that has a multiplicity of 0 or 1 in any environment. It is useful to track constraints as they can be duplicated as needed or removed when they are redundant.[75]

5.2.2.5 Disjunctions

Disjunctions are the union of two or more **R**-exprs, and are denoted using +.

4. $[\![\mathsf{R}\texttt{+}\mathsf{S}]\!]_E = [\![\mathsf{R}]\!]_E + [\![\mathsf{S}]\!]_E$, where $\mathsf{R}, \mathsf{S} \in \mathcal{R}$

Disjuncts correspond with the $\uplus$ as we saw before in section §5.1.1. Disjuncts are usually not constraints as their multiplicity is often greater than 1.[76] This time, the $+$ sign on the right-hand side of this definition is the addition between the *multiplicities* returned by $[\![\mathsf{R}]\!]_E$ and $[\![\mathsf{S}]\!]_E$. The typewriter font plus sign '+' in the semantic brackets is only used to represent disjunctions, as done here.

The free variables in a disjunction are the union of the free variables in disjunctive sub-**R**-exprs: $\text{vars}(\mathsf{R}\texttt{+}\mathsf{S}) = \text{vars}(\mathsf{R}) \cup \text{vars}(\mathsf{S})$. This is in contrast to the earlier **R**-expr kinds where the free variables are simply all of the variables that appeared in the

[75]This is allowed as this does not change the multiplicity of the **R**-expr. e.g. $1 * 1 * 1 = 1$ or $0 * 0 * 0 = 0$

[76]If R and S are both constraints and non-overlapping, then the disjunction R+S is also a constraint.

R-expr.

5.2.2.6 Conjunctions

Conjunctions are the intersection of two or more **R**-exprs, and are denoted using `*`.

5. $[\![\mathtt{R*S}]\!]_E = [\![\mathtt{R}]\!]_E \times [\![\mathtt{S}]\!]_E$, where $\mathtt{R}, \mathtt{S} \in \mathcal{R}$

Conjuncts correspond with $\bowtie$ from before. Conjunctions are often constraints themselves as if both conjuncts `R` and `S` are a constraint, then a conjunction is also a constraint. The typewriter font multiplication symbol '`*`' is only used to represent conjunctions. In the written presentation in this dissertation, I will follow the standard order-of-operations with '`+`' and '`*`', with '`*`' binding more tightly than '`+`'. Further note that we define all conjunctions with a `0` as `0`, even in the case of an ∞ multiplicity. Hence $0 \times \infty = 0$.

Just like with disjunctions, the free variables are the union of the free variables in the conjunctive sub-**R**-exprs: $\text{vars}(\mathtt{R*S}) = \text{vars}(\mathtt{R}) \cup \text{vars}(\mathtt{S})$.

5.2.2.7 Multiplicities

Multiplicities **R**-expr kinds are defined as the multiplicity they represent.

6. $[\![\mathtt{M}]\!]_E = \mathtt{M}$, where $\mathtt{M} \in \mathcal{M}$

Multiplicities always return the same value `M`, and are not influenced by the environment E. For example, the multiplicity `1` represents the bag that contains all environments exactly once, and `0` represents an empty bag. E.g. $[\![\texttt{1}]\!]_E = 1$ and $[\![\texttt{0}]\!]_E = 0$.

5.2.2.8 Conditionals

We allow conditional or `if`-expressions in our relational algebra as follows:

7. $[\![\texttt{if(Q,R,S)}]\!]_E = \textbf{ if } [\![\texttt{Q}]\!]_E > 0 \textbf{ then } [\![\texttt{R}]\!]_E \textbf{ else } [\![\texttt{S}]\!]_E, \qquad \text{where } \texttt{Q},\texttt{R},\texttt{S} \in \mathcal{R}$

Notice that this construction can be used to implement anti-joins, set differences, and priority unions on sets. It does not support bag differences, which would involve subtracting multiplicities, but instead uses `Q` to choose between two multiplicities (which can be `0`). Furthermore, an `if`-expression is a constraint if `R` and `S` are both constraints.[77]

As with disjunctions and conjunctions, the free variables are the union of the free variables from the sub-**R**-exprs: $\text{vars}(\texttt{if(Q,R,S)}) = \text{vars}(\texttt{Q}) \cup \text{vars}(\texttt{R}) \cup \text{vars}(\texttt{S})$.

5.2.2.9 Projection

Next, we define **projection**, which removes a named column (variable) from a bag relation, summing the multiplicities of tuples that have become equal. When translating a Dyna program into an **R**-expr (chapter §7), projections will be used to eliminate a rule's local variables.

[77] This fact will be used for some advanced rewrite rules discussed in chapter §9.

8. $[\![\texttt{proj(X,R)}]\!]_E = \sum_{x\in\mathcal{G}} [\![\texttt{R}]\!]_{E[\texttt{X}=x]}$, where $\texttt{X} \in \mathcal{V}, \texttt{R} \in \mathcal{R}$, and where $E[\texttt{X} = x]$ means a version of E that has been modified such that $E(\texttt{X}) = x$,[78] [79] and $\sum_{x\in\mathcal{G}}$ is a summation of the multiplicities over all possible values in $\mathcal{G}$.[80]

In the case that the variable $\texttt{X}$ is not contained in $\text{vars}(\texttt{R})$, and the multiplicity of $\texttt{R}$ is non-zero ($[\![\texttt{R}]\!]_E > 0$), then the multiplicity of projection is defined as infinity (∞).[81] The reason is that this is equivalent to summing a non-zero value an infinite number of times because $\mathcal{G}$ is an infinite set.

The free variables of projection are again based on the sub-**R**-expr's free variables, but this time, we are removing the variable $\texttt{X}$ from the set of free variables: $\text{vars}(\texttt{proj(X,R)}) = \text{vars}(\texttt{R}) - \{\texttt{X}\}$.

5.2.2.10 Aggregation

Aggregation does the same as projection, but instead of adding multiplicities, it *aggregates* the $\texttt{X}$ value and computes a new value $\texttt{Y}$ as the result of aggregation. Thus, it removes column $\texttt{X}$ from the bag but introduces a new column $\texttt{Y}$.

Throughout this dissertation, I will use the 'sum' aggregator as a prototypical

[78] If $\texttt{X}$ is already contained in $E(\cdot)$, then it will be overridden, otherwise it is added as a new variable.

[79] Recall that $\mathcal{V} = \tilde{\mathcal{V}} \cup \mathcal{G}$, hence it is possible that $\texttt{X} \in \mathcal{G}$ and a ground value. This case is "allowed" though not particularly interesting. We have $[\![\texttt{proj(g,R)}]\!]_E = [\![\texttt{R}]\!]_E$ as there is only one value $g \in \mathcal{G}$ that is equal to g.

[80] Usually $[\![\texttt{R}]\!]_{E[\texttt{X}=x]}$ will have a non-zero multiplicity for a finite number of values of $x \in \mathcal{G}$. Hence, this sum over $\mathcal{G}$ (which is infinite) can be computed by identifying the relevant subset of $\mathcal{G}$. This will be discussed extensively in the following chapters.

[81] For example, $[\![\texttt{proj(X,1)}]\!]_E = \infty$, as this is equivalent to $\sum_{x\in\mathcal{G}} 1$, where $|\mathcal{G}| = \infty$.

aggregator. However, there are other aggregators which have similar semantic denotations and rewrite rules.

9. $[\![\texttt{A=sum(X,R)}]\!]_E =$ **if** $E(\texttt{A}) = \sum_{x \in \mathcal{G}} (x * [\![\texttt{R}]\!]_{E[\texttt{X}=x]})$ **then** 1 **else** 0,

   where $\texttt{A}, \texttt{X} \in \mathcal{V}, \texttt{R} \in \mathcal{R}$. The $*$ in the summand means that the summation includes $[\![\texttt{R}]\!]_{E[\texttt{X}=x]}$ copies of the value x (possibly ∞ copies[82]). If there are no summands (meaning $[\![\texttt{R}]\!]_E = 0$ for all E), then the result of $\sum \cdots$ is defined as `null`, which is a value outside of $\mathcal{G}$. Hence, it has the multiplicity of `0`. In other words, when `R` is `0`, we have $[\![\texttt{A=sum(X,0)}]\!]_E = 0$. If there are non-numeric summands, or the sum is not well defined,[83] then the result of the sum is defined to be an error value in $\mathcal{G}$, such as `NaN`.

Notice that the aggregator **R**-expr is a constraint as its multiplicity is either `1` or `0`. Additionally, observe we do not define $[\![\texttt{sum(X,R)}]\!]_E$, whose value would not be a multiplicity. Rather, the aggregator is written as `A=sum(X,R)`, which denotes where the result of aggregation will be saved (variable `A` in this case) and is considered a more readable notation for what could have been written as `sum(A,X,R)`.

The free variables of aggregation project out the input variable `X` and add the variable `A` to the free variables: $\text{vars}(\texttt{A=sum(X,R)}) = (\text{vars}(\texttt{R}) - \{\texttt{X}\}) \cup \{\texttt{A}\}$

[82]Some aggregators such as `min` and `max` do not care about the *number* of copies, as long as it is more than one. Aggregators such as `sum` can sum an infinite number of zeros and still have the value zero, e.g. $0 = \sum_{i=0}^{\infty} 0$.

[83]For example, the sum $\sum_{i=0}^{\infty} (-1)^i$ does not converge and therefore not well defined.

5.2.2.11 User-defined **R**-exprs

Finally, it is convenient to augment the built-in relations with **user-defined** relation kinds. Choose an identifier $\mathtt{f} \in \mathcal{F}$ and choose some **R**-expr $\mathtt{R}_{\mathtt{f}}$ with $\text{vars}(\mathtt{R}_{\mathtt{f}}) \subseteq \{\mathtt{X}_1, \ldots, \mathtt{X}_n\}$ (which are n distinctly named variables) to serve as the **definition** or **R**-*expr expansion* of f. Now define

10. $[\![\mathtt{f}(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!]_E = [\![\mathtt{R}_{\mathtt{f}}\{\mathtt{X}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{X}_n \mapsto \mathtt{T}_n\}]\!]_E, \quad \text{where } \mathtt{T}_1, \ldots, \mathtt{T}_n \in \mathcal{V}$

    The $\{\mapsto\}$ notation denotes substitution for variables, where free variables of $\mathtt{R}_{\mathtt{f}}$ are renamed, and variables captured by other **R**-exprs, such as projection or aggregation, are first renamed to avoid accidental capture.[84]

User-defined **R**-exprs make it possible to define **R**-exprs which are circularly defined in terms of themselves.[85] In fact, a Dyna program will normally do this. In this case, the definition of $[\![\cdot]\!]_E$ is not an inductive function. Rather, $[\![\cdot]\!]_E$ is interpreted as a variable in a system of equations, where we solve for values of the $[\![\cdot]\!]_E$ that satisfies these constraints. This is allowed by Dyna, as in section §2.5, I stated that the semantics of Dyna allow us to find *any* consistent assignment. In practice, there can be multiple possible assignments, and there is no guarantee that the system will find a particular assignment.[86]

[84] This is equivalent to α-renaming in λ-calculus.

[85] This is similar to a `let rec` construction in a functional programming language.

[86] Dyna does not guarantee which assignment will be found. This is in contrast to other logic programming languages, such as Datalog (section §3.1.2), that guarantee a minimal assignment is found.

5.3 Example **R**-exprs

Given that we have now defined the basic **R**-expr kinds, let us look at some simple **R**-exprs to see how they can represent bags and simple Dyna programs. This is not intended as a complete introduction to how **R**-exprs can represent Dyna programs or bag relations, as we will see more in chapter §7.

5.3.1 Finite Materialized Relation

As a first example, we can express a finite bag relation shown in figure 5-2a as an **R**-expr using conjunctions, disjunctions, and equality assignments as in figure 5-2b.

$$\left\{\begin{array}{l}\langle \mathtt{X}=1,\mathtt{Y}=1\rangle @1\\ \langle \mathtt{X}=2,\mathtt{Y}=6\rangle @1\\ \langle \mathtt{X}=2,\mathtt{Y}=7\rangle @2\\ \langle \mathtt{X}=5,\mathtt{Y}=7\rangle @1\end{array}\right\}$$

(a) Bag

```
(X=1)*(Y=2)+
(X=2)*(Y=6)+
(X=2)*(Y=7)*2+
(X=5)*(Y=7)
```

(b) R-expr

Figure 5-2. Bag of ground assignments to the variables `X` and `Y`. The `(X=1)` **R**-exprs are each individual **R**-expr equality constraints (as previously defined, section §5.2.2.1). The equality constraints are combined into conjunctions using '$*$' (section §5.2.2.6). Finally, the conjunctions are combined into a big disjunction using '+' (section §5.2.2.5).

Each individual tuple of the bag translates into a product ($*$) of several (`Variable` $=$ `value`) expressions. Encoding the multiple tuples contained in the bag is done by simply adding (+) the **R**-exprs for different tuples together using a disjunction.

R-exprs of this form can be used as the basis for user-defined **R**-exprs when representing external data as an **R**-expr (section §5.2.2.11). Finite materialized **R**-exprs are also desirable representations to return to the user when making a query, as they can be easily understood by reading the relevant values out of the **R**-expr (section §2.3).

5.3.2 Bag with Constraints

$$\Big\wr \langle \texttt{X} \rangle @ c : c = \begin{cases} 1 & \textbf{if } \exists y \in \mathbb{Z} \text{ s.t. } 2 * y = \texttt{X} \\ 0 & \textbf{otherwise} \end{cases} \Big\int$$

(a) Bag

```
if(proj(Y,int(Y)*times(2,Y,X)),1,0)
```

(b) **R**-expr

Figure 5-3. Bag of even integers.

The bag expression in figure 5-3a checks whether there exists an integer y such that $2 * y = \texttt{X}$. In figure 5-3b we introduce a new variable `Y` using `proj(Y,·)`. We ensure that the multiplicity of the **R**-expr is either `0` or `1` by using the `if`-expression to match the $\exists$ in the bag definition. Otherwise, the projection could have a multiplicity greater than `1`, though this does not actually happen in this case.

5.3.3 Simple Dyna Rule

As a preview for chapters 6 and 7, figure 5-4 shows a single-rule Dyna program and the equivalent **R**-expr representation.

```
358 | a(X) += (X+7)*X.
```

(a) Dyna

```
a(X,Result) →¹ (Result=sum(Inp,
  proj(Tmp,plus(X,7,Tmp)*times(Tmp,X,Inp))))
```

(b) R-expr

Figure 5-4. Simple Dyna program written as an **R**-expr.

Dyna user-defined terms, which are comprised of rules such as line 358, define their own user-defined **R**-expr kind (section §5.2.2.11) and rewrite rule (shown in figure 5-4b).

The user-defined **R**-expr has the variable `X` for the arguments to the expression, as well as a `Result` variable for the returned value of the expression. A rewrite rule is defined to rewrite the user-defined **R**-expr to its corresponding definition. Rewrite rules are represented as an arrow ($\rightarrow$), with important rewrite rules having a reference number over the arrow that will appear in the text and are clickable (e.g. rewrite rule 1).[87]

Calls to built-ins, written as '+' and '*' in the Dyna source on line 358 are converted into the **R**-exprs `plus(·,·,·)` and `times(·,·,·)`. We reserve the '*' and '+' symbols in the **R**-expr representation for conjunction and disjunction, which are used much more frequently.

All local variables and intermediate variables are projected out in user-defined **R**-exprs. All free variables must appear in the user-defined **R**-exprs. Intermediate

[87] Some PDF viewers have a *link preview* feature when hovering over links. I strongly suggest that you turn that feature on.

variables are introduced as needed when translating from Dyna into **R**-exprs, as exemplified by the variable `Tmp` in this example.

5.4 Conclusion of R-expr Semantics

The **R**-expr kinds defined in this chapter form the basis for our internal representation. In the next three chapters, I will define semantic preserving rewrite rules for **R**-exprs (chapter §6), as well as show in further detail how Dyna is translated into **R**-exprs (chapter §7) and how we can implement a simple term rewriting system using **R**-exprs (chapter §8).

Chapter 6

Rewrites Rules for **R**-exprs

The previous chapter gave the denotational semantics for **R**-exprs. I will now provide the operational semantics. The basic idea is that we can use rewrite rules to *simplify*, an **R**-expr until it is either a finite materialized relation—a list of tuples—or has some other convenient form. All of our rewrite rules are semantics-preserving by construction, so our rewrite system is sound, although it is not and cannot be complete, as chapter §15 will discuss. Our rewrites can be applied to any sub-**R**-expr contained within a larger **R**-expr. Chapter §8 will discuss the procedure we use to apply rewrites. Our rewrite system is *not* confluence, meaning that the order in which rewrites are applied can result in different semantically equivalent **R**-exprs. However, our rewrite system is *normalizing*, in that it eventually reach a state where no more directional rewrites that can be applied.

6.1 Equality Constraints and Multiplicity

We start with rewrites for basic equality constraints.

$(\mathtt{X}=\mathtt{X}) \xrightarrow{2} \mathtt{1}$ ▷*trivially true when the variables/values are the same*

$(\mathtt{X}=\mathtt{Y}) \xrightarrow{3} \mathtt{0}$ **if** $\mathtt{X},\mathtt{Y}\in \mathcal{G}$ **and** $\mathtt{X}\neq\mathtt{Y}$ ▷*values not equal*

$(\mathtt{G}=\mathtt{X}) \xrightarrow{4} (\mathtt{X}=\mathtt{G})$ **if** $(\mathtt{G}\in \mathcal{G}$ **and** $\mathtt{X}\in \tilde{\mathcal{V}})$ **or** $(\mathtt{G}\prec\mathtt{X})$ ▷*canonical order*

$(\mathtt{X}=\mathtt{Y})*\mathtt{R} \xrightarrow{5} (\mathtt{X}=\mathtt{Y})*\mathtt{R}\{\mathtt{X}\mapsto\mathtt{Y}\}$ **if** $\mathtt{X}\in \tilde{\mathcal{V}}$ ▷*equality propagation*

where $\mathtt{R}\{\mathtt{X}\mapsto\mathtt{Y}\}$ denotes replacing the variable X with the value type Y `in the` **R**`-expr R`

First, rewrite rule 2 checks for equivalence between its two arguments. When a **R**-expr constraint has been successfully and completely "checked", it is rewritten as a `1`, as in this case. Conversely, rewrite rule 3 handles the case where `X` and `Y` are not equal. It requires that both `X` and `Y` be known ground values. When both values are known, we can use the equality operator defined on ground values $\mathcal{G}$ to check if `X` and `Y` are equal. Constraints whose check fails are rewritten as `0`, which indicates that the current assignment of variables in the environment $E(\cdot)$ is *inconsistent* and, therefore, *not* contained in the bag represented by the **R**-expr.

For equality, we prefer to have a canonical ordering where we place any ground value as the second argument. If we have two variables, rewrite rule 4 reorders the equality constraint according to an arbitrarily chosen $\prec$ ordering on variables.[88]

The most common operation with equality constraints is performed by rewrite rule 5, which propagates a variable binding to other conjoined **R**-exprs. The `Y` is

[88] We do not care which $\prec$ function is used, as long as it is consistent. An example would be variable string name comparisons.

likely a ground value, in which case propagating the value of Y in place of X enables other rewrites, as we shall see.

6.1.1 Structured Term Equality Rewrites

Structured terms are similar to equality constraints; however, they involve multiple variables and a named term $\texttt{f},\texttt{g} \in \mathcal{F}$.

$(\texttt{X=f[Y}_1\texttt{,...,X,...,Y}_n\texttt{]}) \xrightarrow{6} \texttt{0}$ ▷*occurs check, not true for any ground value of* X

$(\texttt{X=f[Y}_1\texttt{,...,Y}_n\texttt{]})*(\texttt{X=f[Z}_1\texttt{,...,Z}_n\texttt{]}) \xrightarrow{7} (\texttt{X=f[Y}_1\texttt{,...,Y}_n\texttt{]})*(\texttt{Y}_1\texttt{=Z}_1)*\cdots*(\texttt{Y}_n\texttt{=Z}_n)$

$(\texttt{X=f[Y}_1\texttt{,...,Y}_n\texttt{]})*(\texttt{X=g[Z}_1\texttt{,...,Z}_m\texttt{]}) \xrightarrow{8} \texttt{0}$ **if** $(\texttt{f} \neq \texttt{g}$ **or** $n \neq m)$

$(\texttt{G=f[Y}_1\texttt{,...,Y}_n\texttt{]}) \xrightarrow{9} (\texttt{Y}_1\texttt{=Z}_1)*\cdots*(\texttt{Y}_n\texttt{=Z}_n)$ **if** $\texttt{G}\in\mathcal{G}$ **and** $\texttt{G=f[Z}_1\texttt{,...,Z}_n\texttt{]}$

$(\texttt{G=f[Y}_1\texttt{,...,Y}_n\texttt{]}) \xrightarrow{10} \texttt{0}$ **if** $\texttt{G}\in\mathcal{G}$ **and** $\texttt{G}\neq\texttt{f[Z}_1\texttt{,...,Z}_n\texttt{]}$

Rewrite rule 6 implements the occurs check, looking for expressions of the form X=f[Y,Z,X]. In Dyna, we *do not* allow for cyclic data structures (which is allowed in some Prolog implementations). Hence, there is no value $\texttt{X} \in \mathcal{G}$, so this expression is *unsatisfiable*, thus can be rewritten as 0, without having to identify the value of X.

Rewrite rules 7 and 8 perform unification between different structured term rewrites. If two incompatible structured terms are unified together (in this case, $\texttt{f},\texttt{g} \in \mathcal{F}$ and $\texttt{f} \neq \texttt{g}$), then this can be rewritten as 0 as seen in rewrite rule 8.[89] Similarly, when the functor names on two structured term **R**-exprs are equivalent to each other, all of the arguments of the terms are unified together, as in rewrite rule 7. Observe that the result of rewrite rule 7 still includes at least one structured

[89]This would have been equivalent to writing f[X,Y,Z]=g[Q,H], and there is no assignment to X,Y,Z,Q,H that makes this expression true.

unification **R**-expr. The reason is that we still need the constraint on the variable X. Without this constraint, the bag semantics of the **R**-expr would change, and there are likely other conjunctive sub-**R**-exprs in the containing **R**-expr that depend on the value of X.

Rewrite rules 9 and 10 replicate rewrite rules 7 and 8 but are for when the structured term is a ground, fully known value. For example, `f[1,2,3]` is ground and can be entirely represented by the value variable G, and can be handled by rewrite rule 9 or 10. Whereas, `f[1,2,X]` is not ground, as it contains the variable X, and must be handled with rewrite rule 7 or 8.

It can be seen that rewrite rules 2–10 is equivalent to the well-known unification algorithm commonly seen in logic programming [102].

6.1.2 Multiplicity Rewrites

Given the rewrite rules that we have so far, we may end up creating an arithmetic expression like `0*0+1*0+1*1+1*1+0*1`, which can be reduced to the multiplicity `2`. This is done using rewrites that implement basic arithmetic on multiplicities $\mathcal{M}$:

$$\mathtt{M + N} \xrightarrow{11} \mathtt{L} \;\textbf{ if }\; \mathtt{M,N} \in \mathcal{M} \;\textbf{ and }\; \mathtt{M{+}N{=}L}$$
$$\mathtt{M * N} \xrightarrow{12} \mathtt{L} \;\textbf{ if }\; \mathtt{M,N} \in \mathcal{M} \;\textbf{ and }\; \mathtt{M{*}N{=}L}$$

6.2 Joining Relations

So far, we have the rewrites on the equality **R**-expr kinds. To represent a bag with multiple variables, we need to combine multiple equality constraints together, as in section §5.3.1. Furthermore, we need to be able to rearrange the **R**-expr to bring relevant sub-**R**-exprs together so that we can carry out the aforementioned rewrites.

To carry out such rewrites, we use the fact that multiplicities form a commutative semiring under + and $*$. Since any **R**-expr evaluates to a multiplicity, these rewrites can be used to rearrange unions and joins of **R**-exprs $\mathtt{Q}, \mathtt{R}, \mathtt{S} \in \mathcal{R}$:

$\mathtt{1 * R} \xrightarrow{13} \mathtt{R}$ ▷*multiplicative identity*

$\mathtt{0 * R} \xrightarrow{14} \mathtt{0}$ ▷*multiplicative annihilation*

$\mathtt{0 + R} \xrightarrow{15} \mathtt{R}$ ▷*additive identity*

$\infty * \mathtt{R} \xrightarrow{16} \infty$ **if** $\mathtt{R} \in \mathcal{M}$ **and** $\mathtt{R} > \mathtt{0}$ ▷*absorbing element*

$\infty + \mathtt{R} \xrightarrow{17} \infty$ ▷*absorbing element*

$\mathtt{R + S} \overset{18}{\longleftrightarrow} \mathtt{S + R}$; $\mathtt{R * S} \overset{19}{\longleftrightarrow} \mathtt{S * R}$ ▷*commutativity*

$\mathtt{Q + (R + S)} \overset{20}{\longleftrightarrow} \mathtt{(Q + R) + S}$; $\mathtt{Q * (R * S)} \overset{21}{\longleftrightarrow} \mathtt{(Q * R) * S}$ ▷*associativity*

$\mathtt{Q * (R + S)} \overset{22}{\longleftrightarrow} \mathtt{Q * R + Q * S}$ ▷*distributivity*

$\mathtt{R * M} \overset{23}{\longleftrightarrow} \mathtt{R + (R * N)}$ **if** $\mathtt{M,N} \in \mathcal{M}$ **and** $\mathtt{(1+N=M)}$ ▷*implicitly does* $\mathtt{M} = \mathtt{1+N}$

$\mathtt{R} \overset{24}{\longleftrightarrow} \mathtt{R * R}$ **if** $\mathtt{R}$ is a constraint ▷*as defined in section §5.2.2.4*

$\mathtt{R} \overset{25}{\longleftrightarrow} \mathtt{R * S}$ **if** $\mathtt{S}$ is a constraint **and** $\forall_E [\![\mathtt{R}]\!]_E = [\![\mathtt{R*S}]\!]_E$ ▷*S is redundant with R*

Note, rewrite rules 18 to 25 are *bidirectional* rewrites, which may appear contradictory to statement at the start of this chapter that our rewrite system will be *normalizing*. As will be discussed in chapter §8, our rewrite system is normalizing with respect to the *directional* rewrite rules, and bidirectional rewrites, such as rewrite rules 18 to 25, will be handled specially and will only be used to rearrange the **R**-expr to enable directional rewrites.

6.2.1 Simple Example

Let us try applying the rewrites we have to the finite materialized bag **R**-expr, like the **R**-expr from section §5.3.1. Let us compute the intersection between two finite relations **R**-exprs.

$$\begin{pmatrix}\texttt{(X=2)*(Y=6)*3+}\\ \texttt{(Y=7)}\end{pmatrix} * \begin{pmatrix}\texttt{(X=1)*(Y=2)+}\\ \texttt{(X=2)*(Y=6)+}\\ \texttt{(X=2)*(Y=7)*2+}\\ \texttt{(X=5)*(Y=7)}\end{pmatrix} \tag{6.1}$$

We first expand out the first disjunct using the distributive rewrite (rewrite rule 22):

$$\to \quad \texttt{(X=2)*(Y=6)*3 *} \begin{pmatrix}\texttt{(X=1)*(Y=2)+}\\ \texttt{(X=2)*(Y=6)+}\\ \texttt{(X=2)*(Y=7)*2+}\\ \texttt{(X=5)*(Y=7)}\end{pmatrix} \texttt{+ (Y=7) *} \begin{pmatrix}\texttt{(X=1)*(Y=2)+}\\ \texttt{(X=2)*(Y=6)+}\\ \texttt{(X=2)*(Y=7)*2+}\\ \texttt{(X=5)*(Y=7)}\end{pmatrix} \tag{6.2}$$

The assignments to `X` and `Y` can be propagated through the conjunctive **R**-expr using rewrite rule 5:

$$\to \quad \texttt{(X=2)*(Y=6)*3 *} \begin{pmatrix}\texttt{(2=1)*(6=2)+}\\ \texttt{(2=2)*(6=6)+}\\ \texttt{(2=2)*(6=7)*2+}\\ \texttt{(2=5)*(6=7)}\end{pmatrix} \texttt{+ (Y=7) *} \begin{pmatrix}\texttt{(X=1)*(7=2)+}\\ \texttt{(X=2)*(7=6)+}\\ \texttt{(X=2)*(7=7)*2+}\\ \texttt{(X=5)*(7=7)}\end{pmatrix} \tag{6.3}$$

Using rewrite rules 2 and 3 equality constraints which have the same value can be rewritten as `1` and different values are rewritten as `0`:

$$\to \quad \texttt{(X=2)*(Y=6)*3 *} \begin{pmatrix}\texttt{0*0+}\\ \texttt{1*1+}\\ \texttt{1*0*2+}\\ \texttt{0*0}\end{pmatrix} \texttt{+ (Y=7) *} \begin{pmatrix}\texttt{(X=1)*0+}\\ \texttt{(X=2)*0+}\\ \texttt{(X=2)*1*2+}\\ \texttt{(X=5)*1}\end{pmatrix} \tag{6.4}$$

The zeros in the **R**-expr cause branches of the disjunction to be eliminated (rewrite rule 14), and multiplicities can be combined together using multiplication (rewrite rule 12):

$$\rightarrow \quad \texttt{(X=2)*(Y=6)*3 * }\left(\texttt{1}\right)\texttt{+ (Y=7) * }\left(\begin{array}{l}\texttt{(X=2)*2+}\\ \texttt{(X=5)*1}\end{array}\right) \tag{6.5}$$

6.2.2 Structuring Disjuncts and Conjuncts as Tries

Notice that the factored form `(Y=7)*( (X=2)*2 + (X=5)*1 ) + (Y=6)*( (X=2)*3 )` in example (6.5) is a more compact **R**-expr than the sum-of-products form from example (6.1) that we started with, and it is preferable in some settings. In fact, it is an example of a **trie** representation of a bag relation.[90] Like the root node of a trie, the expression partitions the bag of ⟨X, Y⟩ tuples into disjuncts according to the value of Y, and then partitions further by the value of X.

A trie-shaped **R**-expr generally has a smaller branching factor than a sum-of-products **R**-expr. As a result, it is comparatively fast to query it for all tuples that match on a given value of Y, or on a (Y, X) pair, by narrowing down to the matching sub-**R**-exprs at each level of the trie. For example, suppose that we query the **R**-expr from example (6.5) by conjoining it with the query `(Y=5)`, which is represented as an **R**-expr. In this case, we can rewrite `(Y=5)*(Y=7)→0` and `(Y=5)*(Y=2)→0` without having to check the part of the trie for the variable X.

The example query `(Y=5)` provides an opportunity for a larger point. The trie has

[90]We will see **R**-exprs represented as tries again in section §11.6.1.

the form `(Y=7)*(X=2)*R`, where this **R**-expr can be rewritten to `0` based on the first sub-**R**-expr `(Y=7)`, without spending any effort to rewrite `R`, which may represent a large **R**-expr. This is an example of **short-circuiting** evaluation and is the same logic that allows a SAT solver or Prolog solver to backtrack immediately upon detecting a contradiction.

6.3 Built-in R-expr Rewrites

Built-in constraints are an important ingredient in representing bag relations. While they are not the only ingredient,[91] they have the advantage that libraries of built-in constraints, such as `plus(I,J,K)` (section §5.2.2.3), usually come with rewrite rules for reasoning about these constraints [72]. Some of the rewrite rules invoke opaque procedural code.

Recall that the arguments to a `plus` constraint are either variables or ground constants. Not all `plus` constraints can be rewritten, but a library should provide at least the cases:

$$\texttt{plus(I,J,K)} \xrightarrow{26} \texttt{0} \textbf{ if } (\texttt{I},\texttt{J},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{I}+\texttt{J} \neq \texttt{K}) \textbf{ or } \texttt{I} \notin \mathbb{R} \textbf{ or } \texttt{J} \notin \mathbb{R} \textbf{ or } \texttt{K} \notin \mathbb{R}$$

$$\texttt{plus(I,J,K)} \xrightarrow{27} \texttt{1} \textbf{ if } \texttt{I},\texttt{J},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{I}+\texttt{J} = \texttt{K} \qquad \triangleright \textit{Note that } \mathbb{R} \subset \mathcal{G}$$

$$\texttt{plus(I,J,X)} \xrightarrow{28} (\texttt{X=I}+\texttt{J}) \textbf{ if } \texttt{I},\texttt{J} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

$$\texttt{plus(I,X,K)} \xrightarrow{29} (\texttt{X=K}-\texttt{I}) \textbf{ if } \texttt{I},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

$$\texttt{plus(X,J,K)} \xrightarrow{30} (\texttt{X=}\underbrace{\texttt{K}-\texttt{J}}_{\in \mathbb{R}}) \textbf{ if } \texttt{J},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

As an example, the **R**-expr `R` = `proj(J,plus(I,3,J)*plus(J,4,K))` represents the

[91]Other non-recursive base-case ingredients of **R**-exprs are structural equality constraints and user-defined **R**-exprs for user-defined terms.

infinite set of (`I`,`K`) pairs such that $K = (I+3)+4$ arithmetically. (The intermediate temporary variable `J` is projected out.) The rewrite rules already presented (plus a rewrite rule from section §6.4 below to eliminate `proj`) suffice to obtain a satisfactory answer to the query `(I=2)` or `(K=9)`, by rewriting either `(I=2)*R` or `R*(K=9)` to `(I=2)*(K=9)`.[92]

On the other hand, if we wish to reduce the **R**-expr `R` on its own, the above rules do not apply. In the jargon, the two `plus` constraints within `R` remain as **delayed constraints**, which cannot do any work until more of their variable arguments are replaced by constants (e.g., due to equality propagation from a query, as above).

We can do better in this case with a library of additional rewrite rules that implement standard theorems of arithmetic [72]. With these, the `R` from the same example reduces to `plus(I,7,K)`, which is a simpler description of this infinite relation. Such rewrite rules are known as idempotent **constraint handling rules**. Other useful examples concerning `plus` include `plus(0,J,K)` $\xrightarrow{31}$`(K=J)` and `plus(I,J,J)` $\xrightarrow{32}$`(I=0)`, since, unlike the rules at the beginning of this section, they can make progress even on a single `plus` constraint whose arguments include more than one variable.

[92]For example, with `(I=2)*R` the rewrite sequence is as follows:
`(I=2)*proj(J,plus(I,3,J)*plus(J,4,K))` $\xrightarrow{5}$ `(I=2)*proj(J,plus(2,3,J)*plus(J,4,K))`
(Rewrite rule 5, equality propagation)
$\xrightarrow{28}$ `(I=2)*proj(J,(J=5)*plus(J,4,K))` *(Rewrite rule 28 applied to* `plus(2,3,J)`*)*
$\xrightarrow{5}$ `(I=2)*proj(J,(J=5)*plus(5,4,K))` *(Rewrite rule 5, equality propagation)*
$\xrightarrow{28}$ `(I=2)*proj(J,(J=5)*(K=9))` *(Rewrite rule 28 applied to* `plus(5,4,K)`*)*
$\xrightarrow{39,41,13}$ `(I=2)*(K=9)` *(Rewrite rules 13, 39 and 41 used to eliminate projection)*

Similarly, some useful constraint propagators for the `lessthan` relation include `lessthan(J,J)` $\xrightarrow{33}$ `0`; the transitivity rule `lessthan(I,J)*lessthan(J,K)` $\xrightarrow{34}$ `lessthan(I,J)*lessthan(J,K)*lessthan(I,K)`; and `lessthan(0,I)*plus(I,J,K)` $\xrightarrow{35}$ `lessthan(0,I)*plus(I,J,K)*lessthan(J,K)`. The integer domain can be **split** by rules such as `int(I)`$\xrightarrow{36}$`int(I)*(lessthan(I,1)+lessthan(0,I))` in order to allow case analysis of, for example, `int(I)*myconstraint(I)`. Rewrite rules can also be used to propagate the domain information about variable: `lessthan(I,A)*lessthan(I,B)*plus(A,B,C)`$\xrightarrow{37}$ `proj(K,lessthan(I,A)*lessthan(J,B)*plus(A,B,C)*plus(I,J,K)*lessthan(K,C))`. All of these rules apply even if their arguments are variables, so they can apply early in a reduction before other rewrites have determined the values of those variables. Indeed, they can sometimes short-circuit the work of determining those values.

Like all rewrites, built-in rewrites `R` $\to$`S` must not change the denotation of `R`: they ensure $[\![R]\!]_E = [\![S]\!]_E$ for all E. For example, `lessthan(X,Y)*lessthan(Y,X)` $\to^*$ `0` is semantics-preserving because both forms denote the empty bag relation.

6.4 Projection

Projection is implemented using the following rewrite rules. The first two rewrites make it possible to push the `proj(X,⋯)` **R**-expr down through the sums and products of `R`, so that it applies to smaller subexpressions that mention `X`. The third rewrite allows for projections of different variables to commute with each other:

`proj(X,R+S)` $\overset{38}{\longleftrightarrow}$ `proj(X,R) + proj(X,S)` ▷*distributivity over +*

`proj(X,R*S)` $\overset{39}{\longleftrightarrow}$ `R*proj(X,S)` **if** `X` $\notin$ vars(`R`) ▷*see also rewrite rule 45 below*

`proj(X, proj(Y, R))` $\overset{40}{\longleftrightarrow}$ `proj(Y, proj(X, R))` ▷*projections commute with each other*

We can then use the following rewrite rules to eliminate the projection operator from **R**-exprs whose projection is easy to compute. (In other cases, it must remain delayed.)

`proj(X,(X=Y))` $\xrightarrow{41}$ `1` **if** `X`$\neq$`Y`

`proj(X,(X=f[`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`]))` $\xrightarrow{42}$ `1` **if** `X`$\notin \{\mathtt{Y}_1,\ldots,\mathtt{Y}_n\}$

`proj(X,bool(X))` $\xrightarrow{43}$ `2` ▷*cardinality of a variable given bool constraint*

`proj(X,int(X))` $\xrightarrow{44}$ `∞` ▷*cardinality of a variable given int constraint*

`proj(X,R)` $\xrightarrow{45}$ `R*∞` **if** `X` $\notin$ vars(`R`) ▷*cardinality of an unconstrained variable*

`proj(X,proj(Y,boolean_or(X,Y,true)))` $\xrightarrow{46}$ `3` ▷*cardinality of a variable pair given a certain constraint (BOOLEAN_OR has 3 satisfying assignments that result in* `true`*)*

`int(X)*int(Y)*proj(Z,plus(X,Y,Z))` $\xrightarrow{47}$ `int(X)*int(Y)` ▷*cardinality of integer addition (a unique answer always exists)*

How are these rewrites justified? Observe that `proj(X,R)` in an environment E denotes the number of `X` values that are consistent with E's binding of `R`'s other free variables. Thus, we may safely rewrite it as another expression that achieves the same denotation for every E. For example, in the case that a variable is restricted to be a boolean value (e.g. `(X=true)+(X=false)`), then there are two possible assignments to the variable `X` (rewrite rule 43).

6.4.1 Example Projection

As a simple example, let us project the variable `K` out of simple **R**-expr comprised of equality constraints.

```
proj(K,((J=1) * (K=1) +                    ( (J=1) * proj(K, (K=1))
        (J=2) * (K=6) +                    + (J=2) * proj(K, (K=6))
        (J=2) * (K=7) +         →*         + (J=2) * proj(K, (K=7))
        (J=2) * (K=7) +                    + (J=2) * proj(K, (K=7))
        (J=5) * (K=7)  ))                  + (J=5) * proj(K, (K=7)) )

           →* (J=1) + (J=2)*3 + (J=5)
```

Figure 6-1. Example Rewrites with Projection

The second step uses rewrite rules 38 and 39, while the third step involves rewrite rule 41.

6.5 Aggregation

We give the following rewrite rules only for the aggregator `sum`, as isomorphic rewrites apply to the other aggregators. They rewrite `A=sum(X,R)` as a chain of `plus` constraints that maintain a *running total*. The three rewrites handle cases where `R` is expressed as a disjunction of 0, 1, or 2 bag relations, respectively. A larger disjunction such as `(Q+R)+S` is handled recursively.

`A=sum(X,0)` $\xrightarrow{48}$ `0`

`A=sum(X,(X=Y))` $\xrightarrow{49}$ `(A=Y)`

```
A=sum(X, R+S) -50-> proj(B,proj(C,(B=sum(X,(X=agg_null)+R)) *
                              (C=sum(X,(X=agg_null)+S)) *
                              plus(B,C,A) * not_equal(A,agg_null)))
              if R≠(X=agg_null) and S≠(X=agg_null)
```

The first two rewrites, 48 and 49, handle the case where there are 0 or 1 assignments to the variable `X` in the bag relation. If there is nothing contained in the bag being aggregated, then the result of aggregation was defined "as a value not contained in $\mathcal{G}$", which is equivalent to rewriting the expression as `0`. In the case that there is only one value to sum over, then the aggregation does not change the resulting value, hence it can be eliminated.

6.5.1 Rewrite Rule 50 for Handling Disjunctions

The third rewrite rule requires some explanation. Here, we are handling the case where there is a disjunction `R+S` that has been brought to the top of the aggregator. In this case, we can aggregate over `R` and `S` independently of each other and then combine the result using the aggregator's associated operator. In this case, using the built-in `plus(B,C,A)`. To correctly handle the disjunction, there are a few scenarios that we need to handle carefully, namely the cases where `R` $\to^*$ `0` or `S` $\to^*$ `0`.

First, let us consider the easy case where both `R` and `S` are rewritten to have some contributed value. For demonstration purposes, suppose that we have rewritten `R` into `R`$\to$`(X=5)` and rewritten `S` into `S`$\to$`(X=7)`. In this case, the entire **R**-expr would be `A=sum(X, (X=5)+(X=7))`. We need to rewrite this **R**-expr so that the values `5` and `7` are arguments to `plus` so we can compute their sum. Using rewrite rule 50, the resulting **R**-expr includes the sub-**R**-exprs `B=sum(X,(X=agg_null)+(X=5))` `C=sum(X,(X=agg_null)+(X=7))`.[93] In which case we can then use rewrite rule 51 to

[93]The variable names are generated so to not conflict with vars(`R`) or vars(`S`). We are not *literally* using the variable names `B` and `C`. Variables that appear on the right-hand side of a rewrite and inside a projection, such as `B` and `C`, are only chosen for the *visual presentation* of the rewrite, such as in the case of rewrite rule 50.

work around the `(X=agg_null)` disjunct that has been added to this expression.

$$\texttt{A=sum(X, (X=agg\_null)+(X=Y))} \xrightarrow{51} \texttt{(A=Y)}$$

This will assign the value of `(B=5)` and `(C=7)`. Using equality propagation, we get the **R**-expr `plus(5,7,A)`, which can be rewritten using rewrite rule 28 to get the final result `(A=12)`.

Now, let us consider a harder case where one of `R` or `S` is rewritten as `0`. Without loss of generality, we will assume that `R` can be rewritten as `0`. And for demonstration purposes, we will say that `S` can be rewritten as `S`→`(X=7)`. Hence, we could have the rewrite sequence `(A=sum(X,R+S))`→`(A=sum(X,0+S))`→`(A=sum(X,S))`→`(A=sum(X, (X=7)))`→`(A=7)`, where the aggregation is only performed only on `S`. However, using this sequence of rewrite *requires* that we defer rewrite rule 50 until we are sure that all remaining disjuncts *cannot* be rewritten as `0`. This is contradictory to our design, where we allow rewrites to apply whenever they can be matched. As such, rewrite rule 50 can be applied when we match a disjunction inside of the aggregator. To continue with the scenario, for the sake of demonstration, let us consider what happens if we leave off the `(X=agg_null)` disjunct on rewrite rule 50. In which case we will have `(B=sum(X,R))*(C=sum(X,S))` as a sub-**R**-expr on the right-hand side of rewrite rule 50. As already stated, `R` can be rewritten as `0`. Hence, the resulting rewrite sequence would therefore become `(B=sum(X,R))*(C=sum(X,S))`→`(B=sum(X,0))*(C=sum(X,S))` $\xrightarrow{48}$ `0*(C=sum(X,S))` $\xrightarrow{14}$ `0`.

Clearly, this is incorrect. It does not match `(A=7)`, which we know is the correct answer as $\texttt{S}\rightarrow$`(X=7)`. The issue here is that rewrite rule 48 takes `(B=sum(X,0))` and rewrites it as `0`. As such, we need to *suppress* the behavior of rewrite rule 48 while still allowing rewrite rule 49 to rewrite the aggregator so that we can get the contributed value.[94]

To suppress rewrite rule 48 we are going to introduce the disjunct `(X=agg_null)`, so the **R**-expr now be `(B=sum(X,(X=agg_null)+R))*(C=sum(X,(X=agg_null)+S))`. Now when `R` is rewritten as `0`, this causes the first conjunct to become `(B=sum(X,(X=agg_null)+0))` $\xrightarrow{15,49}$ `(B=agg_null)`. Here, the interpretation of the *value* `agg_null` is that the aggregated **R**-expr `R` returned nothing. Given that this no longer rewrites as `0`, this means the **R**-expr conjunctive `(B=agg_null)*(C=sum(X,(X=agg_null)+S))` is not rewritten as `0` but instead rewritten as `(B=agg_null)*(C=7)`. Now, we have to work around the value `agg_null` again. This is done by *extending* the definition of `plus` so that it ignores `agg_null` and treats it like a numerical 0. This is done using rewrite rule 52.

$$\texttt{plus(agg\_null, A, B)} \xrightarrow{52} \texttt{(A=B)}$$
$$\texttt{plus(A, agg\_null, B)} \xrightarrow{52} \texttt{(A=B)}$$

[94] The complexity here is fundamental and must be handled *somehow*. Essentially, we need to disable rewrite rule 48 so that when one disjunctive branch is rewritten as `0`, it does not cause the entire **R**-expr to be rewritten as `0` (e.g. `A=sum(X,0)`$\rightarrow$`0`). Some alternative approaches to solving this problem could be to introduce a second aggregator **R**-expr kind, like `A=sum_of_disjunct(X,R)` or we could introduce a boolean flag onto the aggregator **R**-expr kind to track if the `(X=agg_null)` disjunct is present. In fact, in chapter §11, when discussing a realistic implementation of **R**-exprs, the aggregation kind will use a boolean flag instead of the disjunction trick here. I have chosen to present the disjunction issue using the `(X=agg_null)` approach because I believe this presentation most clearly demonstrates the semantics of aggregation while still representing the issue of aggregating over a disjunction.

This means that we will get `plus(agg_null,7,A)`$\rightarrow$`(A=7)`, which is the correct result for this example.

Finally, there is one last case that we need to handle. This is the case where *both* `R` and `S` are rewritten as `0`. In this case, we have that `(A=sum(X,R+S))`$\rightarrow$`(A=sum(X,0))` $\xrightarrow{49}$ `0`. With `R` and `S` being rewritten as `0` after rewrite rule 50 was applied, we will get the **R**-expr `(B=agg_null)*(C=agg_null)*plus(B,C,A)`. The value `agg_null` will be assigned to the variable `A` because of rewrite rule 52 passing the value through the `plus`. Hence, we will get `(A=agg_null)`, which is not the `0` we want. We can fix this by adding in the constraint `not_equal(A, agg_null)`, which checks that the final value from the aggregator is not `agg_null`. In the case where the value is `agg_null`, this **R**-expr is rewritten as `0`, which causes the conjunction to become `0` which is the behavior we want.[95]

We can further work around the value `agg_null` by also introducing rewrite rule 53, which mimics rewrite rule 50:

$$\texttt{A=sum(X, (X=agg\_null)+R+S)} \xrightarrow{53} \begin{array}{l}\texttt{proj(B,proj(C,(B=sum(X,(X=agg\_null)+R))*}\\ \qquad\qquad\quad\texttt{(C=sum(X,(X=agg\_null)+S))*}\\ \qquad\qquad\quad\texttt{plus(B,C,A)))}\\ \textbf{if } \texttt{R}\neq\texttt{(X=agg\_null)} \textbf{ and } \texttt{S}\neq\texttt{(X=agg\_null)}\end{array}$$

[95]The rewrite rules for `not_equal` can be defined as:
`not_equal(A,B)`$\rightarrow$`1` **if** $\mathtt{A},\mathtt{B} \in \mathcal{G}$ **and** $\mathtt{A} \neq \mathtt{B}$
`not_equal(A,A)`$\rightarrow$`0`

6.5.2 Other Aggregation Rewrites

In addition to rewrite rules 48 to 50, we have rewrite rules to handle other common **R**-exprs seen inside of an aggregator. For example, conjunctions are handled with the following rewrite rules:

$$\texttt{A=sum(X,R*S)} \overset{54}{\longleftrightarrow} \texttt{R*(A=sum(X,S))} \quad \textbf{if } \texttt{R} \text{ is a constraint, and } \texttt{X} \notin \text{vars}(\texttt{R})$$
$$\texttt{A=sum(X,M*(X=T))} \overset{55}{\longrightarrow} \texttt{(A=(T}\times[\![\texttt{M}]\!]\texttt{))} \ \textbf{if } \texttt{M} \in \mathcal{M} \textbf{ and } \texttt{M > 0}$$
$$\texttt{A=sum(X,(X=agg\_null)+M*(X=T))} \overset{56}{\longrightarrow} \texttt{(A=(T} \cdot [\![\texttt{M}]\!]\texttt{))} \ \textbf{if } \texttt{M} \in \mathcal{M} \textbf{ and } \texttt{M > 0}$$

First, rewrite rule 54 allows for aggregators to be permeable to constraints that do not mention the variable X. For example, in the **R**-expr A=sum(X, lessthan(Z,5)*(X=7)), the constraint lessthan(Z,5) does not impact X directly, besides controlling if the entire expression has a non-zero multiplicity. Hence, we can move it out of the aggregator without influencing the overall expression. Additionally, if we have an **R**-expr like (Y=3)*(A=sum(X,plus(4,Y,X))), then the constraint (Y=3) can be brought into the aggregator so that we can evaluate the built-in plus.

Rewrite rules 55 and 56 handle the case that there are other non-constraint **R**-exprs contained inside of the aggregator that does not influence the aggregated value X. For example, if we have an **R**-expr like A=sum(X,(lessthan(Z,5)+ lessthan(1,Z))*(X=3)), then once Z is rewritten as a ground numerical value, then each of the lessthan constraints can be rewritten as either 0 or 1. Summing their multiplicities together, this will result in either 1 or 2. In the case of 1, we can simply remove the 1 from the expression using rewrite rule 13. However, when the multiplicity is 2, we need to count two *copies* of 3 into the aggregator.[96] Additionally, note that rewrite rules 55 and 56 are generalizations of previously presented rewrite rules 49 and 51.

The way non-constraints **R**-exprs are handled differs between different aggregators. For example, rewrite rules 57 and 58 are the same as rewrite rules 55 and 56, but they are defined for the max aggregator instead of sum.

$$\texttt{A=max(X,M*(X=T))} \xrightarrow{57} \texttt{(A=T)} \ \textbf{if} \ \texttt{M} \in \mathcal{M} \ \textbf{and} \ \texttt{M > 0}$$
$$\texttt{A=max(X,(X=agg\_null)+M*(X=T))} \xrightarrow{58} \texttt{(A=T)} \ \textbf{if} \ \texttt{M} \in \mathcal{M} \ \textbf{and} \ \texttt{M > 0}$$

The max aggregator does not care about how many copies a particular value it has, as long as there is at least one copy. In this case, the aggregator ignores the multiplicity M as long as it is greater than 0.

We also define rewrites for handling general constraints involving aggregators.

$$\texttt{A=sum(X,R)} \xrightarrow{59} \texttt{R}\{\texttt{X}\mapsto\texttt{A}\} \ \textbf{if} \ \texttt{R} \text{ is a constraint}$$

With rewrite rule 59, if we are aggregating over a constraint, then we can remove the aggregator (subject to variable renaming). This can be useful when there are nested aggregators, as aggregators themselves are constraints. For example, the **R**-expr (A=sum(X,(X=min(Y,R)))) can be rewritten as (A=min(Y,R)) as the outer sum aggregator does nothing.

[96] Admittedly, A=sum(X,2*(X=7)) can be handled using rewrite rule 23 to first expand two copies of (X=7) as A=sum(X,(X=7)+(X=7)) and then use rewrite rule 50 to sum over the two disjuncts, however this will be less efficient than using rewrite rule 55. For example, suppose that the value of M is several hundred instead of 2. In this case, using rewrite rules 50 and 53 would create a very large **R**-expr.

6.5.3 Rewriting Aggregation With Partial Information

Some aggregators have special behavior, which is expressed using rewrites. One such case in logic programming is an *exists* aggregator (`:-`), which checks that there is *some* true input. As such, once we have found a positive result, we can immediately stop rewriting other disjunctive **R**-exprs contained in the aggregator.

$$\texttt{A=exists(X,(X=true)+R)} \xrightarrow{60} \texttt{A=true}$$

Another case often seen in AI tree search algorithms is alpha-beta pruning with the min/max value of an aggregator. Alpha-beta pruning prunes useless computation once an upper/lower bound is known. This can be expressed using a rewrite rule, which infers `lessthan` constraints once some value is known.

$$\texttt{A=min(X,(X=T)+R)} \xrightarrow{61} \texttt{lessthaneq(A,T)*(A=min(X,(X=T)+lessthan(X,T)*R))} \;\textbf{if}\; \texttt{T} \in \mathcal{G}$$
$$\texttt{A=max(X,(X=T)+R)} \xrightarrow{62} \texttt{lessthaneq(T,A)*(A=max(X,(X=T)+lessthan(T,X)*R))} \;\textbf{if}\; \texttt{T} \in \mathcal{G}$$

The `lessthaneq` on the outside of the aggregator can be used with rewrite rules 34 and 35 to infer if the result of this aggregation will be useful. Similarly, the `lessthan` added to the other disjunctive branch `R` informs that **R**-expr what the value *must be* for it to influence the result of aggregation. This allows for the elimination of useless disjunctive branches of the min/max aggregator.

6.6 Conditional `if`-Expression Rewrites

An `if`-expression switches between two different **R**-exprs. This is used to maintain **R**-exprs, which are only used in a few scenarios to override other **R**-exprs. To rewrite an `if`-expression, the rewrite system must determine if the conditional sub-**R**-expr is "*true*" (has multiplicity greater than zero). Here, the goal is to eventually match the conditional expression using rewrite rules 63 and 64 such that the expression can be rewritten as the true or false branch.

$\texttt{if(M+Q, R, S)} \xrightarrow{63} \texttt{R}$ **if** $\texttt{M} \in \mathcal{M}$ **and** $\texttt{M} > \texttt{0}$ ▷*Return True* **R**-*expr branch*

$\texttt{if(0, R, S)} \xrightarrow{64} \texttt{S}$ ▷*Return False* **R**-*expr branch*

Following from the definition of `if`-expression in section §5.2.2.8, the multiplicity of the `if`-expression **R**-expr is *only* influenced by the *then* (`R`) and *else* (`S`) sub-**R**-exprs. The multiplicity of the condition does not influence the multiplicity besides switching between the two **R**-exprs. However, to be able to rewrite the condition into a form that can be matched by rewrite rules 63 and 64, the conditional sub-**R**-expr needs to be able to "read" from other conjunctive **R**-exprs so it can be rewritten. This is accomplished using rewrite rule 65. Here, *any* conjunctive **R**-expr can be copied into the condition sub-**R**-expr, allowing other rewrites to simplify the conditional **R**-expr. Note, because we only care about the multiplicity of an expression being zero vs. non-zero, duplicating a **R**-expr may change the *magnitude* (e.g., `2` vs `4`) but does not change the result of comparing zero vs. a nonzero multiplicity.

$$\texttt{T*if(Q, R, S)} \overset{65}{\longleftrightarrow} \texttt{T*if(T*Q, R, S)}$$

`If`-expressions provide many opportunities for potential rewrites that may be useful in various contexts. Observe that `if`-expressions allow us to switch between two different **R**-exprs. As such, a rewrite can be used to conditionally *pre*-rewrite some portion of the **R**-expr. This is a central insight to enabling *memoization*, as I will discuss in chapter §10. Doing this as a rewrite, observe that we can introduce an `if`-expression *anywhere* in an **R**-expr as long as both the true and false branches are identical, as in rewrite rule 66.

$\texttt{if(Q, R, R)} \overset{66}{\longleftrightarrow} \texttt{R}$ ▷*Regardless of Q the same* **R***-expr is returned*

Furthermore, we can use the conditional branch of an `if`-expression to rewrite the true branch of the `if`-expression. This is allowed because the true branch is only returned when `Q` is rewritten as nonzero. Hence, the true branch only needs to be *conditionally* semantic preserving under `Q`:

$\texttt{if(Q, R, S)} \xrightarrow{67} \texttt{if(Q, Q*R, S)}$ **if** `Q` is a constraint ▷*Allow possible simplifications of* true*-branch sub-***R***-expr*

Many `if`-expressions can be introduced by the rewriting engine when handling memoization. It is, therefore, beneficial to have a number of rewrites that allow us to rearrange the `if`-expressions to keep the **R**-expr tidy and efficient:

$\texttt{T*if(Q, R, S)} \overset{68}{\longleftrightarrow} \texttt{if(Q, T*R, T*S)}$ ▷*The* `if`*-expression is permeable*

$\texttt{if(Q, R, if(Q2, R2, S))} \xrightarrow{69} \texttt{if(Q+Q2,R+R2,S)}$ **if** $\forall_E$ $[\![\texttt{Q*Q2}]\!]_E\texttt{=0}$ **and** $[\![\texttt{R*R2}]\!]_E\texttt{=0}$[97]
▷*Combine two non-overlapping nested* `if`*-expressions*

$\texttt{if(Q, R, if(Q2,R2,S))} \overset{70}{\longleftrightarrow} \texttt{if(Q+Q2, if(Q, R, R2), S)}$ ▷*Nested* `if`*-expressions can be rearranged*

$\texttt{if(Q, R, S)} \xrightarrow{71} \texttt{S}$ **if** $\texttt{Q}$ is a constraint **and** $[\![\texttt{Q*R}]\!]_E = [\![\texttt{Q*S}]\!]_E$ ▷`if`-*expressions can be eliminated*

$\texttt{if(Q1+Q2, R, S)} \xrightarrow{72} \texttt{if(Q1,R,if(Q2,0,1)*S) + if(Q1,0,1)*if(Q2,R,S)}$ ▷`if`-*expressions can be split*

$\texttt{R*if(R,1,0)} \xleftrightarrow{73} \texttt{R}$ ▷*Redundant* `if`*-expressions constraints can be removed/introduced*

6.7 User-Defined R-exprs Rewrites

User-defined rewrites are used to implement user-defined relations as **R**-exprs, as we will see in chapter §7. These rewrites match with a named functor $\texttt{f} \in \mathcal{F}$ that contain n value types $\texttt{Y}_1, \ldots \texttt{Y}_n \in \mathcal{V}$. The value typed $\texttt{Y}_i$ correspond to the values/variables used where the user-defined **R**-expr kind $\texttt{f}(\cdots)$ appears within a large **R**-expr. Each functor f has <u>at most</u> one rewrite associated with it, and it always rewrites as another **R**-expr $\texttt{R}_\texttt{f}$.

$$\texttt{f}(\texttt{Y}_1,\ldots,\texttt{Y}_n) \xrightarrow{74} \texttt{R}_\texttt{f}\{\texttt{X}_1 \mapsto \texttt{Y}_1, \cdots, \texttt{X}_n \mapsto \texttt{Y}_n\}$$

The variables $\texttt{X}_1, \cdots, \texttt{X}_n$ are identifiers for the free variables contained in the user-defined $\texttt{R}_\texttt{f}$. The free variables of $\texttt{R}_\texttt{f}$ *must* be a subset of $\texttt{X}_1, \cdots, \texttt{X}_n$, otherwise the **R**-expr will become ill-formed ($\text{vars}(\texttt{R}) \subseteq \{\texttt{X}_1,\ldots,\texttt{X}_n\}$). The names of the variables in $\texttt{R}_\texttt{f}$ are renamed to match the names $\texttt{Y}_i$ where f originally appeared.

[97] The condition $\forall_E\ [\![\texttt{Q*Q2}]\!]_E = \texttt{0}$ can be proven by checking if there exists a sequence of rewrites such that $\texttt{Q*Q2} \to^* \texttt{0}$, meaning that *regardless* of the environment, the expression represents an empty bag relation.

6.8 Incompleteness of Included Rewrites

It is impossible to build a sound and complete set of rewrites for all **R**-exprs given that **R**-exprs are sufficiently powerful to express mathematical proofs. This is in contrast to Datalog (section §3.1.2) and SLD resolution [101] that have a more limited representation for terms and can therefore have complete implementations. In other words, a rewrite system that can rewrite *all* **R**-exprs would be capable of solving *all* mathematical proofs—which is impossible under Gödel's incompleteness theorem. This will be shown in section §15.1.2. Rather than attempting to make **R**-exprs and their rewrite complete, we hope to provide a useful but incomplete set of rewrites, much like compute algebra systems such as Mathematica [150] or SymPy [103].

Chapter 7

Conversion of Logic Programs to Relational Expressions

A Dyna program is mechanically converted into **R**-exprs by our front-end parser. I will illustrate the translation to **R**-exprs with a few examples. In general, this chapter should not be surprising for those who are familiar with the design of front-ends for compilers or interpreters.

7.1 Dyna Programs Represent a Key-Value Map

A Dyna program can be thought of as defining a map between keys and valves. Keys are the names of terms in the program, and the value is the returned value computed from a user-defined term's definition in the Dyna program. For example, consider the single rule defined on line 359:

```
359 a(X) += X*X.
```

(a) Dyna

```
a(Arg1,Result) → (Result=sum(Inp,
  proj(X,(X=Arg1)*times(X,X,Inp))))
```

(b) The Program Represented through the `is` relation as an **R**-expr.

Figure 7-1. Single rule Dyna program.

The user-defined term `a(X)` is defined by the rule '`a(X) += X*X.`' on line 359, which defines a relation between *keys* of the form `a(X)`, such as `a(5)`, and the returned values, such as `25` in the case of `a(5)`. We represent this relation as an **R**-expr we denote as `a(X,Result)`.[98] All user-defined terms in Dyna return some value. When representing user-defined *logic* programming expressions, a "dummy" return value of *true* is used.

All **R**-exprs created from Dyna terms have an aggregator at the top of the **R**-expr. The aggregator enforces the functional dependency between the arguments of the user-defined term and its return value. Functional dependencies are not otherwise enforced by **R**-exprs.

7.1.1 Grouping User-Defined Rules by Name

User-defined terms are grouped according to the outer functor name and arity (number of arguments). This grouping is usually referred to using a slash. For example, we write '`a/1`' to reference the term defined in figure 7-1. This grouping

[98] Note that is standard in Prolog to represent functions as a constraint with the return value as the final argument. For example, `append([1,2], [3], [1,2,3])`. This is conceptually equivalent to what we are doing here.

is natural and aligns with the user's expectations in how terms are identified by meta-Dyna rules like `$memo` that controls memoization (section §2.7).

All contributed rules are grouped under an aggregator **R**-expr. For example, `b(X)` in figure 7-2 shows two overlapping rules defined on lines 360 and 361:

```
360 b(X) += X*X.
361 b(X) += 7.
```

(a) Dyna

```
b(Arg1,Result) → (Result=sum(Inp,
  proj(X,(Arg1=X)*times(X,X,Inp))+
  proj(X,(Arg1=X)*(Inp=7))))
```

(b) Program with two user-defined rules combined using a disjunction `+`.

Figure 7-2. Two user-defined rules with overlap.

Arguments to user-defined terms are given placeholder variable names, such as `Arg1` used in these examples. Local variables are introduced using projection and are unified with the placeholder argument variables. This allows us to support expression appearing on the left-hand side of an aggregator without special handling:

```
362 sumlist([]) = 0.
363 sumlist([H|T]) =
364   H+sumlist(T).
```

(a) Dyna

```
sumlist(Arg1,Result) → (Result=only(Inp,
  (Arg1=list_end[])*(Inp=0)+
  proj(H,proj(T,proj(Tmp,
    (Arg1=list_cons[H,T])*
    sumlist(T,Tmp)*
    plus(H,Tmp,Inp))))))
```

(b) **R**-expr

Figure 7-3. The `sumlist` rule (lines 362 to 364 is converted into a single **R**-expr.

This might appear inefficient at first glance, but we can use the rewrite rules that we have already defined to rewrite these **R**-exprs to eliminate unneeded projections (e.g. rewrite rules 5, 39 and 41).

7.1.2 Different Aggregators

Dyna allows for different aggregators to be defined for the same functor name-arity grouping. To support this as an **R**-expr, we first group by the aggregator's name and then nest different aggregators under the 'only' aggregator. The 'only' aggregator corresponds to the "equal-sign aggregator" (=) which ensures that there is only one result; otherwise it will return an error (e.g. `(A=only(X,(X=1)+(X=2)))→(A=error)`).

```
c(X) += X*X
  for X > 0.
c(X) min= exp(X)
  for X < 0.
c(X) min= sin(X)
  for X < 0.
```

(a) Dyna

```
c(Arg1,Result) → (Result=only(Inp1,
  (Inp1=sum(Inp2,
    proj(X,(X=Arg1)*lessthan(0,X)*times(X,X,Inp2))))+
  (Inp1=min(Inp2,
    proj(X, (X=Arg1)*lessthan(X,0)*exp(X,Inp2))+
    proj(X, (X=Arg1)*lessthan(X,0)*sin(X,Inp2))))))
```

(b) only is used to ensure there is only one contribution.

Figure 7-4. Multiple aggregators can co-exist as long as they do not overlap.

7.1.3 Additional Metadata for Aggregators

Some aggregators require additional steps when translating from Dyna into **R**-exprs. For example, the `:=` aggregator allows us to override the value by defining additional rules. To make `last_override` a commutative operator, we annotate each aggregated value with its line number and then select the value contributed from the last defined line number.

```
371 d(X) := X + 1.
372 d(X) := X * 2
373   for X <= 0.
```

(a) Dyna

```
d(Arg1,Result) → (Result=last_overrides(Inp,
  proj(X,proj(Tmp,
    (X=Arg1)
    plus(X,1,Tmp)*
    (Inp=value_from_line[371,Tmp])))+
  proj(X,proj(Tmp,
    (X=Arg1)
    times(X,2,Tmp)*
    (Inp=value_from_line[372,Tmp])*
    lessthaneq(X,0)))))
```

(b) R-expr

Figure 7-5. The translation of `:=` aggregator annotates every contribution with its line number so it can determine which value should be returned.

7.1.4 Built-ins

Built-ins such as `plus` and `times` are mapped from their source code infix representation of '+' and '*'. Other infix operators, such as subtraction '-' do not have their own **R**-expr representation but instead are written in terms of the `plus` relation by reordering its arguments.

```
"subtract"(A,B,C) → plus(A,C,B)      ▷subtract does not exist, it is represented with plus
"divide"(A,B,C) → times(A,C,B)        ▷divide does not exist, it is represented with times
"greaterthan"(A,B) → lessthan(B,A)  ▷greaterthan does not exist, represented as lessthan
```

Figure 7-6. There do not exist **R**-expr kinds for `subtract`, `divide`, or `greaterthan`, hence the "scare quotes" around their name. Instead, they are represented using equivalent built-in **R**-expr kinds by rearranging their arguments.

All built-in callable that are from the source code (such as `C=A+B` in the case of `plus(A,B,C)`) return *some* value, just like the user-defined terms. This includes binary operators expressed as ternary relations like logical and, logical or, less than, and greater than, which all have three value slots. This is done to support logical expressions using *and* '&&', *or* '||', and *not* '!', (e.g. the Dyna expression '`(A<B) || !(C>D)`'). That said, in this dissertation, I will often write relations like `lessthan` using only two of the three slots, as the third argument is almost always the constant value *true*, and this makes the presentation more concise.

`lessthan(A,B,`*true*`)` $\equiv$ `lessthan(A,B)` ▷*Third argument omitted in presentation*

Built-ins are also used to implement language features such as indirect function calls and dictionaries.

```
e(Func,{A, B}) =
  Func(A, B).
```

(a) Dyna, (§2.8)

```
e(Arg1,Arg2,Result) → (Result=only(Inp,
  proj(Func,proj(A,proj(B,proj(Tmp1,
    (Func=Arg1)*
    map_access(Arg2,"A",A,Tmp1)*
    map_access(Tmp1,"B",B,empty_map)*
    indirect_call(Func,A,B,Inp)))))))
```

(b) R-expr

Figure 7-7. Dyna syntactic features with their semantics backed by built-ins

7.1.5 Dynabases

Dynabases and how they are converted into **R**-exprs are deferred until chapter §13.

Chapter 8

A Basic Implementation of R-expr Rewriting

In this chapter, I will discuss a minimal and simple implementation for rewriting **R**-exprs. The system described in this chapter can be implemented in approximately 1000 lines of Python, and the pseudocode for this algorithm can be found at the end of this chapter in section §8.A.

In chapter §11, I will discuss a "more realistic", "fully featured" implementation of term rewriting built on **R**-exprs. I note that the implementation described in the present chapter §8 is sufficient to execute many complicated programs represented as **R**-exprs but not all. Additionally, programs are likely to run *very slowly* and have a suboptimal asymptotic runtime.

The implementations of Dyna using **R**-exprs in this chapter and in chapter §11 are designed to be as close as possible to the "mathematical" design of the **R**-expr semantics and rewrite rules presented in chapters 5 and 6. Internally, Dyna is as *homogeneous* as possible, representing almost all state using **R**-exprs. This includes the program itself, as in chapter §7, queries, query results (as discussed in chapter §2), and memoized and compiled representations (as we will see in chapters 10 and 12).

As discussed in section §2.3, a user's interaction with the Dyna system consists of interleaved queries and updates, much like a database. Queries observe the state of the Dyna program, and updates modify the state of the program. When the Dyna system is queried, it translates the query into an **R**-expr (as in chapter §7), and rewrites the query's **R**-expr into a *semantically equivalent* **R**-expr. Ideally, the resulting **R**-expr will be "*simpler*" and provide a useful answer. I will give an intuitive definition of "*simpler*" shortly and a formal definition in chapter §15. In principle, we are allowed to return *any* semantically equivalent **R**-expr, including the initial query—though echoing back the initial query is undesirable as it is not useful. In this way, Dyna resembles a computer algebra system such as Mathematica [150] or SymPy [103], which has an incomplete but useful collection of identities and proof strategies.

I will start with a high-level description of **R**-exprs, the data structure, and then discuss our core rewriting procedure, a pair of functions called SIMPLIFY and SIMPLIFYNORMALIZE.

8.1 R-exprs, The Data Structure

An **R**-expr is an *immutable* recursive data structure. The **R**-expr is rewritten via recursive functions that return a new **R**-expr or the exact same **R**-expr unmodified. The two most commonly used recursive functions used on an **R**-expr are variable renaming and rewriting for execution via SIMPLIFY described in section §8.2. This design should be familiar to those who have used purely functional data structures and programming languages.

We will usually think of an **R**-expr as a tree data structure, though, in practice, it can be a DAG (directed acyclic graph) with the same sub-**R**-expr pointed to multiple times. This does not matter as the **R**-expr is an immutable data structure. Additionally, sharing internal sub-**R**-exprs can speed up equality checks with pointer equality.

An **R**-expr is always bounded in size. This property ensures that the **R**-expr can be represented in memory and that we can traverse the **R**-expr using recursive depth-first procedures. User-defined **R**-exprs can represent recursive functions (terms that are defined recursively, section §5.2.2.11). At any given point in time, these user-defined **R**-exprs are only expanded up to *some* bounded depth of the recursion.[99]

The class that implements an **R**-expr kind contains named fields that can contain an **R**-expr, value type, or an array of **R**-exprs or value types. A value type can be either a variable identifier[100] or a constant value. This allows us to write both `plus(X,Y,Z)` and `plus(1,2,3)` using the same `plus(·,·,·)` **R**-expr kind. Some classes that implement **R**-exprs contain metadata fields. This allows a single class to implement different **R**-exprs from the Dyna language. For example, equality constraints with structural terms (sections § 2.1.1 and 5.2.2.2 e.g. `X=name[Var1,Var2]`) are all handled by the same **R**-expr kind, where the **R**-expr kind contains array that track the variables (e.g. `[Var1,Var2]`) as well as a string field that tracks the functor's `name`.

Lifetime management of an **R**-expr is handled using the usual garbage collection mechanism in the host programming language.

R-exprs support syntactic equality checking, which requires that all variable names are the same and all sub-**R**-exprs are in the same order. In many cases, this check is bypassed due to shared internal sub-**R**-exprs and by checking pointer equality. All **R**-exprs are hashable and cache their hash code internally. The hash

[99]In other words, if we have an user-defined **R**-expr kind like `a(X)` with a rewrite rule like `a(X)` $\rightarrow$ `a(X) + a(X)`, this will represent an *infinitely large* **R**-expr, due to the recursive expansion. We will only have expanded this **R**-expr up to some limited depth at any given moment. We can continue to perform more and more rewrites on this **R**-expr to continue expanding it without end. As such, in the *limit* it is infinitely size, but for at any given time $t < \infty$ the **R**-expr *will be finite*.

[100]Variable identifiers are allowed to be any Object that supports hashing and equality checks. In practice, the variable identifier is often a string.

code speeds up equality checking when **R**-exprs are *not* equal. It also allows **R**-exprs to be used as the key (rather than just the value) in a hash table.

8.2 Evaluation by Simplifying an R-expr

Dyna programs and user queries are represented as an **R**-expr. Evaluation of the program is performed through the application of the rewrite rules from chapter §6. We call this process **simplification**, and have accordingly named the function that applies these rewrites SIMPLIFY.

The reason we have called this process simplification is that our rewrite rules attempt to rewrite an **R**-expr into another **R**-expr that is "simpler" and "easier to understand". For example, the **R**-expr `plus(1,2,X)` is rewritten into `(X=3)` using rewrite rule 28. The **R**-expr `(X=3)` is *simpler* as no computation is required to identify the value of `X`. Whereas `plus(1,2,X)` requires adding `1` and `2` to get the value `3`. In section §15.3, I will formalize the notion of simpler, but for now, an intuitive definition of "*easier to understand*" and "*requires fewer steps of computation to get a final answer*" will suffice.

8.2.1 Properties of SIMPLIFY

The function SIMPLIFY:$\mathcal{R} \times \mathcal{C} \to \mathcal{R}$ performs a pass of rewriting and takes an **R**-expr `R` and a context $\mathcal{C}$ (to be defined shortly) and rewrites the **R**-expr as another *se-*

mantically equivalent **R**-expr using the rewrite rules from chapter §6.[101] SIMPLIFY is allowed to selectively apply rewrites as long as it guarantees that no rewrite is completely starved.[102] SIMPLIFY is *not* required to apply all possible rewrites every time it is invoked. In fact, to run faster, it will usually only apply a few rewrites each time. SIMPLIFY will return an *identical* **R**-expr if-and-only-if there are *no* directional[103] rewrites that *can* be applied. This property allows SIMPLIFYNORMALIZE to identify that it has finished rewriting[104] the **R**-expr (algorithm 1).

SIMPLIFY recursively invokes itself on the **R**-expr tree structure *depth-first*. For example, when SIMPLIFY encounters an **R**-expr like `R*S`, it rewrites the **R**-expr as "SIMPLIFY(`R`, $\mathcal{C}$)"*"SIMPLIFY(`S`, $\mathcal{C}$)"—recursively invoking SIMPLIFY on `R` and `S` and combining the results using a conjunction '*'. We require that SIMPLIFY runs in a bounded amount of time. This means that SIMPLIFY can scan the entire **R**-expr tree data structure and apply some (but not all) rewrites.[105] This property prevents SIMPLIFY from getting stuck in an *infinite loop* when rewriting an **R**-expr. This property also ensures that we do not starve a rewrite. For example, suppose that we have the **R**-expr `Q*R*S`, where there exists a rewrite sequence such that `R` $\to^*$ `0`.

[101] Pseudocode for SIMPLIFY in section §8.A

[102] Starved meaning that a rewrite is prevented from running because it does not get the "necessary resources" to run. In this case, starved resources mean that the rewrite was not matched (when it could have been) or that it was always not picked to run for some reason.

[103] There are bidirectional rewrites which can be used for rearranging the **R**-expr. Those will be addressed shortly in section §8.2.2.1.

[104] We are done rewriting when there is nothing more to be done. We can identify this case by checking the argument to SIMPLIFY with its returned **R**-expr: SIMPLIFY(`R`,$\mathcal{C}$) == `R`.

[105] A given **R**-expr (with recursion represented as user-defined **R**-exprs) is bounded in size, hence scanning through the entire **R**-expr is guaranteed to be a bounded time operation.

```
1: function SIMPLIFYNORMALIZE(R)
2:     $\mathcal{C} \leftarrow$ ALLCONJUNCTIVEREXPRS(R)                ▷ The context, section §8.2.2.1
3:     repeat
4:         $R_{prev} \leftarrow R$        ▷ The R-expr R is a pointer to an immutable, recursive structure.
5:         $R \leftarrow$ SIMPLIFY(R, $\mathcal{C}$)              ▷ Attempt to make progress by rewriting R
6:     until $R_{prev} == R$                   ▷ If no rewrites are performed, then return R
7:     return R
```

Algorithm 1. SIMPLIFYNORMALIZE invokes SIMPLIFY on the **R**-expr until no more rewrites can be performed by SIMPLIFY. SIMPLIFYNORMALIZE is our *Turing-complete* rewrite processes, whereas SIMPLIFY only performs a bounded number of rewriting steps (section §15.2). SIMPLIFYNORMALIZE is *only* invoked a the *root* of an **R**-expr. This is done because SIMPLIFYNORMALIZE does not guarantee fairness to all potential rewrites, whereas SIMPLIFY ensures that it returns in a bounded amount of time.

We do not want to starve the rewrites on R by spending too much time rewriting Q.

To ensure that an **R**-expr is completely rewritten into the "simplest form possible", we invoke the function SIMPLIFY many times until it reaches a *normal form*, meaning there are no more applicable rewrites, which is indicated by SIMPLIFY returning an identical **R**-expr to the one passed as an argument. Therefore, the normal form is a *fixed-point* of SIMPLIFY, and we call the function that finds this fixed-point SIMPLIFYNORMALIZE (algorithm 1).

SIMPLIFYNORMALIZE is *only* invoked at the *root* of an **R**-expr. The reason for this is that SIMPLIFYNORMALIZE is not guaranteed to return, and on programs that contain *infinite loops* will never return.[106] Conversely, SIMPLIFY is a *single pass*[107]

[106] SIMPLIFYNORMALIZE's non-termination corresponds with the fact that Dyna, and **R**-expr rewriting, is Turing complete. Hence, programs that contain infinite loops, or rewrite forever without cycles, do not terminate. (section §15.2)

[107] SIMPLIFY is allowed to perform more than one rewrite during a single pass of rewriting.

of rewriting, and therefore will not get stuck performing rewrites endlessly and is therefore guaranteed to return.

8.2.2 Finding Applicable Rewrites

Rewrites are grouped by the **R**-expr kind (implementation class) that they match. SIMPLIFY(R, $\mathcal{C}$) will use this grouping and checks all rewrites which *might* match. SIMPLIFY is allowed to perform rewrites on any sub-**R**-expr contained in a larger **R**-expr. For example, if we have the **R**-expr `proj(X,times(Y,X,Z)*plus(1,2,X))`, then SIMPLIFY can rewrite `plus(1,2,X)`→`(X=3)` using rewrite rule 28. The way this is implemented is that recursive **R**-expr kinds, such as such as conjunction (·*·), disjunction (·+·), projection (`proj(X,·)`), or aggregation (`X=sum(A,·)`), have special rules that recursively invoke SIMPLIFY on their sub-**R**-exprs. This design is known as a *visitor pattern* and ensures that SIMPLIFY will visit the sub-**R**-exprs and ensures that SIMPLIFY attempts to rewrite all visited sub-**R**-exprs.

Some rewrite rules only require local syntactic information to match. For example, the **R**-expr `plus(1,2,X)` can be matched with rewrite rule 28 and rewritten as `(X=3)`. The numerical values of the first two arguments are constant value types and are embedded in the **R**-expr. However, this does not work for all rewrites. Many rewrite rules require additional *context* to match. For example, rewrite rules 5, 34, 35 and 37 combine two conjunctive **R**-exprs to infer a third. As presented in chapter §6, these rewrite rules require that the two **R**-exprs combined together must be next to each other to be triggered. Now, the **R**-exprs can be rearranged using the bidirectional rewrites from section §6.2, which handle commutativity, associativity, and distributivity of conjunction and disjunction.[108] However, this also has problems. To efficiently rearrange the **R**-expr would require an "*oracle*" to identify how best to rearrange the **R**-expr. Additionally, these rewrites are bidirectional, meaning that we could get stuck in an unproductive cycle such as in figure 8-1 (e.g. R*S→S*R→R*S). To fix this, we are going to use the *context*.

8.2.2.1 The Context (Filled With Conjunctive R-exprs)

To avoid the issue of rearranging the **R**-expr, we observe that many rewrite rules that apply to conjunctions only change a *part* of the **R**-expr, rather than the entire conjunction. In other words, these rewrites are of the form R*S1→R*S2 where the R remains *unchanged* and only serves as a "*license*" for the rewrite on S1. As such, tracking the presence of R is sufficient to permit rewriting S1→S2.[109] We perform this tracking of conjunctive **R**-exprs using the context $\mathcal{C}$.

The **Context** $\mathcal{C}$ is a set[110] that tracks all **R**-exprs that are conjunctive with the **R**-expr R that passed as an argument to SIMPLIFY. When SIMPLIFY recurses through

[108]This includes rewrite rules 18 to 21 as well as rewrites for permeability of projection and aggregation, rewrite rules 39 and 54.

[109]Our rewrites are semantics preserving, so a rewrite R*S1→R*S2 requires that R*S1 is semantically equivalent to R*S2, however it *does not* require that S1 is semantically equivalent to S2.

[110]The context is a set, not a bag (like **R**-exprs)—it is *allowed* to be *implemented* as a bag, but being a bag does not change the behavior of the context. The reason is that the context is used to identify if a *constraint* is conjunctive with the **R**-expr currently being rewritten. We are allowed to duplicate constraints (section §5.2.2.4, rewrite rule 24), and no rewrites are performed on the context itself.

Rewriting Without a Context and Explicit Bidirectional Rules

Step	Rewrite rule used	R-expr
1)	Initial **R**-expr	`(X=1)*proj(Y,(Z=2)*plus(X,3,Y))`
2)	`(X=1)` is moved into `proj` via rewrite rule 39	`proj(Y,(X=1)*(Z=2)*plus(X,3,Y))`
3)	`(X=1)` and `(Z=2)` are flipped via commutativity (rewrite rule 19)	`proj(Y,(Z=2)*(X=1)*plus(X,3,Y))`
4)	`(X=1)` propagates into `plus(X,3,Y)` via rewrite rule 5	`proj(Y,(Z=2)*(X=1)*plus(1,3,Y))`
5)	`plus(1,3,Y)` is computed using the built-in rewrite rule 28	`proj(Y,(Z=2)*(X=1)*(Y=4))`
6)	`(Z=2)*(X=1)` are moved out of `proj` via rewrite rule 39	`(Z=2)*(X=1)*proj(Y,(Y=4))`
7)	`proj(Y,(Y=4))` is simplified as `Y` is known, via rewrite rule 41	`(Z=2)*(X=1)*1`
8)	Useless application of commutativity (rewrite rule 19)	`(X=1)*(Z=2)*1`

Figure 8-1. If we directly apply rewrite rules without knowledge of other conjunctive **R**-exprs, then we need to use the bidirectional rewrite rules on steps 1,2 and 5 to rearrange the **R**-expr first. Without an "oracle" to tell us when to apply a bidirectional rule, this can result in unproductive rearranging of the **R**-expr or cycles such as on steps 7 and 8.

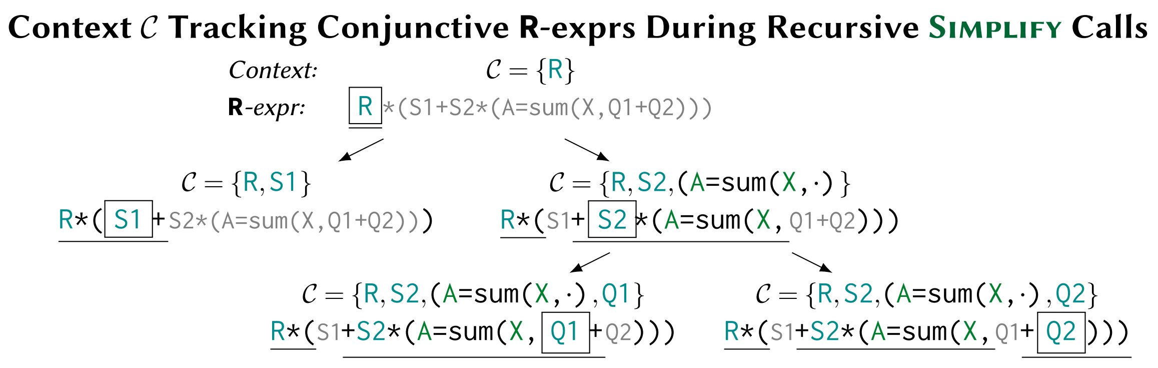

Context 𝒞 Tracking Conjunctive R-exprs During Recursive Simplify Calls
Context:
R-expr:
𝒞 = {R}
R*(S1+S2*(A=sum(X,Q1+Q2)))
𝒞 = {R,S1}
R*(S1+S2*(A=sum(X,Q1+Q2)))
𝒞 = {R,S2,(A=sum(X,·)}
R*(S1+S2*(A=sum(X,Q1+Q2)))
𝒞 = {R,S2,(A=sum(X,·),Q1}
R*(S1+S2*(A=sum(X,Q1+Q2)))
𝒞 = {R,S2,(A=sum(X,·),Q2}
R*(S1+S2*(A=sum(X,Q1+Q2)))

Figure 8-2. The context $\mathcal{C}$ tracks the sub-**R**-exprs that are conjunctive (shown in color and with an underline) when Simplify is invoked on a sub-**R**-expr shown with a $\boxed{\text{box}}$. The **R**-exprs `R`, `S1`, `S2`, `Q1`, `Q2` represent **R**-exprs which are leaves of the **R**-expr expression, such as built-in constraints or user-defined **R**-exprs which have not been expanded yet.

a disjunct, it adds all sub-**R**-exprs that become conjunctive, as seen in figure 8-2. This allows Simplify to find the conjunctive sub-**R**-exprs without scanning the entire **R**-expr. The context, unlike **R**-exprs, can be *mutated* in place, allowing for efficient tracking of conjuncts.[111] When Simplify recurses through the **R**-expr, shallow copies[112] of the context are made to prevent sub-**R**-exprs disjuncts from being added into the parent's context.[113]

The context allows us to avoid using bidirectional rewrites. This means that we do not need an "oracle" to determine how to arrange an **R**-expr and that we can

[111]The context can be seen as similar to the constraint store in constraint logic programming (section §3.1.5).

[112] Shallow copies of the context mean that internal structure is shared as much as possible. The internal data structures used by the context are immutable; hence, they can be shared between copies of the context.

[113]Prolog-based systems use an undo list rather than copies. Both making copies and undo lists solve the same problem. In my opinion, neither approach is strictly better than the other, as each has its own advantages and disadvantages.

Rewriting With a Context and No Bidirectional Rules

Step	Rules used / Explanation	R-expr	Context ($\mathcal{C}$)
1)	Initial **R**-expr and empty Context	`(X=1)*proj(Y,(Z=2)*plus(X,3,Y))`	{}
2)	Conjunctive constraints are added to $\mathcal{C}$	`(X=1)*proj(Y,(Z=2)*plus(X,3,Y))`	{`(X=1)`,`(Z=2)`, `plus(X,3,Y)`}
3)	`X` is propagated using the context	`(X=1)*proj(Y,(Z=2)*plus(1,3,Y))`	{`(X=1)`,`(Z=2)`, `plus(X,3,Y)`}
4)	`plus` is rewritten using rewrite rule 28	`(X=1)*proj(Y,(Z=2)*(Y=4))`	{`(X=1)`,`(Z=2)`, `plus(X,3,Y)`}
5)	`(Z=2)` is lifted out of `proj` via rewrite rule 39	`(X=1)*(Z=2)*proj(Y,(Y=4))`	{`(X=1)`,`(Z=2)`, `plus(X,3,Y)`, `(Y=4)`}
6)	`proj(Y,(Y=4))` is simplified as `Y` is known, via rewrite rule 41, constraints that reference `Y` are removed from $\mathcal{C}$	`(X=1)*(Z=2)*1`	{`(X=1)`,`(Z=2)`}

Figure 8-3. With a context, all conjunctive sub-**R**-exprs are tracked in $\mathcal{C}$. This eliminates the need to rearrange the **R**-expr to apply rewrites such as rewrite rule 5 on step 3 and can be used to implement lifting out of the projection on step 5.

avoid unproductive rewrites. For example, if we use SIMPLIFY to rewrite the example from figure 8-1, then we do not have to explicitly pull down the conjunct `(X=1)` into the projection, and propagation of `X`'s assignment can be handled via the context, as shown in figure 8-3.

The context is used frequently when checking for matches for **R**-exprs. As such, it is important that looking up an **R**-expr in the context is as efficient as possible. To enable this, the context includes indexes that allow for efficient retrieval of relevant **R**-exprs. For example, a particularly frequent case that we have to handle is **R**-exprs that assign a value to a variable (e.g. `(X=7)`, section §5.2.2.1). These **R**-exprs can be efficiently represented using an associative map (hash-map) from the variable's name (e.g. `X`) to the assigned value (e.g. the number `7`).

8.2.3 Canonical Ordering of an **R**-expr

Although bidirectional rewrites such as rewrite rules 18 to 24 are not used for *finding* conjunctive **R**-exprs, there are still some **R**-expr orderings/arrangements that are preferable. For example, placing **R**-exprs that are more likely to be rewritten as `0` earlier can make SIMPLIFY more efficient, as it can stop rewriting earlier, such line 29 of the pseudocode in section §8.A that stops the evaluation of a conjunction once one of the conjuncts is rewritten as `0`.[114]

As such, we have a *canonical* form for the **R**-expr and *do* use bidirectional rewrites to *attempt* to rearrange the **R**-expr into this form. We perform these rearrangement rewrites on an "*is convenient basis*", and they are not considered *necessary*. This means that we perform these rearrangements as long as it is not *too expensive* to match the rewrite and rearrange the **R**-expr.[115]

[114]In the visual presentation in this dissertation, the **R**-exprs which appear to the left are evaluated before the ones on the right. Hence, we can think of SIMPLIFY as evaluating in a *left-to-right* order.

[115]An oracle using only equality constraints and bidirectional rewrites on conjuncts and disjunctions is NP-hard in that it can be used to solve SAT formula. Therefore, it is unrealistic to require that we rearrange all **R**-exprs into an ideal form.

The general principle is that we want the **R**-expr as *factored* as possible. Sub-**R**-exprs are pulled as *high up* as possible in the **R**-expr and out of projection, aggregation, and disjunctions. For example, rewrite rule 22 is used to rewrite `R*S + R*Q`$\rightarrow$`R*(S+Q)`, and rewrite rules 39 and 54 are used to pull out of projection and aggregation: to repeat for convenience, `proj(X,R*S)`$\xrightarrow{39}$`R*proj(X,S)` and `(A=sum(X,R*S))`$\xrightarrow{54}$`R*(A=sum(X,S))` where `X` $\notin$ vars(`R`), and `R` must be a constraint when using rewrite rule 54.

8.2.3.1 Why factored R-exprs are Preferable

In logic programming *without aggregation*, creating factored **R**-exprs would only be a *nice feature*. However, aggregation can make it essential that we factor **R**-exprs. The reason is that pulling a constraint from an aggregator *requires* that it is conjunctive with the entire body of the aggregator—meaning it has been factored from any nested disjunctions.

To see this, let us start with a Prolog-style program and translate it into **R**-exprs with and without Dyna's aggregators.

```
a(X) :- q(X) for X > 5.
a(X) :- s(X) for X > 5.
b(Y) :- r(Y) for Y < 5.
b(Y) :- t(Y) for Y < 5.
should_be_nothing :- a(Z), b(Z).
```

Figure 8-4. Dyna program which requires solving the intersection of `Z` > 5 and `Z` < 5 to identify `should_be_nothing` is unsatisfiable (rewrites as `0`).

The rule `should_be_nothing` represents the intersection of a set of values greater than 5 and another set of values less than 5, which is empty, as such this rule results in nothing: `lessthan(Z,5)*lessthan(5,Z)`$\xrightarrow{34,33}$`0`.

First, let us consider what `should_be_nothing` looks like as an **R**-expr when there are no aggregators:

```
((lessthan(5,Z)*q(Z,ARes))+                                    ▷rule a(X)
 (lessthan(5,Z)*s(Z,ARes)))*                                   ▷rule a(X)
((lessthan(Z,5)*r(Z,BRes))+                                    ▷rule b(Y)
 (lessthan(Z,5)*t(Z,BRes)))                                    ▷rule b(Y)
```

Figure 8-5. Translation of figure 8-4 without aggregators.

We can prove that the **R**-expr in figure 8-5 is empty by combining `lessthan(5,Z)` and `lessthan(Z,5)` into the same conjunction. Without aggregation, this can be accomplished using the distributive rewrites to expand this into four different conjunctive cases:

```
(lessthan(5,Z)*q(Z,ARes)*lessthan(Z,5)*r(Z,BRes))+
(lessthan(5,Z)*q(Z,ARes)*lessthan(Z,5)*t(Z,BRes))+
(lessthan(5,Z)*s(Z,ARes)*lessthan(Z,5)*r(Z,BRes))+      → 0
(lessthan(5,Z)*s(Z,ARes)*lessthan(Z,5)*t(Z,BRes))
```

Figure 8-6. Figure 8-5 expanded into four conjunctive cases, and each individually rewritten as `0`.

Each conjunction can be individually rewritten as `0`, which allows the entire

R-expr to be rewritten as `0`, proving that `should_be_nothing` is empty.[116]

Now, let us again consider the program from figure 8-4, but this time translate it into an **R**-expr with aggregators, as in figure 8-7:

```
(ShouldBeNothing=exists(true,
  proj(Z,
    (true=exists(true,                                              ▷rule a(X)
      (proj(ARes, (lessthan(5,Z)*q(Z,ARes))+
       proj(ARes, (lessthan(5,Z)*s(Z,ARes))))))*
    (true=exists(true,                                              ▷rule b(Y)
      (proj(BRes, (lessthan(Z,5)*r(Z,BRes))+
       proj(BRes, (lessthan(Z,5)*t(Z,BRes))))))))
```

Figure 8-7. Figure 8-4 translated into **R**-exprs with aggregators.

In figure 8-7, the aggregator `exists` *prevents* us from using the distributive property to expand this **R**-expr into the four conjunctive cases. Furthermore, the four `lessthan` constraints are nested under a disjunction, as the `lessthan` constraint was contributed by each rule from the original Dyna program contributing its own `lessthan` constraint rather than having a "global" `lessthan` constraint on `a(X)` or `b(Y)`. As such, the only way to solve this program is to factor the `lessthan` constraint out of the **R**-expr and pull it out of the projection, the disjunction, and then the aggregator. The resulting **R**-expr is shown in figure 8-8.

[116]As an interesting side note, observe that the expansion in figure 8-6 is conceptually equivalent to what Prolog (and constraint logic programming built on Prolog) does. The reason is that each line of figure 8-6 represents a conjunction of constraints. This would be that each conjunction is generated by Prolog when expanding the program using backward chaining (section §3.1.1).

```
(ShouldBeNothing=exists(true,
  proj(Z,
    lessthan(5,Z)*
    (true=exists(true,                                        ▷rule a(X)
      (proj(ARes, q(Z,ARes))+
       proj(ARes, s(Z,ARes)))))*
    lessthan(Z,5)*
    (true=exists(true,                                        ▷rule b(Y)
      (proj(BRes, r(Z,BRes))+
       proj(BRes, t(Z,BRes)))))))
```

Figure 8-8. Figure 8-7 after factoring `lessthan` constraint out of the projections, disjunctions and aggregators.

Once we have constructed the **R**-expr in figure 8-8, we have that both `lessthan(5,Z)` and `lessthan(Z,5)` are in the same conjunct. This means that we can use rewrite rules 33 and 34 to rewrite this conjunction as `0`. This proves that `should_be_nothing` is false.

8.A Appendix: Basic Simplify Pseudocode

Here is the high-level pseudocode of Simplify. The Simplify function recursively calls itself on the **R**-expr. Simplify starts by matching against the *kind* of the **R**-expr. Then, a more detailed matching is performed using the context. This includes checking if a variable is assigned some value (using isGround) and getting the value (using getValue). The context is mutated in place to track conjuncts that are added. When Simplify recurses through recursive **R**-exprs like projection, disjunction, or aggregation, it makes a "copy"[112] of the context to scope conjunctions that are added to the context. All new conjuncts are immediately added to the context using allConjunctiveRexprs to find all conjunctive **R**-exprs before applying the rewrites.

```
function Simplify($\mathcal{C}$, R)
    if R matches (X=Y) :                                  ▷ Equality constraints, section §5.2.2.1
        if isGround($\mathcal{C}$, X) and isGround($\mathcal{C}$, Y) :
            if getValue($\mathcal{C}$, X) == getValue($\mathcal{C}$, Y) :
                return 1                  ▷ Return the multiplicity R-expr 1 to indicate success
            else
                return 0                  ▷ Return the multiplicity R-expr 0 to indicate failure
        else if isGround($\mathcal{C}$, Y) :
            $\mathcal{C}$[X] ← getValue($\mathcal{C}$, Y)                 ▷ Record value of X into the context
            return 1                                 ▷ Remove equality constraint from R-expr
        else if other similar rewrites omitted for brevity :
            other similar rewrites omitted for brevity
    else if R matches (X=f[$Y_1$, ⋯ $Y_n$]) :                ▷ Structured terms, section §5.2.2.2
        if isGround($\mathcal{C}$, X) :
            x ← getValue($\mathcal{C}$, X)
            if x matches f[$Z_1$, ⋯, $Z_n$] :
                return ($Y_1$=$Z_1$)*⋯*($Y_n$=$Z_n$)             ▷ Unpack the value, rewrite rule 9
            else
                return 0                                 ▷ Incompatible value, rewrite rule 10
        else if isGround($\mathcal{C}$, $Y_1$) and ⋯ and isGround($\mathcal{C}$, $Y_n$) :
            $z_1$ ← getValue($\mathcal{C}$, $Y_1$), . . ., $z_n$ ← getValue($\mathcal{C}$, $Y_n$),
            return (X=f[$z_1$, . . . , $z_n$])
        else if (X=f[$Z_1$, ⋯, $Z_n$]) ∈ $\mathcal{C}$ :
            return ($X_1$=$Z_1$)*⋯*($X_n$=$Z_n$)      ▷ Unify with another R-expr that is the same
structural term, rewrite rule 7
        else if (X=g[$Z_1$, ⋯, $Z_m$]) ∈ $\mathcal{C}$ :
            return 0                  ▷ Incompatible with different term name, rewrite rule 8
    else if R matches Q*S :
        Q' ← Simplify($\mathcal{C}$, Q)
        if Q' matches 0 :   ▷ Avoid unnecessary rewriting when we get a 0, rewrite rule 14
```

```
            return 0
        S' ← SIMPLIFY($\mathcal{C}$, S)
        if Q' ∈ $\mathcal{M}$ and S' ∈ $\mathcal{M}$ :
            return Q' ∗ S'                    ▷ Multiply the multiplicities together, rewrite rule 12
        else if S' matches 0 :
            return 0
        else if Q' matches 1 :                ▷ Handle identity, rewrite rule 13
            return S'
        else if S' matches 1 :
            return Q'
        else
            return Q'*S'
    else if R matches proj(X, Q) :
        $\hat{\mathcal{C}}$ ← COPY($\mathcal{C}$)      ▷ A "copy" is made to avoid interfering with the parent context
        $\hat{\mathcal{C}}$ ← $\hat{\mathcal{C}}$∪ ALLCONJUNCTIVEREXPRS(Q)      ▷ R-exprs that are now conjunctive are added to the context
        Q' ← SIMPLIFY($\hat{\mathcal{C}}$, Q)
        if ISGROUND($\hat{\mathcal{C}}$, X) :          ▷ If ground in the context used inside the projection
            $x$ ← GETVALUE($\hat{\mathcal{C}}$, X)
            return Q'{X ↦ $x$}     ▷ Rename X to the constant $x$, eliminate the projection
        if Q' matches Q1*Q2 where X ∉ vars(Q1) and ISCONSTRAINT(Q1) :
            return Q1*proj(X,Q2)     ▷ Pull out constraints that do not depend on X
        return proj(X, Q')                    ▷ Can not eliminate the projection
    else if R matches Q+S :
        $\hat{\mathcal{C}}_1$ ← COPY($\mathcal{C}$)                  ▷ Rewrite first branch of disjunct
        $\hat{\mathcal{C}}_1$ ← $\hat{\mathcal{C}}_1$∪ ALLCONJUNCTIVEREXPRS(Q)
        Q' ← SIMPLIFY($\hat{\mathcal{C}}_1$, Q)
        $\hat{\mathcal{C}}_2$ ← COPY($\mathcal{C}$)                  ▷ Rewrite second branch of disjunct
        $\hat{\mathcal{C}}_2$ ← $\hat{\mathcal{C}}_2$∪ ALLCONJUNCTIVEREXPRS(S)
        Q' ← SIMPLIFY($\hat{\mathcal{C}}_2$, S)
        if Q' matches 0 :
            $\mathcal{C}$ ← $\mathcal{C} \cup \hat{\mathcal{C}}_2$       ▷ One disjunct is 0, so return the other
            return S'
        else if S' matches 0 :
            $\mathcal{C}$ ← $\mathcal{C} \cup \hat{\mathcal{C}}_1$
            return Q'
        else
```

```
66:             $\mathcal{C} \leftarrow \mathcal{C} \cup (\hat{\mathcal{C}}_1 \cap \hat{\mathcal{C}}_2)$          ▷ Common constraints are pulled out
67:             ▷ Constraints not kept in $\mathcal{C}$ are encoded back into the R-expr
68:             ▷ (Equality constraints are removed from the R-expr by line 9 and saved back)
69:             return ( $(\hat{\mathcal{C}}_1 - \mathcal{C})$*Q') + ( $(\hat{\mathcal{C}}_2 - \mathcal{C})$*S')
70:     else if R matches (A=sum(X,Q)) :
71:         $\hat{\mathcal{C}} \leftarrow$ copy($\mathcal{C}$)          ▷ A "copy" is made to avoid interfering with the parent context
72:         $\hat{\mathcal{C}} \leftarrow \hat{\mathcal{C}} \cup$ allConjunctiveRexprs(Q)      ▷ R-exprs that are now conjunctive are added to the context
73:         Q' ← Simplify($\hat{\mathcal{C}}$, Q)
74:         if Q' matches (X=$x$) :
75:             return (A=$x$)                    ▷ Aggregation "complete" return, rewrite rule 49
76:         else if Q' matches Q1+Q2 and Q1 & Q2 not match (X=agg_null) :
77:             return proj(B,proj(C,(B=sum(X,(X=agg_null)+Q1))* ▷ Rewrite rule 50
78:             (C=sum(X,(X=agg_null)+Q2))*plus(B,C,A)*not_equal(A,agg_null)))
79:         else if Q' matches Q1*Q2 where X $\notin$ vars(Q1) and isConstraint(Q1) :
80:             return Q1*(A=sum(X,Q2))        ▷ Pull out constraints that do not influence X
81:         return (A=sum(X,Q'))                ▷ Return the R-expr with newly rewritten Q'
82:     else if R matches if(Q,R,S) :
83:         $\hat{\mathcal{C}} \leftarrow$ copy($\mathcal{C}$)          ▷ A "copy" is made to avoid interfering with the parent context
84:         Q' ← Simplify(Q, $\hat{\mathcal{C}}$)
85:         if Q' matches 1+T   (for some T) :                        ▷ rewrite rule 63
86:             return R                                ▷ Return true branch of if-expression
87:         else if Q' matches 0 :                                     ▷ rewrite rule 64
88:             return S                                ▷ Return false branch of if-expression
89:         else
90:             return if(Q',R,S)  ▷ Unable to evaluate conditional, return delayed if-expr
91:     else if R matches f(Y$_1$, Y$_2$, $\cdots$, Y$_n$) :                 ▷ Calling a user function
92:         R$_f$ ← LookupUserDefinition(f, $n$)
93:         return R$_f\{$X$_1 \mapsto$ Y$_1$,X$_2 \mapsto$ Y$_2, \cdots,$X$_n \mapsto$ Y$_n\}$ ▷ Rename variables to names used by the R-expr currently being rewritten, R
94:     else if R matches plus(I,J,K) :
95:         if isGround($\mathcal{C}$, I) and isGround($\mathcal{C}$, J) and isGround($\mathcal{C}$, K) :    ▷ Rule 27 and 26
96:             if getValue($\mathcal{C}$, I) + getValue($\mathcal{C}$, J) == getValue($\mathcal{C}$, K) :
97:                 return 1                ▷ Assignment to I, J and K is consistent with plus
98:             else
```

```
 99:                return 0                 ▷ Assignment to I, J and K is inconsistent with plus
100:         else if isGround($\mathcal{C}$, I) and isGround($\mathcal{C}$, J) and not isGround($\mathcal{C}$, K) : ▷ Rule 28
101:             $k \leftarrow$ getValue($\mathcal{C}$, I) $+$ getValue($\mathcal{C}$, J)          ▷ Compute using addition
102:             return (K=$k$)
103:         else if isGround($\mathcal{C}$, I) and not isGround($\mathcal{C}$, J) and isGround($\mathcal{C}$, K) : ▷ Rule 29
104:             $j \leftarrow$ getValue($\mathcal{C}$, I) $-$ getValue($\mathcal{C}$, K)       ▷ Compute using subtraction
105:             return (J=$j$)
106:         else if not isGround($\mathcal{C}$, I) and isGround($\mathcal{C}$, J) and isGround($\mathcal{C}$, K) : ▷ Rule 30
107:             $i \leftarrow$ getValue($\mathcal{C}$, J) $-$ getValue($\mathcal{C}$, K)       ▷ Compute using subtraction
108:             return (I=$i$)
109:         return R                                     ▷ Return the **R**-expr unmodified
110:     else if R matches lessthan(I,J) :
111:         if isGround($\mathcal{C}$, I) and isGround($\mathcal{C}$, J) :
112:             if getValue($\mathcal{C}$, I) $<$ getValue($\mathcal{C}$, J) :
113:                 return 1                 ▷ Assignment to I and J is consistent with lessthan
114:             else
115:                 return 0               ▷ Assignment to I and J is inconsistent with lessthan
116:         else if lessthan(J,K) $\in \mathcal{C}$ and lessthan(I,K) $\notin \mathcal{C}$ :          ▷ Rewrite rule 34
117:             $\mathcal{C} \leftarrow \mathcal{C} \cup \{$lessthan(I,K)$\}$ ▷ Track via context that a new lessthan is inferred
118:             return R*lessthan(I,K)
119:         else if lessthan(K,I) $\in \mathcal{C}$ and lessthan(K,J) $\notin \mathcal{C}$ :          ▷ Rewrite rule 34
120:             $\mathcal{C} \leftarrow \mathcal{C} \cup \{$lessthan(K,J)$\}$ ▷ Track via context that a new lessthan is inferred
121:             return R*lessthan(K,J)
122:         return R                                     ▷ Return the **R**-expr unmodified
123:     else if R matches $\cdots$ :
124:         Omitted for brevity: Many other rewrites and matching rules.
```

Algorithm 2. An example of how a few rules of Simplify are implemented.

```
function isGround($\mathcal{C}$, V)
    if V $\in \mathcal{G}$ :                    ▷ A ground constant as a value type is always ground
        return true
    else if $\mathcal{C}$[V] :                  ▷ Check if there exists a binding to V in the context $\mathcal{C}$
        return true
    else
        return false
function getValue($\mathcal{C}$, V)
    if V $\in \mathcal{G}$ :
        return V
    else if $\mathcal{C}$[V] :
        return $\mathcal{C}$[V]
    else
        throw  Error                            ▷ Should not get value if one does not exist
function allConjunctiveRexprs(R)
    ▷ Return all of the conjunctive R-expr
    if R matches Q*S :
        return allConjunctiveRexprs(Q) $\cup$ allConjunctiveRexprs(S)
    else if R matches proj(X,Q) :
        c $\leftarrow$ allConjunctiveRexprs(Q)
        V $\leftarrow$ makeDummyVariable( )
        return $\{$S$\{$X $\mapsto$ V$\}$ : S $\in c\}$      ▷ Rename variable X to a dummy variable name
    else if R matches A=sum(X,Q) :
        c $\leftarrow$ allConjunctiveRexprs(Q)
        V $\leftarrow$ makeDummyVariable( )
        return $\{$S$\{$X $\mapsto$ V$\}$ : S $\in c\} \cup \{$A=sum(X,Q)$\}$      ▷ Rename variable X
    else if R matches Q+S :
        return $\{$Q+S$\}$                      ▷ Return the disjunction itself, not its children
    else
        return $\{$R$\}$
function copy($\mathcal{C}$)
    ▷ Make a copy of the context $\mathcal{C}$. The copied context shares any immutable internal state
      to avoid making a deep copy.
```

Algorithm 3. Helper functions used by Simplify for interacting with the context.

Chapter 9

Rearranging R-exprs to Enable Further Rewriting

In chapter §8, I introduced the SIMPLIFY procedure, which is central in rewriting **R**-exprs. While SIMPLIFY is capable of handling a large number of programs, it is not complete. This is both in a theoretical sense, in that there will always exist programs which SIMPLIFY cannot be reduced into a "simple" representation, and in a practical sense, in that there are programs that we *know* how to solve but SIMPLIFY is unable to solve.

In this chapter, I will introduce a formalism for additional rewrites, which can be used to rearrange the **R**-expr in useful ways. These rewrites, like our bidirectional rewrites from chapter §6, are *not* explicitly used. Rather, these rewrites will *license* us to perform more complicated sequences of rewrites in chapter §11, just as the rewrites in chapter §6 licensed the SIMPLIFY procedure in chapter §8.

9.1 The Problem with SIMPLIFY from Chapter §8

Let us first work through an example where SIMPLIFY fails to produce a "useful" and "simple" result.

Our SIMPLIFY function creates *factored* **R**-exprs so that it can pull constraints out of an aggregator (section §8.2.3). As already discussed in section §8.2.3.1, this is something we have to do to work around aggregators. However, there are many problems that require us to use distributive rewrites to *expand* an **R**-expr into smaller bags[117] to be able to solve the program. Hence, we need *some way* to divide the **R**-expr into smaller, solvable units while still maintaining the factored **R**-expr so that we can work around aggregators.

To see this, let us work through a program that requires subdividing the problem into smaller subproblems. One such program is boolean SATisfiability, commonly known as the SAT problem [41, 96]. We can write a SAT formula as a Dyna program, as I have done in figure 9-1. A SAT formula is comprised of boolean variables and clauses. Variables can take on the value *true* or *false*. Each clause is comprised of one or more literals, which are either variables or the negation of variables. Each clause must have at least one literal that is *true*. Solving a SAT formula requires finding an assignment to all of the variables such that all clauses are satisfied or

[117] By smaller bag, I mean that there are fewer elements in the bag. For example, the bag $\wr\langle \mathsf{X} = 1\rangle@1\int$ is smaller than the bag $\wr\langle \mathsf{X} = 1\rangle@2, \langle \mathsf{X} = 7\rangle@3\int$. A *smaller bag* can have *more constraints* and *more conjunctive* **R**-*exprs*.

proving that there is no assignment that can satisfy all clauses.

```
boolean(1).                                                       % true value
boolean(0).                                                       % false value
negate(1) = 0.               % negate the argument by mapping true to false
negate(0) = 1.                                                % and vice versa
one_true(1, _, _).              % check that at least one argument is true
one_true(_, 1, _).
one_true(_, _, 1).
one_true(1, _).             % check one argument is true for two variables
one_true(_, 1).
one_true(1).                          % check that the one argument is true

   % represent a SAT problem as a conjunct of disjuncts (using one_true)
is_sat :- boolean(A), boolean(B), boolean(C),    % variables are boolean
          one_true(A, B, negation(C)),                        % SAT clause
          one_true(negation(A), negation(B)),                 % SAT clause
          one_true(C),                                        % SAT clause
          one_true(A, negation(B)).                           % SAT clause
```

Figure 9-1. The Boolean SAT formula $(A \lor B \lor \neg C) \land (\neg A \lor \neg B) \land C \land (A \lor \neg B)$ expressed as a Dyna program

The SAT problem is known to be NP-complete and, therefore, difficult to solve. SAT can be solved using the DPLL algorithm [45].[118] The DPLL algorithm works by searching for possible assignments to variables in the problem. The DPLL does this by alternating between *branching* and *propagating*. When branching, the DPLL algorithm will arbitrarily pick an unassigned variable[119] and loop over the variable's

[118]Note: The DPLL algorithm from [45] is no longer considered the state-of-the-art approach for solving SAT problems. However, current state-of-the-art methods for solving SAT (such as conflict driven clause learning, CDCL) still use DPLL at their "*core*".

[119]Branching is allowed to select any unassigned variable. The order of variables that are selected can have a big impact on the wall clock runtime of the DPLL algorithm, and there is a lot of research focused on developing better branching heuristics.

domain (in this case, the domain is $\{1, 0\}$), trying each assignment in turn. When a variable is assigned, the DPLL algorithm *propagates* the variable's assignment to all clauses in the program. If an assignment to a variable is found to be inconsistent, the DPLL algorithm backtracks to the last branching location. Propagation can also cause other variables to be assigned, which also need to be propagated. If propagation completes without any inconsistencies, then another variable will be selected for branching. The DPLL algorithm will continue this process of alternating between branching and assignment.

Given the above explanation of the DPLL algorithm, let us take the SAT formula from figure 9-1 and see what happens when we run it under the SIMPLIFY function defined in chapter §8. The first step is to translate figure 9-1 into an **R**-expr as shown in figure 9-2 using the method described in chapter §7.

```
IsSat=exists(true,
  proj(A, proj(B, proj(C,
    (true=exists(true,(A=1)+(A=0)))*              ▷boolean constraint on A, line 393
    (true=exists(true,(B=1)+(B=0)))*              ▷boolean constraint on B, line 393
    (true=exists(true,(C=1)+(C=0)))*              ▷boolean constraint on C, line 393
    (true=exists(true,(A=1)+                                ▷Clause from line 394
                      (B=1)+
                       proj(NegC,                  ▷Negation expanded from line 394
                            (NegC=only(Inp,(Inp=0)*(C=1)+
                                           (Inp=1)*(C=0)))*
                            (NegC=1))))*
    (true=exists(true, proj(NegA,                           ▷Clause from line 395
                            (NegA=only(Inp,(Inp=0)*(A=1)+
                                           (Inp=1)*(A=0)))*
                            (NegA=1))+
                       proj(NegB,
                            (NegB=only(Inp,(Inp=0)*(B=1)+
                                           (Inp=1)*(B=0)))*
                            (NegB=1))))*
    (true=exists(true, (C=1)))*                             ▷Clause from line 396
    (true=exists(true, (A=1)+                               ▷Clause from line 397
                       proj(NegB,
                            (NegB=only(Inp,(Inp=0)*(B=1)+
                                           (Inp=1)*(B=0)))*
                            (NegB=1))))))))
```

Figure 9-2. Boolean SAT formula (figure 9-1) translated into an **R**-expr. User-defined **R**-exprs have been expanded (section §6.7). Some variables (such as `NegA`) can be rewritten away due to their assignment (e.g. with `(NegA=1)`), but were left in for clarity about the return value of the negation function (lines 383 to 384).

Once we have the **R**-expr in figure 9-2, the Simplify function will look for opportunities to apply its rewrite rules. The relevant rewrite rule in this case is lifting `(C=1)` out of the aggregator `(true=exists(true, (C=1)))` from line 396 using rewrite rule 54. This rewrite leaves us with `(C=1)*(true=exists(true,1))`. The

```
IsSat=exists(true,
  proj(A, proj(B, proj(C,
    (C=1)*                                               ▷Assignment from propagation
    (true=exists(true,(A=1)+(A=0)))*                   ▷boolean constraint on A, line 393
    (true=exists(true,(B=1)+(B=0)))*                   ▷boolean constraint on B, line 393
    (true=exists(true, (A=1)+                                     ▷Clause from line 394
                       (B=1)))*
    (true=exists(true, (1=only(Inp,(Inp=0)*(A=1)+                 ▷Clause from line 395
                                    (Inp=1)*(A=0)))
                       (1=only(Inp,(Inp=0)*(B=1)+
                                    (Inp=1)*(B=0)))))*
    (true=exists(true, (A=1)+                                     ▷Clause from line 397
                       (1=only(Inp,(Inp=0)*(B=1)+
                                    (Inp=1)*(B=0))))))))))
```

Figure 9-3. SAT formula from figure 9-2 after propagation and some rewrites. The SIMPLIFY function from chapter §8 does not perform further rewrites on this **R**-expr.

(true=exists(true,1)) can be rewritten as 1, therefore removed from the **R**-expr, and (C=1) is now at the top level of the **R**-expr and is therefore used to propagate the assignment of true (represented as 1) to the variable C.

The propagation of (C=1) is done by adding it to the context $\mathcal{C}$ and will cause all **R**-exprs of the form (C=0) to be rewritten as 0. Hence, the negation(C)) from line 394, which is equivalent to (C=0) is rewritten as 0 and *removed* from this clause entirely.

Once propagation is completed, we are left with the **R**-expr in figure 9-3 and there are no more rewrites that SIMPLIFY *will* apply.[120] Continuing from this point requires that one of the remaining unassigned variables, A or B, is assigned some

[120]The SAT formula at this point is equivalent to $(A \vee B) \wedge (\bar{A} \vee \bar{B}) \wedge (A \vee \bar{B})$ which does not contain any unit clauses that can be used for propagation.

value. This requires branching, which SIMPLIFY from chapter §8 does not support. In this way, we can see that SIMPLIFY is capable of propagation but does not support branching.

9.2 Using Nested Constraints

When we have an **R**-expr similar to the one in figure 9-3, we want to expand the **R**-expr by branching to subdivide the **R**-expr into smaller **R**-exprs which are solved individually. In other words, given an **R**-expr of the form `R*(Q1+Q2)`, it should rewrite into `R*Q1+R*Q2`.[121] However, before we can get to an **R**-expr of the form `R*(Q1+Q2)`, there is another problem that we *must* solve. Observe that in figure 9-3, there is no *top-level* disjunction of the form `Q1+Q2`. Hence, there is no top-level disjunction that can be expanded. Therefore, we need to make a top-level disjunction.

To make a top-level disjunction, observe that in figure 9-3, there are disjuncts nested under the aggregator that could be used if we could bring them to the top of the **R**-expr. To accomplish this, I will introduce the idea of an *optional constraint*. An **optional constraint** is a constraint that we are allowed to be *ignored* or relaxed. In essence, an optional constraint is something that *must* be true but can be ignored when it is inconvenient (expensive to compute) or can be modified to make it easier to work with.

[121]Admittedly, this is the distributive rewrite which, as already discussed in section §8.2.3.1, is used to *factor* the **R**-expr rather than expanding out the **R**-expr as done here. We are going to have a solution for this in section §11.7.

An optional constraint is denoted with `opt(·)` and is defined using the following semantic definition:

11. $[\![\texttt{opt(R)}]\!]_E = \textbf{if } [\![\texttt{R}]\!]_E = 0 \textbf{ then } (\textbf{nondeterminstic choice of } 1 \textbf{ or } 0) \textbf{ else } 1$

From this semantic definition, we have that in the case of `opt(0)`, we "allow" this **R**-expr to nondeterministically represent the multiplicity of `1` or `0`. The reason we allow this nondeterminism with optional constraints is that optional constraints are *never* used on their own. Instead, they are used in the context of a larger **R**-expr, where regardless of the semantic interpretation used (1 or 0), the larger **R**-expr's semantic interpretation will not change. In other words, if we find `opt(0)` in an **R**-expr, then we know that the larger context will look something like `opt(0)*···*···*0`. Hence, we can rewrite `opt(0)` as `0`, stopping the evaluation early, knowing that some other conjunctive sub-**R**-expr in the **R**-expr would have eventually been rewritten as `0`.

Additionally, note that the **R**-expr `R` wrapped contained in the optional constraint *does not* need to be a constraint, as being a constraint, with a multiplicity of at most 1, is enforced by the optional constraint **R**-expr.

The nondeterministic behavior of the optional constraint can be defined with the following rewrite rules:

$\texttt{R} \xrightarrow{75} \texttt{R*opt(R)}$ ▷*Introduce optional constraint*

$\texttt{opt(R)} \xrightarrow{76} \texttt{if(R,1,0)}$ ▷*make optional constraint, non-optional*

$\texttt{opt(0)} \xrightarrow{77} \texttt{0}$ ▷*Short cut for rewrite rule 76 when zero*

$$\texttt{opt(R)} \xrightarrow{78} \texttt{1}$$ ▷*Ignore optional constraint*

$$\texttt{T*opt(R)} \xrightarrow{79} \texttt{T*opt(T*R)}$$ ▷*Enable rewrites by reading from context*

$$\texttt{opt(R)} \xrightarrow{80} \texttt{opt(R+S)}$$ ▷*Weaken the constraint with any disjunct* `S`

$$\texttt{opt(R)} \xrightarrow{81} \texttt{opt(proj(X,R))}$$ ▷*Weaken the constraint by ignoring any variable* `X`

$$\texttt{opt(R*S)} \xrightarrow{82} \texttt{opt(R)}$$ ▷*Weaken the constraint by removing a conjunct*

Observe that rewrite rules 76 to 78 are for solving the optional constraint. If we have `opt(0)`, we are allowed to apply any of these rewrites nondeterminically. Rewrite rule 78 allows us to rewrite the optional constraint as `1` *before* we know the multiplicity of `R`. This allows us to ignore the optional constraint entirely; hence, it does not affect containing **R**-expr's multiplicity. Rewrite rule 76 turns the optional constraint back into a non-optional constraint using the `if`-expression. The `if`-expression can be removed in the case that `R` is a constraint using rewrite rule 83.

$$\texttt{if(R,1,0)} \xrightarrow{83} \texttt{R} \quad \textbf{if } \texttt{R} \text{ is a constraint}$$

Optional constraints can be introduced from *any* **R**-expr using rewrite rule 75. As we can see in rewrite rule 75, the optional constraint is *conjunctive* with the **R**-expr `R` which *will enforce* the multiplicity of the **R**-expr, meaning the optional constraint is redundant. As a brief reminder of what I said at the beginning of the chapter, the optional constraint does not "actually exist". Rather, optional constraints and rewrites on optional constraints serve as a *license* for complicated sequences of rewrites that we will see in chapter §11. As such, we do not have unproductive rewriting cycles like $\texttt{R}\xrightarrow{75}\texttt{R*opt(R)}\xrightarrow{78}\texttt{R*1}\xrightarrow{13}\texttt{R}$ with optional constraints.

Optional constraints can also "read" from the surrounding context using rewrite rule 79 to copy in the conjunctive **R**-exprT, allowing rewrites to be performed internally—just like the conditional in an `if`-expression with rewrite rule 65.

Finally, optional constraints can be *weakened* using rewrite rules 80 to 82. This means that the constraint will be *true* more often. For example, rewrite rule 80 adds in the additional disjunct `S`, which may be true (nonzero) in cases where `R` is false. Similarly, rewrite rule 81 projects a variable, allowing it to ignore the value that is assigned to some variable. To see an example of weakening an **R**-expr constraint, consider the **R**-expr `int(X)*times(3,X,Y)`, which defines a constraint on both `X` and `Y`, as well as a bag relation on two variables. If we project out the variable `X` from this **R**-expr, we will have `proj(X,int(X)*times(3,X,Y))`. This **R**-expr is *only* a constraint on the variable `Y`. It defines that `Y` *must* be some multiple of the number 3, but otherwise has no influence on the variable `X`.[122]

Given this, we can think of an optional constraint as defining a valid upper bound on the support[123] of an **R**-expr. As such, the optional constraint *never eliminates* something that is true. They allow us to use useful constraints that are nested deep inside of an **R**-expr.

[122]For example, `proj(X,int(X)*times(3,X,Y))*(X="hello")` cannot be rewritten further as the fact that `X` is assigned to the string `"hello"` is not prevented by `proj(X,int(X)*times(3,X,Y))`. However, if we have `int(X)*times(3,X,Y)*(X="hello")`$\rightarrow$`0` this can be rewritten as `0`, as the `int(·)` constraint is conjunctive with `(X="hello")`.

[123]The set of values (named tuples of ground values) which are not mapped to zero.

9.2.1 Example Using Optional Constraints

To see how optional constraints can be used to split the domain of a problem, suppose that we have the following **R**-expr

```
(A=sum(Y,(X=1)*(Y=1)+
         (X=1)*(Y=3)+
         (X=2)*(Y=5)))
```

Figure 9-4. **R**-expr with aggregation over a disjunction

We can create the optional constraint from any **R**-expr in this expression. For this example, we are going to go ahead and create optional constraints from `(X=1)` and `(X=2)` using rewrite rule 75 as this will allow us to illustrate how optional constraints are usually used.

```
(A=sum(Y,(X=1)*(Y=1)*opt((X=1))+
         (X=1)*(Y=3)*opt((X=1))+
         (X=2)*(Y=5)*opt((X=2))))
```

Figure 9-5. Optional constraint introduced for `X`.

In figure 9-5, the optional constraints that were introduced are not too useful at first. We will use rewrite rule 80 to weaken these optional constraints with a disjunction.

```
(A=sum(Y,(X=1)*(Y=1)*opt((X=1)+(X=2))+
         (X=1)*(Y=3)*opt((X=1)+(X=2))+
         (X=2)*(Y=5)*opt((X=2)+(X=1))))
```

Figure 9-6. Optional constraint weakened with a disjunct.

The weakening with the new disjunctive **R**-expr can be done using *any* **R**-expr. However, we have intelligently selected the **R**-expr that we add in this case. Observe that now we have the constraint `opt((X=1)+(X=2))` under all branches of the disjunction. Hence, we can now factor the optional constraint out of the disjunct using rewrite rule 22.[124]

```
(A=sum(Y,opt((X=1)+(X=2))*((X=1)*(Y=1)+
                           (X=1)*(Y=3)+
                           (X=2)*(Y=5))))
```

Figure 9-7. Optional constraints from figure 9-6 factored out of the disjunct.

Because the optional constraint in figure 9-7 does not interact with `Y`, we can further lift it out of the aggregator to the top of the **R**-expr. We can further use rewrite rule 76 and rewrite rule 83[125] to convert the optional constraint back into a normal constraint.

```
((X=1)+(X=2))*(A=sum(Y,((X=1)*(Y=1)+
                        (X=1)*(Y=3)+
                        (X=2)*(Y=5))))
```

Figure 9-8. The optional constraint has been lifted to the top of the **R**-expr and turned into a normal constraint.

[124]Note: Rewrite rule 80 and rewrite rule 22 are usually performed simultaneously, in which case the rewrite looks like `R1*opt(S1)+R2*opt(S2)`$\xrightarrow{84}$`opt(S1+S2)*(R1+R2)`. Rewrite rule 84 was not presented in the text as it is less general than rewrite rule 80.

[125]Note, the **R**-expr `(X=1)+(X=2)` is a constraint as its multiplicity is ≤ 1 in all cases. In general, our implementation assumes that all disjunctions are *not* constraints, but when the **R**-expr is of a special form (such as a list of ground assignments) as we have here, then we can detect that it is a *disjunctive constraint*.

From this point, we use the distributive rewrites on figure 9-8 to expand the **R**-expr and handle each of these cases separately. For this example, we expand this out using the distributive rewrite and have the case where `(X=1)` and `(X=2)`.

```
(X=1)*(A=sum(Y,((X=1)*(Y=1)+
                (X=1)*(Y=3)+
                (X=2)*(Y=5))))+          (X=1)*(Y=4)+
                                   →*
(X=2)*(A=sum(Y,((X=1)*(Y=1)+             (X=2)*(Y=5)
                (X=1)*(Y=3)+
                (X=2)*(Y=5))))
```

Figure 9-9. Aggregator evaluated and rewritten.

Admittedly, the use of the distributive rewrite in figure 9-9 goes against our philosophy of using the distributive rewrite (rewrite rule 22) to split the **R**-expr into `(X=1)` and `(X=2)` cases. As such, we can see that the "*only factor*" philosophy is not sufficient in all cases. In section §11.7, I will continue to build on optional constraints so that an **R**-expr can be split into a disjunction of finitely many smaller **R**-exprs where each will be individually handled.

Solving the SAT formula To solve the SAT formula at the beginning of this chapter (figure 9-2), the system will use the same lift nested boolean assignments trick. The SAT formla contains **R**-exprs of the form `(true=exists(true, (A=1)+(A=0)))`. Using optional constraints, the *constraint* `(A=1)+(A=0)` can be lifted to the top of the **R**-expr, eventually resulting in `(true=exists(true, (A=1)+(A=0)))*((A=1)+(A=0))`. The `(A=1)+(A=0)` can be used to split the **R**-expr, and perform case analysis (propagation of assignments and further branching).

9.2.2 Can Optional Constraints Solve *all* Disjunctive **R**-exprs?

No. Optional constraints must represent a *valid* upper bound on the support of an **R**-expr. They accomplish this by lifting up *existing* constraints through disjunctions and aggregations. Unfortunately, sometimes even the *tightest* upper bound is *useless*. For example, consider the program in figure 9-10.

```
398 a(X) += 1.
399 a(2) += 2.
400 a(3) += 3.
```

(a) Dyna

```
a(X,A) →
  (A=sum(Y,
         1*(Y=1)+
     (X=2)*(Y=2)+
     (X=3)*(Y=3)))
```

(b) R-expr

```
opt(1+
    (X=2)+
    (X=3))
```

(c) Optional constraint on X

Figure 9-10. The lack of a constraint on line 398 prevents us from creating a *useful* optional constraint for the variable X.

Because of line 398, the *upper bound* on the support for values of X is any value. The reason is that without a constraint on X, line 398 always matches. Therefore, any optional constraint derived from a/1 will be always be true, represented as opt(1).[126] If we have an optional constraint that is always true combined with any other optional constraint through a disjunction, then the resulting optional constraint is also going to be always true. In this case, the rewrite sequence is opt(1+⋯) $\xrightarrow{76}$ if(1+⋯, 1, 0) $\xrightarrow{63}$ 1.

[126]Or we could derive opt((Y=1)) from the first line, but when pulling the constraint on Y through the aggregator we would get opt(proj(Y,(Y=1))) $\xrightarrow{41}$ opt(1).

As such, the best we can do in the case of a query `a(X)?` against the program in figure 9-10 would be to return the **R**-expr in figure 9-10 (b).

Chapter 10

Memoization, Reactivity, Cycles, and Updates

This chapter covers how we implement memoization, updates, and cyclic programs with **R**-exprs. For this chapter, I only assume the implementation described in chapter §8. Therefore, I will defer some of the discussion around being *efficient* till section §11.6.2, and instead focus on the fundamental ideals of memoization as it pertains to **R**-exprs and rewriting.

I should note ahead of time that memoization, updates, and cyclic programs were previously studied by Nathaniel Filardo in work on the Dyna project. As such, I recommend that anyone interested in the memoization topic in general should also read Filardo's dissertation [66]. In [66], Filardo developed theoretical algorithms that allowed the values associated with ground terms to be updated in different ways using message passing given a known computation graph. Filardo's work focused on making recomputation efficient and allowed for updates to be processed at different points in the algorithms running.[127] In this chapter, I will instead focus on the details necessary to introduce memoization into an **R**-expr-based rewrite system. The system presented here uses a more complex representation for memos in that we allow **R**-exprs to be memoized vs. Filardo's algorithm, which only supports ground values. Because we are memoizing **R**-exprs, we support memos of non-ground terms and partially evaluated computation (represented as an **R**-expr)—a capability that we have been unable to find elsewhere in the memoization literature. Given that the focus here is on the implementation of memos and **R**-exprs, the kinds of update messages that our system currently supports are much more limited than what Filardo's algorithm supported. We will only use invalidation messages that cause our system to run a recomputation. Hence, the approach for handling updates may be conceptually closer to a simple implementation of reactive programming [14]. Future work may wish to investigate how the different message types supported by Filardo can be integrated into the **R**-expr-based memoization presented here.

Chapter Outline The presentation in this chapter is ordered as follows. I will start by reviewing the necessary background on memoization and its procedural implementation. Then, I will transition to discussing the same abstract ideas of memoization as they apply to **R**-exprs by focusing on "simple" Dyna programs

[127] By this, I mean that Filardo allowed for modifications to the memoized values to be intermixed with when the message is sent about when change is about to happen or has already happened.

without external or internal updates.[128] This means that the result of any computation will not change, allowing us to temporarily ignore the issue of sending invalidation messages (cache invalidation). Once we have developed the necessary background on unchanging Dyna programs with memos, I will introduce the additional mechanisms that are needed to handle external[129] and internal updates (section §10.4). Finally, once we have developed the necessary background on the ideas underlying memoization and updates, I will present further clarifications about how this is *actually* implemented with **R**-exprs (section §10.6), how updates are handled (section §10.7) and how `$memo(·)`[130] is implemented and used to control memoization (section §10.8).

In this chapter, you will notice that the memoization approach presented here is capable of memoizing *any* **R**-expr expression. However, the `$memo(·)` memoization control mechanism defined in section §2.7 is *only* capable of defining memoization policies for user-defined terms. The discrepancy between `$memo(·)` and this chapter is not a mistake. The reason is that controlling the "full power" of memoization mechanisms presented in this chapter is difficult and requires direct manipulations of **R**-exprs. Instead, our implementation of Dyna provides automated manipulations

[128]An internal update happens as a result of a cycle in the program. A cycle is where a value depends on itself. This is a specific kind of recursion that requires that the system solve an equation rather than just expand calls until a base case is reached. This was introduced previously in section §2.5 and will be reintroduced in section §10.4.

[129]Dyna allows for additional rules to be added to be added via the REPL. This can cause memos (cached computation) to become invalidated. Section §2.3

[130]Recall that `$memo` is a user-definable rule which allows for controlling what and how something is memoized. Section §2.7

of **R**-exprs that are limited to adding memoization to user-defined terms and are controlled using `$memo(·)` (these "automated manipulations" will be discussed in section §10.8). Future work may consider extending `$memo(·)` to expose more control or may consider researching automatic control of the memoization mechanisms presented here (section §16.4).

10.1 What is Memoization?

Let us review what memoization is before we jump into how it will work with **R**-exprs. Memoization is a common concept that is taught to virtually all programmers. Hence, most readers will probably already have some preconceived notion of what memoization is. In this section, we are going to *distill* memoization down to a few simple conceptual elements. Then, we will look for those same elements in our **R**-expr rewrite system.

Memoization is a technique to avoid redundant computation by storing in memory a *memo*, which is the result of a computation, and retrieving it later to avoid redoing the same computation. The technique of storing memos was in fact called "Machine Learning" in the original memoization paper by Michie in 1968 [104]. To be able to store and later retrieve a memo, there needs to be a "*signature*" of the computation.[131] Ideally, the signature should be the same every

[131]The signature should equivalence class the computation. Ideally, the signature will also allow for efficient look ups of memoized values (e.g. hash-table). This is not a strict requirement, as a system could (in theory) perform a linear scan over all memoized signatures.

time that the same computation is performed—however, this is not a requirement, in which case a memoized algorithm will redo work and be less efficient but is still "correct". Finally, we need to modify the program to "intercept" relevant calls to the computation with a check to see if an equivalent computation was previously done and intercept the returned result of a computation so that it can be retrieved later.

10.1.1 Example of Memoization in a Procedural Programming Language

I have chosen to start this discussion with an example in Python, a procedural programming language. The reason for this is that it will allow us to discuss the procedural steps required for memoization that we need to replicate with **R**-exprs.

To illustrate this, we start with the Fibonacci program in figure 10-1. The Fibonacci program was chosen as it is an iconic example used when discussing memoization. The Fibonacci sequence is defined as the sequence (computed by the function in figure 10-1) where each number in the sequence is the sum of the previous two numbers: $0, 1, 1, 2, 3, 5, 8, 13, 21, 24, 55, \ldots$. When the previous two numbers in the sequence are available, the next number in the sequence can be efficiently computed in $O(1)$ time. However, if the Fibonacci program is written such that every value is recomputed entirely from scratch each time it is used, then the program will take $\Omega(2^{n/2})$ time[132] to compute the n-th element in the sequence.

[132]This is actually $\Theta(\phi^n)$ where ϕ is the golden ratio.

```
401 def fib(N):
402   if N == 0:
403     return 0
404   elif N == 1:
405     return 1
406   else:
407     return fib(N-1) + fib(N-2)
```

Figure 10-1. Fibonacci program written in Python so that we can discuss the procedural processes of how memoization is implemented. On line 407, the Fibonacci function calls itself twice. Hence, if we run Fibonacci as written here, it will take $\Omega(2^{n/2})$ time to compute the result.

To add memoization to the Python `fib` function (line 401), we first need to identify the "*signature*", which sufficiently identifies the computation to be performed. In this case, we can use the argument for the `fib` function `N`. Note: The arguments to the function are sufficient in the case that the function is a *pure function*[133], as here, meaning that the result of the function is entirely determined by its arguments. In general, Python does not *require* that functions are pure, so the arguments might not be a sufficient signature, and it is a Python programmer's responsibility

[133]The term *pure function* is a programming language term which would generally just be called a *function* in a mathematical setting. The returned result from a pure function is entirely determined by its arguments, which means that it returns the same value every time it is invoked and that the function is side-effect-free. Meaning that there is no other external observable behavior, such as modifying a global or class variable.

to ensure that a memoized function is pure.[134] Dyna automatically tracks the necessary dependencies and changes for a memo without having to concern the Dyna developer further.

Now that we have identified the signature of the Fibonacci function as its argument, we need to create some "global" storage associated with the function where the results of the computation will be stored.

Finally, we need to check upon being called if the requested computation was previously performed and retrieve it from storage. Or, if the computation was not previously performed, do the computation and save it to storage for later use.

Adding this to the Python program from figure 10-1, we get the program shown in figure 10-2.

[134] An example of an impure function would be something that modifies global state. This can either be explicit, by mutating some global state, or implicitly by depending on the result of something like a random number generator (which maintains global state internally). E.g.

```
x = 0
def not_pure():
  global x
  x += 1
  return x
```

```
408 def fib_original(N):
409   if N == 0:
410     return 0
411   elif N == 1:
412     return 1
413   else:
414     return fib(N-1) + fib(N-2)
415
416 memoized_values = {}
417 def fib(N_signature):  # override fib, so it is now memoized
418   if N_signature in memoized_values:
419     return memoized_values[N_signature]
420   r = fib_original(N_signature)
421   memoized_values[N_signature] = r
422   return r
```

Figure 10-2. The Fibonacci program in Python with memoization manually added by the programmer. On line 416, a global hash map is created that is used to hold the memoized values. The hash map is checked at the top of the memoized `fib` function on line 418 before the original `fib_original` is called (performing computation). In the original `fib_original` function, the memoized version of the function is called on line 414 as we have overridden the `fib` function. After the original `fib_original` function performs the computation, the result is saved by the memoizing version of `fib` on line 421.[135]

10.1.2 The Facets of Memoization

Taking a step back from the Fibonacci example, we can see that there are four things that we need to identify when designing memoization for **R**-exprs:

1. A location (in memory) for a data structure to store a memo (short for mem-

[135]Here, I have shown the explicit transformation of how memoization is added to a Python function. Python does provide a decorator that can perform this transformation automatically, using only one additional line of code. `https://docs.python.org/3/library/functools.html#functools.cache`

oized result of computation). This can either be a data structure that exists globally for the entire running of the program (as in figure 10-2), or a data structure that only exists within the context of a single computation (e.g. within the context of a single query handled by SimplifyNormalize).

2. A signature to identify the computation requested. The signature must be sufficient to identify the computation, otherwise the memo will be incorrect.[134] The signature should occur many times, otherwise there will be no advantage when memoizing the computation under an ineffective signature. Usually, the signature is the *arguments* to a function; however, a *backed off* version of the arguments could also be used. For example, if we have the arguments $x = -3$, then we could choose to memoize for the negative interval $x < 0$, as a backed-off version (section §10.2.3).

3. A way to intercept the calls to function and identify the signature of what computation is requested. This is done to check the data structure that contains the stored results before attempting to perform further computation.

4. A way to store the result of a computation so that it can be reused later. Generally, in a language like Python, the result is the value returned from a function (as on line 421); however, as we will see, this can be generalized with **R**-exprs.

5. A way to control what is memoized. In the case of Python, this is how the

programmer has modified their program to enable memoization (as in figure 10-2). In Dyna, the "program" itself does not require any changes. Instead, we will control memoization using `$memo`.

By the end of this chapter, we will be able to identify these same elements (and more) in **R**-exprs-based memoization.

10.2 First Steps Towards R-expr Memoization

Given the requirements of memoization in section §10.1.2, let us identify where these requirements exist in the context of **R**-exprs. To keep things "simple" in this section, I will only focus on creating simple memos, and I will defer modifications, cyclic programs, internal & external updates, and change propagation until section §10.4.

Now, the goal of memoization is to avoid redoing computation. Computation, in the **R**-expr setting, are rewrites performed by SIMPLIFY(R, $\mathcal{C}$) against an **R**-expr, to get another semantically equivalent **R**-expr.

As such, a first attempt would be to apply memoization to the SIMPLIFY(R, $\mathcal{C}$) function. Unfortunately, memoizing SIMPLIFY does not work that well. The reason is that the arguments for SIMPLIFY are both a *large* **R**-expr R and the entire context $\mathcal{C}$, which contains many irrelevant details. Therefore, a signature match against R and $\mathcal{C}$ would be inefficient and ineffective. Furthermore (and more importantly),

```
m(X) = 2*X + 3.
```

(a) Simple Dyna rule.

```
m(X,Res) →                              ▷user-def' with memo
  if((X=1)+(X=2),                       ▷Disjunction of signatures
     ((X=1)*(Res=5)+                    ▷The memos[136]
      (X=2)*(Res=7) ),
     (Res=only(Inp,                     ▷Original user-defined R-expr
      proj(Tmp,
        times(2,X,Tmp)*plus(Tmp,3,Inp)))))
```

(b) R-expr with a memo for 1 and 2

Figure 10-3. Simple **R**-expr with the memoized values for 1 and 2 stored *inside* of the **R**-expr. `X` is the argument and `Res` is the returned result of the user-defined **R**-expr.

memoizing Simplify does not work with handling internal or external updates, as will be discussed in section §10.5.3.1.

Rather, we are going to create memos for sub-**R**-exprs using an approach based on the previously introduced `if`-expression kind **R**-expr (section §5.2.2.8). Recall that given any **R**-expr `R`, for which we want to create memos for, the system can use rewrite rule 66 to introduce an `if`-expression as `R`$\xrightarrow{66}$`if(S,R,R)` (as the same **R**-expr `R` is returned regardless of the `S` that the system picks). As we will see, `S` is the signature of the memo. I will get into details of how `S` is selected later, but for now, we will just assume that `S` is selected to be "useful for memoization".

Once we have this **R**-expr, the system can use rewrite rule 67 to rewrite the true branch as `if(S,R,R)`$\xrightarrow{67}$ `if(S,S*R,R)` provided that `S` is a constraint. We can

[136]The structure of `Memo` here is shown as a generic **R**-expr. In practice, we might want to ensure that the **R**-expr is represented as a hash-map so that it is efficient. Efficient disjunctive **R**-expr kinds will be discussed in section §11.6.1.

perform `S*R` $\rightarrow^*$ `S*RMemo` using SIMPLIFY.[137]

The **R**-expr now has a structure like `if(S,S*Memo,R)`. A more complete example of `if(S,S*Memo,R)` **R**-expr is shown in figure 10-3. Let us look at this **R**-expr and check which facets of memoization are satisfied.

First, memoization requires that there is *some* location in memory where the result of a computation is stored. Here, `RMemo` is the result of SIMPLIFY applied to `R` in the context of `S`. Additionally, `RMemo` is stored inside of the **R**-expr `if(S,S*RMemo,R)`.

Second, there needs to be a *signature* of the computation. The signature needs to be checked and return the memo in the case it matches, or bypass the memo and perform computation in the case the memo does not exist. Here, `S` is the signature of the computation. To see that `S` is indeed the memo's signature, let us consider the third requirement of memoization, that we can intercept "calls" to compute and instead return the memoized result. The way in which we perform computation with **R**-exprs is by rewriting an **R**-expr with SIMPLIFY in the context of other conjunctive **R**-exprs tracked via the context $\mathcal{C}$. For example, suppose that we have the **R**-expr `R`, which is being rewritten in the context of `Q`. Hence, the system is rewriting the **R**-expr `Q*R`.[138] Now, let us consider a version `R` with a memo. The **R**-expr `R` with a memo has the form `if(S,S*RMemo,R)` (where the `if`-expression was introduced using rewrite rules 66 and 67). Now, when rewriting `if(S,S*RMemo,R)` in the context

[137]The `RMemo` **R**-expr in this case might be better thought of as a "bulk" computed memo rather than an individual memo. This, of course, depends on the computation specified by `S`.

[138]SIMPLIFY `R` in the context of `Q` is equivalent to making the SIMPLIFY call SIMPLIFY(R, $\mathcal{C} = \{Q\}$).

of Q, the system is rewriting the **R**-expr `Q*if(S,S*RMemo,R)`. Using rewrite rule 65, the `if`-expression's conditional is allowed to "read" the context Q, resulting in the **R**-expr `Q*if(Q*S,S*RMemo,R)`. If the signature S *matches* the context, then there will exist a sequence of rewrites such that $\texttt{Q*S} \to^* \texttt{Q*(1+T)}$ for some T.[139] This means that the *true*-branch of `Q*if(Q*S,S*RMemo,R)` will be returned ($\texttt{Q*if(Q*(1+T),S*RMemo,R)} \xrightarrow{65,63} \texttt{Q*S*RMemo}$). Hence, we have intercepted the "call" to R and substituted in the memo. Retrieving the memo was previously called LOOKUP in Filardo's work [66, 67].

Similarly, if Q is not memoized, then $\texttt{Q*S} \to^* \texttt{0}$, causing the *false*-branch of the `if`-expression to be returned: $\texttt{Q*if(Q*0,S*RMemo,R)} \xrightarrow{64} \texttt{Q*R}$. This corresponds with falling back to the original computation (known as COMPUTE in Filardo's work [66, 67]).

This way in which this **R**-expr-based `if`-expression works can be seen as conceptually the same as the `if`-expression in the Python program (line 418).[140] When the signature in the Python program matches, the true branch, which reads from the hash-table, is used. If the signature fails to match, then the false branch is used, which falls back to the original definition. However, by representing the signature S and the query Q as **R**-exprs, we can memoize non-ground relations and condition

[139] In SIMPLIFY the system does not *explicitly* copy Q into the conditional. Instead, it depends on Q by including it in the context. Hence, the system will instead check that SIMPLIFY(S, $\mathcal{C} = \{$Q$\}$) returns an **R**-expr that matches 1+T for some T.

[140] Additionally, note that accessing the memo contained in the `if`-expression required *no* modifications to SIMPLIFY. In this way, SIMPLIFY is comparable to a bytecode interpreter, and the addition of the `if`-expression to the **R**-expr can be considered a modification to the program, just like how we modified the Python program (not the Python interpreter).

memos on general expressions rather than ground terms used as keys for a memo table.

10.2.1 Advantages of Homogeneity

The advantage of a homogeneous system which represents all state as **R**-exprs shows through in our representation of memos.

For example, in the procedural Fibonacci example (section §10.1.1), the signature of a memo was the argument to the function—which is an integer value like the number 7. Using a value like this is quite typical of memoization. However, with our **R**-expr `if`-expression representation, we can further generalize memos to use *any executable* **R**-*expr* as the signature. For example, we can choose to memoize all negative values by using `lessthan(X,0)` as a signature. Admittedly, a programmer could *manually* introduce the relevant test and if-expression in another programming language. However, the **R**-expr backed formalism allows us to do this *automatically*.

A second advantage of homogeneity is that the result returned from the memo is an **R**-expr. This means that we are not limited to memoizing values, but instead, we can memoize *partially completed* computation. For example, suppose that we have the `sum3` rule defined in figure 10-4 that sums up its three arguments.

```
sum3(A,B,C) = A+B+C.
```

(a) Sum3 in Dyna

```
proj(Tmp, plus(A,B,Tmp)*plus(Tmp,C,Result))
```

(b) Body of Sum3 as an **R**-expr

Figure 10-4. Rule defining sum of its 3 arguments.

In memoizing `sum3` with only 2 of its three arguments (say `A` and `B`), we will still have one `plus` **R**-expr remaining. The memo can memoize the intermediate sum of `A+B` but must still represent the remaining `plus` as shown in figure 10-5.

```
if((A=1)*(B=2), (A=1)*(B=2)*plus(3, C, Result),
                 proj(Tmp, plus(A,B,Tmp)*plus(Tmp,C,Result)))
```

Figure 10-5. `sum3` with a memo created where only two of its arguments (`A`,`B`) are known. Computation to compute $1+2=3$ was performed, leaving an **R**-expr which represents the computation `3+C` that still needs to be completed. Admittedly, in this example, we are not saving that much "work" by memoizing $1+2$, however, in general, we would memoize a computation that takes more time to complete.

10.2.2 Persisting Memos to Make them Globally Usable

In our design, so far, a partial memo is represented as another **R**-expr, namely `R`$\xrightarrow{66,67}$`if(S,S*R,R)`. This is a very general approach to creating and storing a memo, as there always exists *some* **R**-expr, and therefore the system can create a memo *whenever it wants*. However, "whenever it wants" is a double-edged sword, in that some points where a memo could be created may be useless. For example, suppose that a memo is created inside of an **R**-expr that was created to evaluate a user's query (section §2.3). In this case, the memo is usable only within the context of that query. The **R**-expr with the memo does not persist within the Dyna system after the query has been completed.

Ideally, a memo should be usable throughout the entire running of the program—meaning that it should persist between queries initiated by the user. This can be accomplished by carefully choosing *when* and *where* a memo is constructed. Recall from chapter §7 that the user's program is defined in terms of user-defined named **R**-exprs, recreated in figure 10-6 for convenience.

```
425 | f(X) += X*X.
```

(a) Dyna

$f(X,Res) \xrightarrow{74}$ `(Res=sum(Inp,`
`  times(X,X,Inp)))`

(b) R-expr

Figure 10-6. Recall how Dyna is translated to **R**-exprs, chapter §7.

Every time that a user-defined **R**-expr is referenced, it indirects through rewrite rule 74. This means that if we modify rewrite rule 74, then the resulting **R**-expr will be accessible *globally*. The resulting **R**-exprs will look something like figure 10-7.

f_original(X,Res) $\rightarrow$ (Res=sum(Inp,
  times(X,X,Inp))) ▷*Original* ***R****-expr (like* `fib_original` *in figure 10-2)*

f(X, Res) $\xrightarrow{74}$ if((X=3), ▷*Disjunction of all signatures*
  (X=3)*(Res=9), ▷*Memo*
  f_original(X,Res)) ▷*Call to original definition*

Figure 10-7. The user-define `f` **R**-expr with a memo for the value of `3`.

10.2.3 Things to Consider When Choosing a Signature for Memoization

So far, we have that the memo is contained inside of the **R**-expr that user-defined **R**-exprs are rewritten as. However, we need to discuss *how* the system determines what to memoize. More concretely, we need to pick the signature `S` used by rewrite rule 66 when introducing an `if`-expression (`R`$\xrightarrow{66,67}$`if(S,S*R,R)`). The common approach we saw with the Python example in section §10.1.1 is to wait until the function is called with an unmemoized query and then use the value that was used to check the memo table as the signature (as in figure 10-2 lines 418, 420 and 421). However, the Python program hard coded that the signature was the argument of the Fibonacci function. Whereas with **R**-exprs, we have more flexibility in choosing the signature as we are allowed to choose *any* **R**-expr, and we do not have to commit to a particular `S` until just before the memo is created by rewrite rule 66. Further complicating this, the choice of `S` will have a significant impact on the usefulness and efficiency of our memo.

As an example of choosing a signature, suppose that we are performing a query `Q` against the **R**-expr `R`. If the system chooses `S` as having no relation to `Q`, then it is likely that `S` and `Q` will be incompatible (e.g. `Q*S`$\to^*$ `0`). This means that spending time constructing a memo using a bad signature `S` would actually *delay* the computation with useless rewrites: `Q*R`$\to$`Q*if(S,S*R,R)`$\to$ `Q*if(S,S*RMemo,R)`$\to$`Q*R`.

This gives us our first hint of how to pick a signature. A useful signature must have a non-empty intersection with the query `Q`.

A second thing to consider when choosing a signature is the amount of *reuse* we will get from a memo. A signature that is *more general* can be used in more cases. For example, suppose that we are making the query `lessthan(5,W)*(X=1)*(Y=2)*(Z=3)` against the **R**-expr `R`. We can choose to use the signature `(X=1)*(Z=3)`, ignoring `lessthan(5,W)*(Y=2)`. This will mean that we get an **R**-expr with a memo like `if((X=1)*(Z=3),(X=1)*(Z=3)*RMemo,R)`. Now, if we get a second query `(X=1)*(Z=3)*(Y=7)` against this **R**-expr, we can reuse `RMemo` as the signature `(X=1)*(Z=3)` matches this query.

A third consideration when choosing the signature `S` is the data structure that will be used to store the memo. For example, ground assignments to variables (e.g. `(X=1)*R1+(X=7)*R2`), can be represented efficiently using a hash table (section §11.6.1.2). Whereas a signature like `int(X)*lessthan(0,X)*lessthan(X,10)+int(X)*between(X,15,18)+` $\cdots$ `+int(X)*lessthan(321,X)*lessthan(X,325)` requires that the system scan through all of the `lessthan` constraints to check if any of them match the current query.

Finally, I note that when selecting the signature `S`, the system has access to the context $\mathcal{C}$ for the current query. This means that the system can efficiently retrieve conjunctive **R**-exprs when accessing the memo. Hence, the signature `S` will be chosen as a relevant subset of $\mathcal{C}$ that satisfies the considerations listed above.

10.3 Example: Memoization of an **R**-expr

Let us use the representation in memoization discussed so far to work through an example on the Fibonacci program, this time represented as a Dyna program and an **R**-expr in figure 10-8. Note that the **R**-expr here is semantically equivalent to the Dyna program, though not *identical* to the **R**-expr that would result from the mechanical translation of Dyna to **R**-exprs (as defined in chapter §7).

```
fib(0) += 0.
fib(1) += 1.
fib(N) += fib(N-1)
        for N > 1.
fib(N) += fib(N-2)
        for N > 1.
```

(a) Dyna

```
fib(N, Res) → (Res=sum(Inp,
  (N=0)*(Inp=0)+
  (N=1)*(Inp=1)+
  proj(Tmp, lessthan(1,N)*plus(Tmp,1,N)*
            fib(Tmp,Inp))+
  proj(Tmp, lessthan(1,N)*plus(Tmp,2,N)*
            fib(Tmp,Inp))))
```

(b) **R**-expr

Figure 10-8. Fibonacci program shown as a Dyna program and **R**-expr.

First, suppose that a query is initiated against the `fib` **R**-expr. For example, this might be `fib(5,Res)` where the system is computing the Fibonacci value for the number 5. The Fibonacci **R**-expr will be simplified by expanding the user-defined sub-**R**-expr until it reaches the base cases of `(N=0)` and `(N=1)`. At this point, the system can identify `(N=0)` and `(N=1)` as two queries made against the Fibonacci relation. Both are combined into a signature to indicate that the system will memoize the values for both `(N=0)` and `(N=1)`. The resulting **R**-expr is shown in figure 10-9:

```
fib(N, Res) → if((N=0)+(N=1),
  ((N=0)*(Res=0)+                                    ▷The memo
   (N=1)*(Res=1) ),
  (Res=sum(Inp,                                      ▷The original R-expr
    (N=0)*(Inp=0)+
    (N=1)*(Inp=1)+
    proj(Tmp, lessthan(1,N)*plus(Tmp,1,N)*
              fib(Tmp,Inp))+
    proj(Tmp, lessthan(1,N)*plus(Tmp,2,N)*
              fib(Tmp,Inp)))))
```

Figure 10-9. Fib **R**-expr with memo for 0 and 1

Next, the value for `fib(2,Res)` can be computed and stored. The current disjunction of the signatures `(N=0)+(N=1)` indicates that the `(N=2)` value is not currently memoized,[141] and therefore will fall back to the original **R**-expr. A new memoizing `if`-expression is nested under the *false*-branch of the original memo. This is consistent with the idea that we are *creating* memos on any **R**-expr, rather than *modifying* the existing memo. The resulting **R**-expr is shown in figure 10-10.

[141]We can check the signature by performing the rewrite sequence for `((N=0)+(N=1))*(N=2)` $\to^*$ `0`.

```
fib(N, Res) → if((N=0)+(N=1),
  ((N=0)*(Res=0)+                                        ▷The previous memos
   (N=1)*(Res=1) ),
  if((N=2),              ▷The new memo nested under the false-branch of the first memo
    (N=2)*(Res=1),
    (Res=sum(Inp,        ▷The original R-expr deeply nested under the false-branches of all
   memos
      (N=0)*(Inp=0)+
      (N=1)*(Inp=1)+
      proj(Tmp, lessthan(1,N)*plus(Tmp,1,N)*
                fib(Tmp,Inp))+
      proj(Tmp, lessthan(1,N)*plus(Tmp,2,N)*
                fib(Tmp,Inp))))))
```

Figure 10-10. Fib **R**-expr with a new memo for `(N=2)`, which was introduced using rewrite rules 66 and 67. The newly added memo is under a *second* `if`-expression, and does not modify the previously created memo for `(N=0)+(N=1)`.

Observe that falling through several `if`-expressions will be inefficient when there are a large number of memoized **R**-exprs and **R**-expr signatures. Therefore, the system can use rewrite rule 69 to merge the `if`-expressions together, which avoids creating a nested chain of `if`-expressions:

```
fib(N, Res) → if((N=0)+(N=1)+(N=2),
  ((N=0)*(Res=0)+                                    ▷The memos (now combined)
   (N=1)*(Res=1)+
   (N=2)*(Res=2) ),
  (Res=sum(Inp,                                             ▷The original R-expr
    (N=0)*(Inp=0)+
    (N=1)*(Inp=1)+
    proj(Tmp, lessthan(1,N)*plus(Tmp,1,N)*
              fib(Tmp,Inp))+
    proj(Tmp, lessthan(1,N)*plus(Tmp,2,N)*
              fib(Tmp,Inp)))))
```

Figure 10-11. Fibonacci with `if`-expressions merged using rewrite rule 69

This process of creating nested `if`-expression for new memos and merging the `if`-expressions will continue all of the way up to `fib(5,Res)`, which was initially requested by the user's query. Once the query `fib(5,Res)` has been answered, the user-defined Fibonacci **R**-expr `fib(N,Res)` will only contain the memos for $0, 1, 2, 3, 4, 5$, and the system will stop modifying the definition `fib(N,Res)`.

Throughout this entire process, the rewrite that defines the user-defined `fib(N,Res)` was having its right-hand side modified. However, the semantic interpretation of `fib(N,Res)` *never* changed. This is because all of the rewrites performed against `fib`'s definition are semantics preserving. The only difference is that we are now clever about where and when we apply the rewrite rules from chapter §6.

10.3.1 Conclusion of Basic Memos

So far, we have developed a sufficient understanding to implement memoization in the case of unchanging programs. The `if`-expression can represent the memos *inside* of **R**-exprs. If we think of the **R**-expr itself as representing the program, then modifications of the **R**-expr with the `if`-expression are conceptually equivalent to the modifications done to the Python program when adding memoization to the Fibonacci function (figure 10-2).[140] With **R**-exprs, the *true*-branch of the `if`-expression represents the memo itself, the signature is used to indicate *what* is memoized, and the *false*-branch is used to represent falling back perform a computation using the original definition of a user-defined **R**-expr.

10.4 Handling Change

So far, we have focused only on programs where the result does not change, ignoring issues such as updates (cache invalidation). However, Dyna allows the rules to change. This can be the result of externally driven updates (such as new rules added at the REPL), or due to cycles in the program iterating until a fixed point (section §2.5, e.g. figure 10-12a). To properly handle this, we must track any memo that depends on a value that has changed and needs to be updated or invalidated. We accomplish this by adapting a design similar to that of other reactive programming languages and libraries [14].

```
e += 1.
e += e/2.
print e. % prints 2
```

(a) A cyclic Dyna program which converge to e=2.

```
435 a += 1.
436 print a.  % prints 1 (query)
437 a += 2.   % external update
438 print a.  % prints 3
```

(b) Dyna program with an update (line 437) which modifies the program.

Figure 10-12. Example Dyna Programs which demonstrate rules (e and a) can change. Therefore, any downstream memoized values must be updated accordingly.

```
final class Assumption {
  private boolean isValid = true;                          // Starts valid
  private Set<MessageListener> subscribers;
  public void subscribe(MessageListener);   // Dependents are subscribed
  public void sendMessageToAll(Message);      // Send to all subscribers
  public boolean getIsValid() { return isValid; }
  public void invalidate() { isValid = false;     // Can only invalidate
                             sendMessageToAll(···); }
}
```

Figure 10-13. Class design for an Assumption

10.4.1 Assuming Reads Never Change

First, we have that *all* values and expressions in the language which *can* change are protected by an *assumption*[142] that the values have not changed. Some examples of changeable values include all the **R**-expr definitions of user-defined terms and the mutable memo tables (which I will define shortly).

An **assumption** (figure 10-13) is an object within the Dyna implementation that tracks the immediate downstream dependencies of a particular value. When a

[142]We have adopted the terminology of *assumption* from the JIT compiler literature [149].

value is changed, its associated assumption is invalidated and will be replaced with a new assumption for the new value. An invalidated assumption sends notification messages to all downstream dependencies that it has been invalidated. An invalid assumption can never become valid again. This design was made with the intention that future work on parallel/concurrent processing can take advantage of this design to avoid race conditions.

When computation is performed in the Dyna implementation, assumptions that protect read values will be automatically subscribed to.[143] Downstream dependencies are allowed to run "any code they want" when they receive a notification that an upstream assumption has become invalidated. This allows us to integrate assumptions and invalidation messages into many different aspects of Dyna implementation outside of memoization. Currently, assumptions are only used to trigger recomputation of memos, however future work may which to depend on this mechanism.[144]

Note: Memoized values can be subscribed to their own assumption. This happens in the case of a cyclic program as in figure 10-12a. This is not a problem as long as

[143]Note: In a (future) parallel processing environment, it is possible that an invalid assumption to be "depended" on. This means that the computation that depends on an invalid assumption is already "stale" when it is computed. Most likely, it should be immediately thrown out and recomputed, as there is little use for a stale value.

[144]Future work has considered *streaming queries*, which are queries made against the Dyna program that are updated as the program changes rather than getting back a single value. Streaming queries can be implemented by subscribing to assumptions with a custom subscriber class that listens to invalidation messages.

the memoized **R**-expr eventually *converges.*[145]

10.4.1.1 Updating Memoized R-exprs

The most common message that a subscriber will receive is a notification that an upstream **R**-expr has changed and its corresponding assumption has been invalidated.[146] In this case, this means that there is an **R**-expr like `if(S,RMemoExisting, R)` and the system needs to recompute `RMemoExisting`. This is done by starting again from the original **R**-expr definition `R`, and combining it with the memo's existing signature `S` to get `R*S`. The system rewrites `R*S` using SimplifyNormalize, as we did before, and will get `S*R` $\to^*$ `RMemoNew`. In the process of creating `RMemoNew`, all reads performed, and assumptions depended on will be tracked. Finally, `RMemoNew` and `RMemoExisting` are compared with each other for "*semantic equivalence*".[147] If the system can *prove* that `RMemoNew` and `RMemoExisting` are semantically equivalent, then we do not have to invalidate the assumption associated with the memo `RMemoExisting` and can stop propagating update messages forward. In the case that the system *cannot* prove that `RMemoNew` and `RMemoExisting` are semantically

[145]If the memoized **R**-expr does not converge, e.g. `a = !a.`, then the memo not converging and running forever is consistent with the semantics of Dyna, section §2.5.

[146]This can either be an **R**-expr, which was user-defined and changed via an external update, or a change resulting from a memo being recomputed.

[147]Proving two **R**-exprs are semantically equivalent is Turing-complete, hence can be difficult to check. We allow for *one-sided error* in that the system must *prove* that two **R**-exprs are equivalent when it returns that two **R**-exprs are equivalent. However, it is not required to prove that **R**-exprs are not equivalent. Hence, it may overestimate that two **R**-exprs are not equivalent. We handled this by having a procedure that attempts to determine semantic equivalence in the case of minor reordering of the **R**-expr and renaming of variables. Section §12.4.2 also discusses checking **R**-exprs for semantic equivalence in the context of compilation.

equivalent, it *must* replace `RMemoExisting` with `RMemoNew`, and *then* invalidate the assumption associated with `RMemoExisting`.

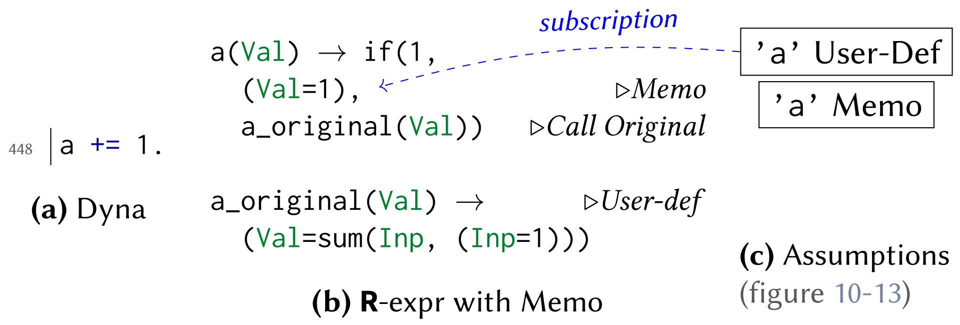
This process of receiving messages about upstream assumptions being invalidation and recomputing `RMemo` will continue until the system converges[148] and all messages have been processed.

10.4.2 Example: Updating a Dyna Program

To see how assumptions work in the context of a program receiving updates, let us work through the example given in figure 10-12b.

First, we have the initial **R**-expr for 'a', and the system will have created a memo using the `if`-expression around the original **R**-expr. Furthermore, I have shown the valid assumptions as black boxes and the subscription from the assumptions as a blue dashed arrow in figure 10-14.

[148]It is up to the user to write Dyna programs that converge. The Dyna system does not provide any automatic detection (or restrictions) that ensures that the programs converge (section §2.5).



```
a += 1.
```

(a) Dyna

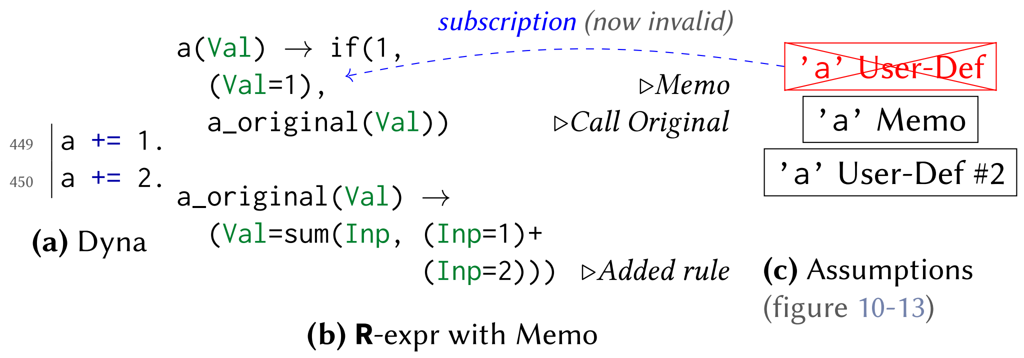
```
a(Val) → if(1,
  (Val=1),                      ▷Memo
  a_original(Val))     ▷Call Original

a_original(Val) →         ▷User-def
  (Val=sum(Inp, (Inp=1)))
```

(b) R-expr with Memo

(c) Assumptions (figure 10-13)

Figure 10-14. A memo created for the Dyna rule a. Assumptions are shown as boxes on the right-hand side, and the subscription to the "*definition of* 'a'" is shown as a blue dashed line. In this example, the signature is *always* the **R**-expr 1, meaning that the entire relation is memoized.

When the definition of 'a' is updated, the original **R**-expr is modified, and the assumption that is associated with the definition of 'a' is invalidated. This is shown in figure 10-15.



```
a += 1.
a += 2.
```

(a) Dyna

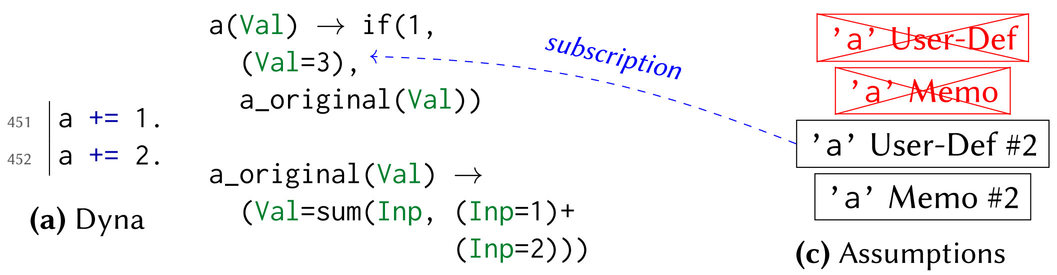
```
a(Val) → if(1,
  (Val=1),                      ▷Memo
  a_original(Val))     ▷Call Original

a_original(Val) →
  (Val=sum(Inp, (Inp=1)+
                (Inp=2)))   ▷Added rule
```

(b) R-expr with Memo

(c) Assumptions (figure 10-13)

Figure 10-15. The original **R**-expr (Dyna program) has been modified. The assumption that is associated with the user's definition has been invalidated, and the subscription to the memo needs to be processed.

This will trigger the memo to be recomputed and brought up to date with the new value as defined by the **R**-expr constructed from the user's program as shown in figure 10-16.



```
a += 1.
a += 2.
```

(a) Dyna

```
a(Val) → if(1,
  (Val=3),
  a_original(Val))

a_original(Val) →
  (Val=sum(Inp, (Inp=1)+
                (Inp=2)))
```

(b) **R**-expr with Memo

(c) Assumptions (figure 10-13)

Figure 10-16. The memoized value has been updated to `Val=3` by recomputing using the original **R**-expr. The Memo now depends on the new valid assumption "'a' User-Def #2". The old assumption for the memo has been invalidated and replaced with a new assumption. Anything that was subscribed to the old assumption will have received a notification of its invalidation.

10.5 Handling Cyclic Programs

So far, in section §10.4, I have glossed over the details of how the computation for creating a memo is performed. In general, SimplifyNormalize (algorithm 1) is invoked on the original user-defined **R**-expr in conjunction with a query, and the resulting **R**-expr is saved as the memo. With a non-cyclic program, we are guaranteed that the fully expanded **R**-expr is bounded in size. Hence, SimplifyNormalize is able to fully rewrite the **R**-expr.

Unfortunately, the same guarantee does not exist with cyclic programs. A cyclic program recurses back onto itself. For example, the definition of 'e' in figure 10-17 is cyclic as the term '`e()`' depends on the term '`e()`'. Conversely, the Fibonacci

```
453 e += 1.
454 e += e/2.
```

(a) Dyna

```
e(Val) → (Val=sum(Inp,
  (Inp=1)+
  proj(Tmp, e(Tmp)*times(Tmp, Inp, 2))))
```

(b) R-expr without Memos

Figure 10-17. The equation $e = 1 + \frac{e}{2}$ is represented as an **R**-expr on lines 453 and 454. The rule 'e' depends on what is in both line 454, and in the **R**-expr, with `e(Val)` appearing on both the left and right-hand side of the rewrite rule.

program from before is recursive, not cyclic, as *terms* in the computation graph do not depend on themselves, rather terms with the name `fib(·)` depend on *other terms* which also have the name `fib(·)`.

When rewriting a cyclic **R**-expr, SIMPLIFYNORMALIZE will never terminate. The reason is that the **R**-expr is expanded at every invocation of SIMPLIFY to increasing depths of the recursion.

10.5.1 Making Guesses

To avoid this issue, we must avoid expanding the same recursive call when an **R**-expr is rewritten as depending on itself as a sub-**R**-expr, as in figure 10-17 with `e(Val)`. This is accomplished by placing a "marker" along the recursive calls to avoid performing the same query against a memo twice. This is akin to setting and checking a VISITED bit in a depth-first search algorithm.

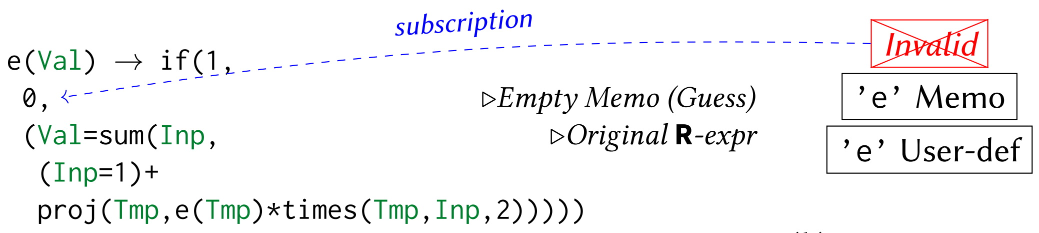
Like in the depth-first search, when the query is re-encountered, the system avoids redoing the exact same computation by *guessing* the **R**-expr result to a query.

A **guess** is a memo that is created *before* any computation is performed. A guess can be *any* **R**-expr, though in practice, we always guess an empty bag, hence the **R**-expr `0`. A guess initially depends on an *invalid* assumption; therefore, it must be recomputed/checked immediately after it is created. Checking a guess is crucial as it enforces the self-consistency of a value on a cycle. Furthermore, the process of guessing and then forward-chaining invalidation messages until convergence is *equivalent* to Datalog-style forward chaining, though with **R**-exprs rather than ground values (section §3.1.2).

I should note that guessing in the case of cycles is not novel to this dissertation and has appeared in prior work that influenced our design. First, XSB-Prolog [135, 145, 151] added memoization to standard Prolog style back-chaining by detection with marking. When a cycle is found, a guess is made. Filardo and Eisner [67] focused their work on finite circuits, which can contain cycles and allow guessing and forward-chaining messages to update memoized values (also see Filardo's dissertation [66]).

10.5.1.1 Example: Using Guessing with a Cyclic Program

To see how guessing works, let us work through an example of 'e' defined in figure 10-17. In figure 10-18, I have written the **R**-expr with an initial guess. The initial guess for 'e' is `0`, and the memo is subscribed to the *invalid* assumption. We also already have the guess protected by its own assumption "e Memo".



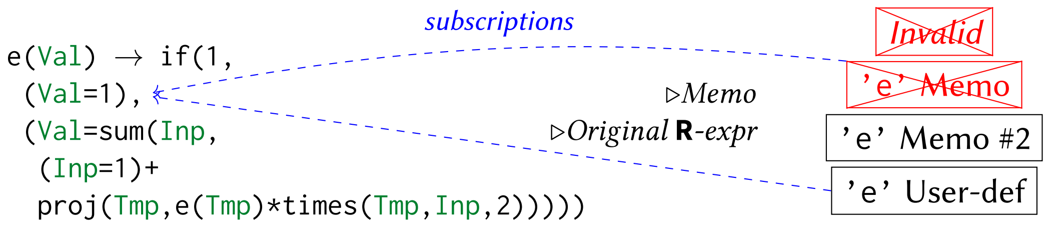
```
e(Val) → if(1,
 0,
 (Val=sum(Inp,
  (Inp=1)+
  proj(Tmp,e(Tmp)*times(Tmp,Inp,2)))))
```

(a) R-expr

(b) Assumptions (figure 10-13)

Figure 10-18. Initial memo being created as guess of `0`. The guess is subscribed to an invalid assumption, so the memo must be validated by recomputing the memo.

In the process of recomputing the memo for 'e' now, the recursive call to `e(Tmp)` is rewritten as `0` (as that is the memo). This means that only the contribution from the `(Inp=1)` (line 453) branch will be counted. Hence, the new memoized value is `(Val=1)`, as shown in figure 10-19.



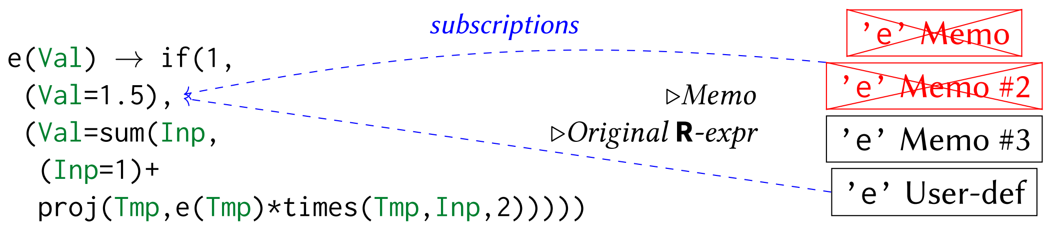
```
e(Val) → if(1,
 (Val=1),
 (Val=sum(Inp,
  (Inp=1)+
  proj(Tmp,e(Tmp)*times(Tmp,Inp,2)))))
```

(a) R-expr

(b) Assumptions (figure 10-13)

Figure 10-19. The memo for the cyclic 'e' after **one** iteration of recomputation. The memo `(Val=1)` depends on the old memoized **R**-expr of `0`. However, that **R**-expr was replaced, and its associated memo ("'e' Memo") has been invalidated. Hence, the system is required to *again* recompute the memo for 'e'.

Unfortunately, the new memo in figure 10-19 depended on the old memo of `0`, which has since been invalidated. Hence, the system must again recompute the memo, as in figure 10-20.

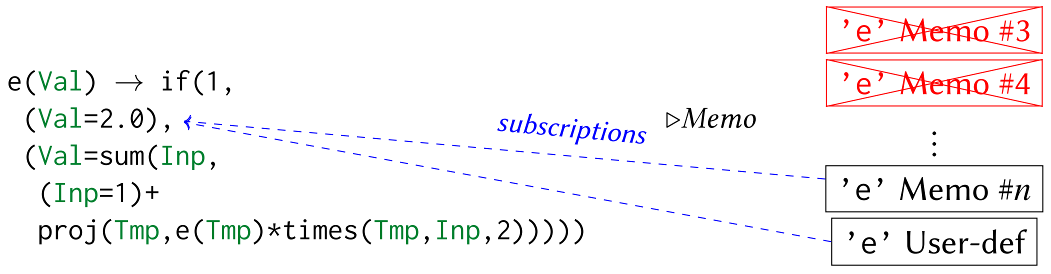


```
e(Val) → if(1,
 (Val=1.5),
 (Val=sum(Inp,
  (Inp=1)+
  proj(Tmp,e(Tmp)*times(Tmp,Inp,2)))))
```

(a) R-expr

(b) Assumptions (figure 10-13)

Figure 10-20. The memo for the cyclic 'e' after **two** iterations of recomputation. The memo is now (Val=1.5), as the previous memoized value was 1, and by the definition of line 454 we have defined this as $1+e/2$. Just like before, we have that the memo of (Val=1.5) depends on the assumption of the previous read ("'e' Memo #2") which has been invalidated as the memoized **R**-expr was replaced. This process of recomputing the memo will continue until the memo is consistent with itself as in figure 10-21.



```
e(Val) → if(1,
 (Val=2.0),
 (Val=sum(Inp,
  (Inp=1)+
  proj(Tmp,e(Tmp)*times(Tmp,Inp,2)))))
```

(a) R-expr

(b) Assumptions (figure 10-13)

Figure 10-21. The memo has converged[149] with (Val=2.0). When the system rechecks this memo, it constructs the same **R**-expr, which can be checked for semantic equality. Hence, the memo is *not* changed, and the assumption "'e' Memo #*n*" is subscribed to by the same **R**-expr it protects. The assumption remains valid, so no further propagation is required.

[149]This program will converge in the sequence $1, 1.5, 1.75, 1.875, \ldots 2.0$. In theory, this is an infinitely long series converging to 2.0, so one might think that this never stops/converges. In our case, however, we are using standard IEEE floating point numbers, which have limited numerical precision. Therefore, this sequence converges when run with floating point when it runs out of numerical precision and is rounded to 2.0.

10.5.2 Choice of default Guesses

As stated earlier, we are allowed to use any **R**-expr as our guess. The choice of initial guess can have a significant impact on the value to which the program converges and the speed at which the program converges. For example, we could have chosen to "*guess*" the **R**-expr `(Val=2.0)`[150] for the 'e' definition in section §10.5.1.1. In which case the system would have been able to immediately validate that the guess `(Val=2.0)` is correct and avoid the iterative process of converging to the value of `(Val=2.0)`.

That said, we have made the decision to make the initial guess always the empty **R**-expr `0`. This choice was made in hopes of minimizing the amount of surprise that results from guessed values. As such, a user-defined term like 'g' in figure 10-24 would have "no value".

10.5.2.1 User Override for Initial Guesses

The choice of the initial guess being empty can be "overridden" by the user of Dyna if they are willing to introduce a small modification into their program. Using the `:=` aggregator, we can set an initial value for when nothing exists. An example of this is shown in figure 10-22.

[150]This could have been computed using the closed form formula for a geometric series: $\frac{1}{1-r}$ where $r = \frac{1}{2}$ in this case.

```
455 g_with_guess = 2*g - g**3.  % g can either be 1 or -1
456 g := .5.                    % User defined guess
457 g := g_with_guess.
```

Figure 10-22. Users can override the default guess using the `:=` aggregator. Here on line 456, the default guess is set as `.5`. When `g_with_guess` is an empty **R**-expr—which is the hard-coded default for a guess—the rule on line 457 will not contribute any value. Therefore, '`g`' will take the value from line 456. Once `g_with_guess` has *some* non-null value, then '`g`' will take the value from line 457. This initial guess causes '`g`' to converge to value 1 instead of -1.

10.5.3 Guesses are Un-bypass-able

Guesses have additional requirements on them compared to "*normal*" memos from section §10.1.2. Most importantly, guesses are not *bypassable*. The reason is a guess can change the observed semantics of a Dyna program. The semantics of Dyna only require that the system find an assignment to all terms in the program that is *consistent* (section §2.5). In practice, there are many different consistent assignments to the terms in the program. For example, consider the program in figure 10-23, which has a cycle involving '`a`' and '`b`'.

```
458 a := 1.
459 a := b.
460 b := 2.
461 b := a.
462 print b.  % print either 1 or 2 depending on guess
```

Figure 10-23. Dyna program which prints 1 or 2 non-deterministically depending on where the back-chaining cycle is broken and how a guess is made. If the value of ‘a’ is guessed as null, then the value of ‘b’ will be set to 2 on line 460. This causes ‘a’ to take on the value 2 as well, by the definition of the := aggregator, the last line of the program which is non-null for a rule defines its value, and in this case it will be line 459 is now non-null as ‘b’ is defined as 2. Conversely, if ‘b’ is guessed as null, then ‘a’ will be set to 1 due to line 458, therefore ‘b’ will also take on the value of 1 by the definition of the := aggregator and line 461.

The value which is returned by ‘a’ and ‘b’ can either be 1 or 2 depending on the order in which guesses are made. The Dyna specification says that either value is acceptable, as Dyna only requires that the assignment to all expressions is consistent.

A more extreme version of figure 10-23 when considering Dyna’s requirement to make an assignment consistent is shown in figure 10-24.

```
463 g = g.
```

(a) Dyna

```
g(Val) → (Val=only(Inp, g(Inp)))
```

(b) **R**-expr

Figure 10-24. The Dyna rule ‘g’ can take on any value and still be consistent with its definition on line 463. Hence, whatever value is guessed for ‘g’ will persist.

Given the definition of rule ‘g’, it is “*allowed*” to take on any value. For example, if ‘g = 77’, then we would have 77 = 77 which is *consistent*. Similarly, ‘g’ can take on the value `g = "hello"`, as `"hello" = "hello"`, and is also *consistent* with `g`'s definition.

10.5.3.1 Why Guessing Requires `if`-expressions

In section §10.2, I claimed that memoization involving cycles does not work if we are memoizing Simplify directly. The reason is that we need to ensure that we intercept user-defined **R**-exprs with guesses. If we instead memoized Simplify, then we might encounter something that needs to be overridden by a guess but cannot be easily recognized due to the difficulty of comparing **R**-exprs for semantic equivalence. Essentially, this means that we would either need to have a check for semantic equivalence between **R**-exprs (which is impossible in general because checking semantic equivalence is Turing-complete), or accept that guesses sometimes end up getting bypassed. Neither of these is acceptable, as the semantics of Dyna *requires* consistent set of assignments to the values associated with terms (section §2.5).

10.6 Cleaning up the R-expr Presentation of Memoization—Memos Held Outside of the R-expr

So far, I have waved my hands when it comes to how memos are managed in an **R**-expr. I have demonstrated that an `if`-expression can be used to *override* the original **R**-expr (as needed) for memoization, and that the system can store the memoized **R**-expr inside the *true*-branch of the `if`-expression. However, we still

```
class MemoizationContainer implements MessageListener {
  final Rexpr original_rexpr;  // The original R-expr  (false if branch)
  Rexpr signature_have_memoized;           // what is currently memoized
  final Rexpr what_want_memoized;                // $memo representation
  Assumption assumption;         // replaced when memo changes (§10.4.1)
  Rexpr the_memo;                      // The memo itself (true if branch)
}
```

Figure 10-25. The memoization container

lack details on how the **R**-expr is modified and how the relevant assumptions are tracked with each **R**-expr.

In this section, I will lessen the abstraction and talk more specifically about how these different pieces of memoization fit together.

First, we will move the memo out of the **R**-expr. To do this, we are going to replace the `if`-expression **R**-expr and use a `memoRead(···)` **R**-expr to represent the memo. The `memoRead` **R**-expr maintains a pointer/reference to the relevant memoization container and has a list of one or more local variable names: `memoRead(` *memo_container_pointer*, $\mathtt{Y}_1$, $\mathtt{Y}_2, \ldots, \mathtt{Y}_n$`)`. The memoization container (figure 10-25) is a mutable data structure that contains the memo. The memoization container can be referenced by multiple `memoRead` **R**-exprs at the same time. This allows all `memoRead` **R**-exprs who access the same memo to have a consistent view of the currently memoized **R**-expr. The fields on the memoization container are as follows:

1. The original **R**-expr.—This is used when something is configured to bypass the memo or is not current computed

2. The signature represented as an **R**-expr.—This corresponds with the conditional of the `if`-expression and records what is currently stored in the memo table.

3. An **R**-expr that tracks what we *want to be memoized.*—This is the `$memo(·)` memoization control mechanism, which I will discuss in section §10.8.

4. An assumption that is associated with the current memo. Anytime the memo is changed, the assumption is invalidated, and messages are sent to all downstream dependents, as previously discussed in section §10.4.1.

5. The memo itself, represented as an **R**-expr.

The variables $\mathtt{Y}_1$, $\mathtt{Y}_2, \ldots, \mathtt{Y}_n$ on the `memoRead(`*memo_container_pointer*`,`$\mathtt{Y}_1$, $\mathtt{Y}_2$, $\ldots, \mathtt{Y}_n$`)` refer to local variables, which allows variable renaming in the **R**-expr which uses `memoRead` as a sub-**R**-expr. Inside the memoization container, variable names are normalized to known variable names like $\mathtt{X}_1$, $\mathtt{X}_2, \ldots, \mathtt{X}_n$. When `memoRead` is rewritten as an **R**-expr returned from the memoization container, the returned **R**-expr's variables are renamed to $\mathtt{Y}_1$, $\mathtt{Y}_2, \ldots, \mathtt{Y}_n$ to match the local context.

10.6.1 Memoization Container

Comparing the memoization container with the `if`-expression, we can see that we maintain all the same capabilities as with the `if`-expression. The signature in the conditional test of the `if`-expression is held in the `signature_have_memoized` variable, the true-branch corresponds to the `the_memo` **R**-expr, and the false-branch corresponds to the `original_rexpr`. The new addition in the memoization container is the `what_want_memoized` **R**-expr variable.

The `what_want_memoized` **R**-expr is the memoization policy, which is defined using `$memo`. The reason we separate the *policy* from the *signature* of what is memoized is that sometimes the policy is not helpful when it comes to the memoized **R**-expr.

For example, consider the case where the signature is directly derived from the memoization policy. Suppose that our memoization policy says that the system should memoize "*everything*". In this case, the policy can be represented as the **R**-expr `1`. However, this is not a useful **R**-expr when it comes to our rewrite rules. Recall that we can introduce an `if`-expression using any **R**-expr (which represents our memoization policy, rewrite rules 66 and 67). As such, if we use a memoization policy of `1`, we end up with `R→if(1,R,R)→if(1,1*R,R)` where `1*R` does not yield any useful rewrites.

Alternately, consider the case where the memoization signature is different from the memoization policy. Again, suppose that the memoization policy says memoize everything, which is represented as the **R**-expr `1`. However, we are now going to say that the signature represents queries that we have encountered before and have managed to do useful rewriting on the **R**-expr. In other words, we can now have a memoization signature like `(X=2)*(Y=3)+(X=7)*(Y=11)`. This means that the memo corresponds with an **R**-expr like `R→if((X=2)*(Y=3)+(X=7)*(Y=11),` (X=2)*(Y=3)*R+(X=7)*(Y=11)*R,R). The **R**-expr (X=2)*(Y=3)*R and (X=7)*(Y=11)*R are likely to have useful rewrites as we can condition those rewrites on the values of X and Y.

10.6.2 Handling Memos we "*Want*" but do not "*Have*"

It is often the case that what a user wants memoized, as defined *policy*, and what the system *has*, memoized as tracked by the signature, are often not aligned. As such, when an inconsistency is detected, the memo's signature and memoized **R**-expr are updated. This is done by waiting for queries to be made against the memoRead **R**-expr, which are then checked against the policy. When the policy indicates that a *query* should be memoized, that query will have a memo created, just like in sections § 10.2 and 10.5. Conceptually, the memoization policy held in what_want_memoized can be handled by an if-expression as shown in figure 10-26.

```
memoRead(container, ···) ≈ if(container.what_want_memoized(···),
                                readOrMakeMemo(container, ···),
                                container.original_rexpr(···))
```

Figure 10-26. Approximate interpretation for the memoRead(···) **R**-expr. When the memoization policy what_want_memoized is rewritten as 0 or 1+R for some R, this allows the system to determine if something should be memoized—in the same way that an if-expression conditions on 0 or 1+R. In the case that the policy states that something should be memoized, the "*readOrMakeMemo*" will check the current memoization signature to see if the requested query is contained in the memo table. In the case that the requested query is not contained, the memo and signature will be updated as discussed in sections § 10.2 and 10.5.

Just like with an if-expression, when the memoization policy cannot be rewritten as `0` or `1+R` for some `R`, the system will have to defer accessing the memo. Hence, a bad memoization policy can cause the system to become inoperable.

10.7 Update Loop

It is often the case that there are many different updates that are pending at any given point in time. Dyna permits us to process updates to memos in any order we choose. Dyna only requires that all memos are consistent with each other when all updates have finished processing. Therefore, to handle this, we buffer all updates using a (priority) queue.

The update queue is a global object. The system runs a loop processing updates until the queue is completely empty. When processing an update, the system will check if the current memo is consistent by recomputing it from the original **R**-expr using SimplifyNormalize (algorithm 1) and check if the resulting **R**-expr is semantically equivalent to the currently memoized **R**-expr.[151] In the case that the memo must be updated, the memo is updated first, and then the assumption associated with the memo is invalidated, causing downstream dependencies to receive notifications.

When a notification is received, a pending update is added to the queue,[152] and

[151]In general, checking the semantic equivalence of **R**-exprs is difficult unless we take special care to ensure that the structure of the memo is sufficiently "*simple*".

[152]The queue has previously called an agenda in prior publications on Dyna. This was done previously to align the terminology with parsing algorithms [59].

the update loop will continue its processing.

This is shown in algorithm 4.

```
1:  function ProcessUpdates( )
2:      global UpdateQueue
3:      while not Empty(UpdateQueue) :
4:          work ← Pop(UpdateQueue)                  ▷ Pop from the queue according
5:                                                      to the priority function
6:          ▷ "work.memo" is a pointer to mutable container which holds the memo itself. It con-
            tains the fields that track the original R-expr, the current memoized R-expr, and the
            assumption which protects read operations performed against the memo.
7:          R_original ← work.memo.original_rexpr
8:          global assumptionsTracker ← ∅     ▷ A set of all assumptions depended on during
9:                                               SimplifyNormalize, automatically tracked
10:         R_new ← SimplifyNormalize(R_original)
11:         if SemanticallyEquivalent(R_new, work.memo.the_memo) :
12:             No Op                          ▷ No update happens if it is semantically equivalent
13:         else
14:             oldAssumption ← work.memo.assumption ▷ Save a pointer to existing assumption
15:             work.memo.the_memo ← R_new                         ▷ Update the memo
16:             work.memo.assumption ← new Assumption( )
17:             oldAssumption.Invalidate( )               ▷ Invalidate the old assumption
18:         for assumption ∈ assumptionsTracker :
19:             assumption.subscribe(work.memo)
```

Algorithm 4. A rough outline of the Process Updates Function. The function uses a global update queue (which can be a priority queue) to choose the next unit of work. It recomputes the **R**-expr R from scratch based on the original definition. If the newly computed **R**-expr is semantically equivalent, then the memo is not modified.

10.8 Controlling Memoization

When it comes to memoization, there are many different memoization policies, which can have a significant impact on the runtime and memory efficiency of the overall system. For example, the order in which updates are processed can be the difference between a program running in linear time or exponential time (section §2.7.1). Furthermore, there exist programs where a bad memoization policy can cause the program to become not terminating.[153]

Ideally, the Dyna system would *automatically* figure out how to best apply memoization and prioritize updates. Unfortunately, at this time, automatic configuration of memoization is still an open research question. Instead, what we have currently is a mechanism to allow users to define their own memoization policies and update orders using `$memo` and `$priority`, which are described here again but were previously introduced in section §2.7.

10.8.1 `$memo(·) = "none"|"null"|"unk".` [154]

The most significant control mechanism for memoization is `$memo`, which essentially controls if memoization is used or not, and should therefore *fallback* to the original un-memoized **R**-expr. As previously discussed in section §2.7, `$memo(·)` takes one argument that matches the structured-term with the same name as the user-defined term and "returns" the string `"none"`, `"null"`, or `"unk"` to indicate "how" and what should be memoized.[154]

Internally, in the implementation of memoization, we are instead going to think

[153] For example, in the Fibonacci program figure 10-8, a memoization policy of `$memo(fib[X:$free]) = "null".` will cause the system to eagerly memoize *all* Fibonacci numbers, for which there are an infinite number of. Hence, this will not terminate.

[154] We could have used any ground value as the returned value from `$memo`. The names `"none"`, `"null"`, and `"unk"` were chosen to maintain continuity with prior work on memoization on the Dyna project [66, 67].

about `$memo` slightly differently from the surface-level syntax. Instead, we will say that the memoization policy defined by `$memo` and represented as an **R**-expr is in one of three "states". These states are as follows:

1. The query is memoized (or it should be memoized),

2. The query should not be memoized, and it should fallback to the original **R**-expr, and

3. There is not enough information yet to determine whether the memoization policy is in state 1 or state 2.

The first state where a memo should exist corresponds with `"null"` being returned by the `$memo` function. The second state with no memo corresponds with `"none"` being returned and is also the default state if nothing else is defined for `$memo`. The third state of being "unsure" corresponds with having `$memo` represented as an **R**-expr that cannot be *fully rewritten* yet. Notably, the keyword `"unk"` does not correspond to any single state (section §10.8.3). Instead, `"unk"` represents a common case that switches between the third state of "unsure" and the first state where a memo exists once enough variables are grounded and will be discussed further in section §10.8.3.

10.8.2 $ground and $free Annotation

When it comes to memoization policy, the system needs to determine when it *must* check the memo, and it is allowed to bypass the memo and fallback to the original **R**-expr. Sometimes, though, we do not care about the value that is being stored but rather that there is enough information available in the query such that any memo that we create will be useful. To support different memoization policies, we have `$ground(·)` and `$free(·)` which work as annotations on the state of the variable.

To see how `$ground` and `$free` work, let us consider the following memoization policy for the rule `foo(X,Y,Z)`:

```
471 $memo(foo[X,Y,Z]) = "null".
```

Given the policy on line 471, there is no requirement on the value for the arguments to `foo`, hence the memo for the user-defined `foo(·,·,·)` is allowed to be equivalent to the original **R**-expr.[155]

Conversely, suppose that we have the memoization policy:[156]

```
472 $memo(foo[X:$ground,Y:$ground,Z]) = "null".
```

Here, we require that `X` and `Y` are known ground values before a memo is created. This means that our memo signatures will look something like '`(X=1)*(Y=7)+ (X=5)*(Y=3)+ ···`', and that the memoized **R**-expr can depend on the values of `X` and

[155] In the case of a cycles, having a `"null"` memo can still be useful as the `"null"` memo will break the cycle, as seen in section §10.5.

[156] This is making use of the colon notation, which was presented as a Type Declaration section §2.8.3.

Y. Hopefully, this will result in useful rewrites being completed before the resulting **R**-expr is stored in the memo.

The way that `$ground` is implemented is that it requires that its argument take *some* value before it returns true. If the value is not known, then `$ground` becomes a delayed constraint and that prevents the **R**-expr from being rewritten. This can be implemented by defining a single rewrite rule that checks that a variable is ground, as in figure 10-27.

12. $[\![\texttt{ground(X)}]\!]_E = 1$ $\qquad\qquad$ $\texttt{ground(X)} \xrightarrow{85} 1 \quad \textbf{if } \texttt{X} \in \mathcal{G}$

Figure 10-27. Semantic definition and rewrite rule for $ground

Because `$ground` prevents the memoization policy **R**-expr from being rewritten, it delays accessing the memo table as the memoization policy must be rewritten as a multiplicity *before* the system is allowed to check the memo or bypass the memo (section §10.6.2).

To complement `$ground`, `$free` is designed to annotate variables that may not yet know their value (which is typically called free in logic programming). Unlike `$ground`, `$free` is a no-op and only serves to indicate that `$ground` was not forgotten.

The reason that `$free` is a no-op is that we want to avoid inconsistent behavior when it comes to reading a memo. For example, suppose that we had a memo for the query `foo(1,X)` where X is a free variable. This memo can answer the query `foo(1,3)`, as it is possible to unify these two expressions together with `X=3`. However, if we *explicitly* matched against the variable `X` being free, then this would be an inconsistent memoization policy and could potentially bypass guesses, which is not allowed (section §10.5.3).

10.8.3 "unk" Memos

The keywords `"null"` and `"unk"` were chosen to maintain continuity with prior publications on memoization and guessing in the Dyna programming language [66, 67]. That said, only the externally observable behavior of `"null"` and `"unk"` is consistent with the prior work. The conceptualization of `"null"` is quite similar to the prior work, but `"unk"` is very different.

In the prior work [66, 67], `"null"` and `"unk"` were used to represent the "*default*" value of a memo. With `"unk"` standing for "*unknown value*", and `"null"` representing that the memo has a "*null valued*" guess that must be validated. When an `"unk"` value was encountered during back-chaining, the system would immediately perform the computation using the original program and save the result. Conversely, a null guess would be eagerly computed ahead of time—as in the style of forward-chaining Datalog. Hence, the distinction between `"null"` and `"unk"` was deciding *when* the computation is performed.

In the prior work, without **R**-exprs, it was in fact very important to make this distinction about when computation happens. Delightfully, with **R**-exprs we can be a bit more *fast and loose* about when computation happens in terms of our memoization formalism. The reason is that **R**-exprs can represent computation rather than a value in the language (e.g. a memo that can contain `plus(1,2,X)` vs. the value 3). Therefore, we can memoize **R**-exprs that represent partially evaluated computation, which was not allowed under the prior formalism.

As such, we do not have to worry about ensuring that our memoized **R**-exprs are fully rewritten into a single value assignment. This leaves one question: what did an "unk" memo represent (according to the prior work), and how are we going to emulate "unk" with **R**-exprs. In the prior work, the memo table consisted of fully grounded terms as keys (e.g. `foo(1,2,"hello")`), and associated a value with each key (e.g. 3, 7, `"hello"`, `bar[1,2]`, etc.). Therefore, when using a memoization policy of "unk", the prior work did *not* allow queries of the form `foo(1,Y,Z)` against the memo table, as the previous approach *cannot* make any guarantees about terms not found in the memo table, as their value is unknown. Hence, the prior work *only* allowed queries for *fully grounded terms* like `foo(1,2,"hello")` when using an "unk" memoization policy. As such, the "unk" memoization policy can be seen as equivalent to requiring that all variables grounded: `$memo(foo[X,Y,Z]) = "unk".` ≡ `$memo(foo[X:$ground,Y:$ground,Z:$ground]) = "null".`

10.8.4 Implementation of `$memo`

With `"unk"` now represented in terms of `"null"`, we now only need to handle the three cases when implementing `$memo`:

1. When "null" is returned, we need to read from the memo table,

2. When "none" is returned, we need to skip the memo table and

3. When we do not yet know if it is "null" or "none" and need to defer the read operation.

Conceptually, these three cases can be handled with an if-expression like **R**-expr. If the conditional branch of the if-expression can be completely rewritten as either 1+R for some R or as 0, then we do not know which branch of the if-expression to return. This is identical to what I presented in figure 10-26, in how we use the memoization policy. As such, we can define the variable what_want_memoized as the **R**-expr:

```
proj(Name,
  (Name=foo[X1, X2, X3])*
  $memo(Name, "null"))
```

Figure 10-28. The "what want memoized" **R**-expr (line 467 of figure 10-25), which adapts from the $memo memoization policy represented as an **R**-expr that returns "null" or "none" or not rewrite and becomes a delayed constraint, which happens if there is enough information in the context $\mathcal{C}$ rewrite the user-defined $memo **R**-expr.

The Name variable holds the structured term for the user-defined term for which a memo represents. The $memo(Name,"null") is a call to the $memo **R**-expr definition. When $memo returns "null", this **R**-expr will be rewritten as 1, causing the if-expression in figure 10-26 to access the memo table. When $memo returns "none", then this will be rewritten as `0`, causing the `if`-expression in figure 10-26 to fall through to the original **R**-expr. In the case that `$memo` cannot be fully rewritten (possibly caused by a `$ground` annotation), the `if`-expression in figure 10-26 will defer reading or bypassing the memo.

10.8.5 Ordering Updates with `$priority`

The order in which updates are processed can have a significant impact on the runtime of the program. `$priority(·)` supports different ordering of updates by defining a floating point number which is used to define the priority of an update. Whenever an assumption is invalidated and an update is pushed to the update queue, the `$priority` function is consulted, using the user-defined rule name and any values that are known to compute a floating point value. This value is used to sort the priority queue, which contains all pending updates. The priority is only computed when updates are pushed to the queue to avoid the complexity of reordering the queue due to values that `$priority` depended on changing.

Chapter 11

A Realistic Implementation of R-exprs

So far we have been discussing an abstract idea of **R**-exprs (chapters 5 and 6), rewriting **R**-exprs using SIMPLIFY (chapter §8) and extensions to this design in the form of rearranging **R**-exprs (chapter §9) and adding memoization (chapter §10). In this chapter, I am going to transition from abstract ideas and focus on a *real implementation* of **R**-exprs and **R**-expr rewriting. Note that some design choices discussed in this chapter were made with the intention of JIT compiling **R**-exprs, which will be discussed in chapter §12.

11.1 Design Goals for our Implementation

Before diving into the details about the implementation itself, let us discuss what the *goals* are for this implementation of Dyna and **R**-expr rewriting.

1. It should be usable and capable of supporting the kinds of Dyna programs that we expect users to write.

2. It should be "efficient" so that it can be used on "real" problems. — Recall that our target audience for Dyna is Machine Learning (ML) and Artificial Intelligence (AI) researchers. This means that the kinds of problems we expect often have the form of performing *very similar* computations on slightly different data many times in a loop. An example of this kind of computation would be matrix multiplication.

3. It should make the implementation of **R**-exprs and **R**-expr rewrites *easy*. — A complete Dyna implementation requires dozens of different **R**-expr kinds and hundreds of rewrite rules. Additionally, there is a lot of potential for future work to introduce new **R**-expr kinds and rewrite rules.

4. It should support *all* of Dyna. — There have been several implementations of Dyna over the years.[157] However, previous implementations have only implemented a small subset of Dyna. Building a complete implementation of Dyna demonstrates that the term rewriting approach in this dissertation is the "right formalism" and sufficient to implement a complicated logic programming

[157]There are seven implementations of Dyna, including the one discussed in this chapter (depending on how you count). The major implementation attempts are linked at `http://dyna.org/#downloads`. The source discussed in this chapter is at `https://github.com/argolab/dyna3`. Previous Python prototypes that use rewriting can be found at `https://github.com/argolab/dyna-R` and `https://github.com/argolab/dyna-R/blob/backend-v2/dyna_match_paper/rexprs.py`.

language like Dyna.

5. It should support (JIT) compilation of Dyna programs. — This is a secondary goal of wanting the implementation to be “efficient” and “fast”, but this is a significant design constraint on the implementation, so I mention this at the start as a design goal.

11.1.1 What does “Efficient Implementation” Mean?

We would like Dyna to execute programs at a level that is comparable to the “first implementation (of an algorithm) that a user would write in a procedural language.” In other words, we understand that a skilled programmer can make an algorithm *faster* through the use of clever tricks and knowledge about computer architecture. Our goal is not to compete with the skilled programmer. Instead, our goal is that an *average skilled programmer* will be able to develop programs in less time with Dyna when compared to a procedural language (due to Dyna’s simplicity) and have the program run at a level that is comparable to what an *average* skilled programmer would write in a procedural programming language.[158]

To illustrate this point, let us consider a program which computes a matrix-vector product:

```
a(X) += b(X,Y) * c(Y).
```

[158]At this point, we are not considering the constant factor overheads of implementation (such when choosing to write a program in slower Python vs faster C).

If we ran line 473 using the execution strategies that we have discussed so far, we would represent `b(X,Y)` and `c(Y)` as the disjunctions in figure 11-1.

```
b(X,Y, Val) →
  (X=1)*(Y=1)*(Val=3)+
  (X=2)*(Y=1)*(Val=7)+
  (X=1)*(Y=2)*(Val=9)+
  ⋮
```

(a) Example **R**-expr for `b(X,Y)`

```
c(Y, Val) →
  (Y=1)*(Val=11)+
  (Y=9)*(Val=23)+
  (Y=5)*(Val=0)+
  ⋮
```

(b) Example **R**-expr for `c(Y)`

Figure 11-1. Example **R**-exprs used by line 473.

Accessing the element from the disjunctions in figure 11-1 requires many steps of rewriting. We would have to use the distributive rewrites to expand out the **R**-expr out (e.g. `(X=1)*(Y=1)*(ValB=3)*((Y=1)*(ValC=11)+(Y=9)*(ValC=23)+(Y=5)*(ValC=0)+⋯)+⋯`) which would create a large **R**-expr and be inefficient, requiring hundreds of rewrites to access the relevant elements from the `b(X,Y)` and `c(Y)` relations.

Conversely, let us consider how line 473 would be written in a procedural language:[159]

[159] Assuming that we are writing out the loops for the matrix-vector product and not using a library function.

```
1: function MatrixVectorProduct(b[0..$N$, 0..$M$], c[0..$M$])
2:     ret ← MakeZeroVector($N$)
3:     for $X$ = 0 to $N$ :
4:         for $Y$ = 0 to $M$ :
5:             ret[X] ← ret[$X$] + b[$X, Y$] * c[$Y$]
6:     return ret
```

Algorithm 5. Example procedural implementation of matrix-vector product

Here, MatrixVectorProduct is able to assume that the representation of 'b' and 'c' are *dense arrays/matrices* of numbers. This makes accessing elements in the array (line 5) efficient, taking $O(1)$ time. Furthermore, the MatrixVectorProduct function is able to *know* the size of the matrix and vector using the integers N and M and have an iterator that loops over the domain of the matrix and vector, being the integers between $[0,N)$ and $[0,M)$ (lines 3, 4). Finally, the place for the final result is allocated before the computation is started (line 2). This means that the procedural code does not have to store all the intermediate multiplications and instead aggregates the value along the way into the final location (in memory).

Our goal, with **R**-expr-based rewriting, is to maintain the *representational power* of **R**-exprs, while having the execution of **R**-exprs be comparable to algorithm 5, in that we can avoid the overhead of naively rewriting **R**-exprs.

11.2 Implementation Overview

Given the requirements that I listed above for a realistic implementation, I evaluated a number of different possibilities for the implementation of Dyna. I make no claim that this design is the “*best*” approach. However, the design presented in this chapter is the third implementation of Dyna that I have completed during my Ph.D., and it has been designed with the insight gathered from the prior two implementations. I believe that the design presented here is worthy of consideration for anyone looking to build a term rewriting system for a programming language.

First, for the choice of programming language, I choose to use Clojure [92]. Clojure is a LISP-like programming language that targets the Java Virtual Machine (JVM). Clojure is a reasonably fast language, running a little slower than Java but faster than many other high-level languages. Clojure can also interface with Java libraries, which is convenient for interfacing Dyna with Java (and Python through the use of Java-Python interfaces). The *main* reason for choosing Clojure was that there are a few features that are essential for implementing Dyna. These are macros, the ability to modify ASTs of the host language, and a runtime evaluate function. Macros and AST manipulation are essential for making the implementation of the many different rewrite rules and **R**-expr kinds efficient, and the runtime evaluation function is used to generate **R**-expr-dependent procedural code. Admittedly, any LISP-like language satisfies these requirements.

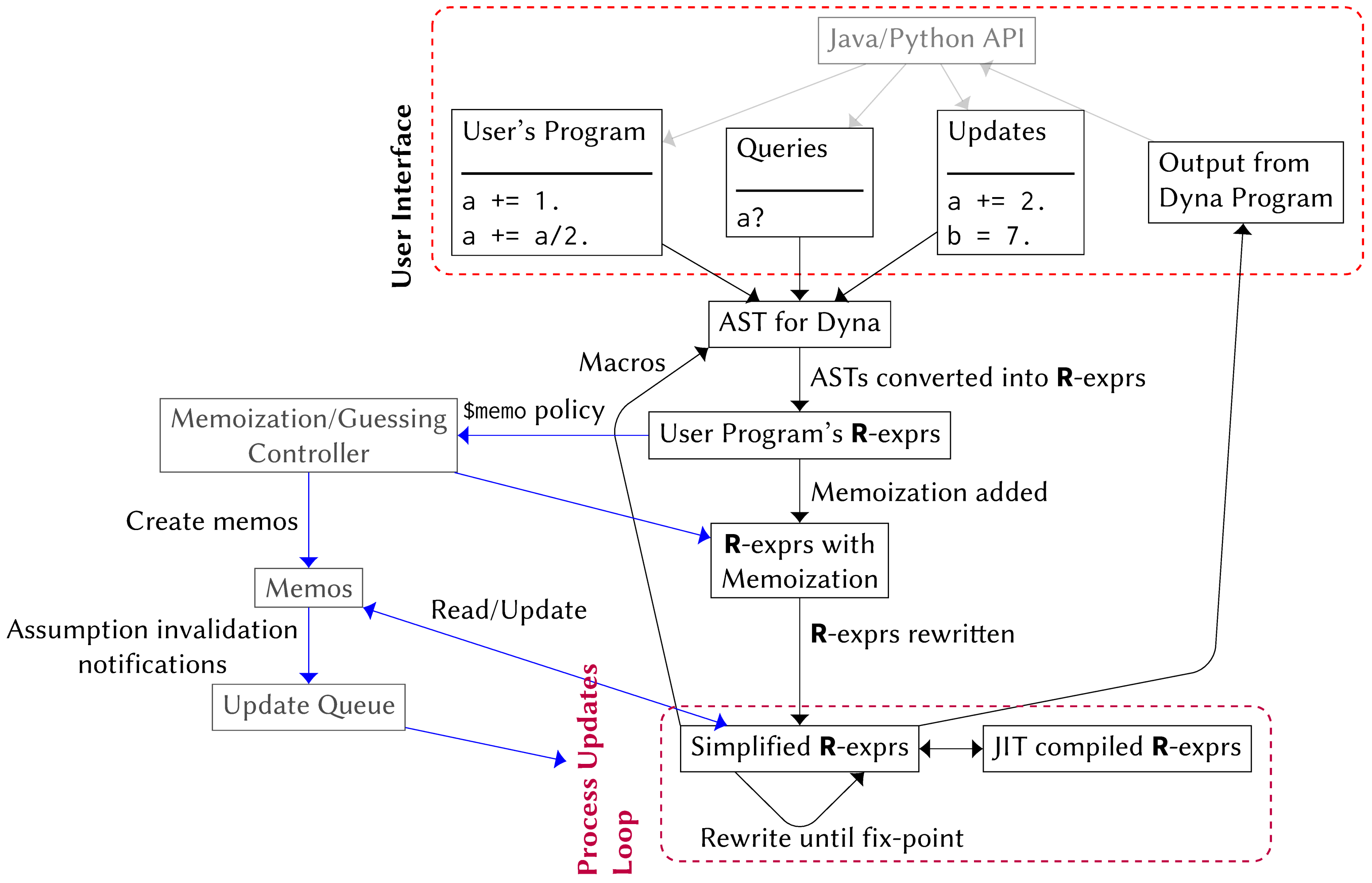


Figure 11-2. A high-level figure representing the major components in the Dyna implementation. The user's program, queries, and updates start at the top and then are converted into the Dyna AST. The AST uses the same structured term class as section §2.1.1, allowing Dyna programs to manipulate the ASTs using macros (section §2.10.2). ASTs are converted into **R**-exprs as in chapter §7. Memoization (chapter §10) is added to the program by rewriting the user's **R**-exprs with the `memoRead` **R**-exprs and constructing the relevant memo tables. The memoization policy (`$memo`(·)) is represented as an **R**-expr, which is processed specially by the memoization controller. Queries and updates made against the Dyna program are converted into **R**-exprs and rewritten using SimplifyNormalize (algorithm 7) until there are no more rewrites to apply, and there are no further updates to memos. The resulting **R**-expr is then returned to the user. JIT compilation (chapter §12) attempts to make execution with **R**-exprs faster.

Dyna's internal state is represented using **R**-exprs. All the major components inside Dyna pass **R**-exprs between them. The flow of data/**R**-exprs is shown in figure 11-2.

11.3 Dyna's Front-end

Dyna provides a Read-Print-Eval loop (REPL) as a front-end, as well as Java and Python API (section §2.3.1). The API, and REPL accept strings of Dyna code that represent queries and updates (as previously shown in section §2.3.1). The Dyna source is converted into **R**-exprs and then rewritten completely.

The conversion process from a string of Dyna source code into an **R**-expr is done with the help of two special **R**-exprs. The first is `string_to_ast(S,Ast)`, which converts a string of Dyna code into the Dyna AST. The implementation of the `string_to_ast(S,Ast)` rewrite rules uses Antlr4 [110][160] to parse the string of Dyna code and assigns the AST to the `AST` variable. The Dyna AST is represented using the same term class used to represent structured terms (section §2.1.1). This is similar to Prolog where rules are represented using terms. This allows macros defined in Dyna to modify the Dyna AST before it is converted into **R**-exprs (section §2.10.2).

The second **R**-expr is `ast_to_rexpr(Ast,ResultVariable,` *VariableNameMap*`)`. `ast_to_rexpr` takes three arguments. The first variable is the AST itself. This can be a top-level AST, which comes from parsing an entire file or a line of code from the

[160]Antlr4 is an off the shelf parser/lexer generator which generates a parser in Java.

REPL, or this can be a smaller unit of the AST, which corresponds to an expression inside of the Dyna source code (as defined in chapter §2). The second variable `ResultVariable` is the result from evaluating an expression in Dyna. In Dyna, all expressions return some value. For example, the AST corresponding to '`1 + 2`' is well formed and will return the value `3` once evaluated. The reason we need the return variable as an argument is that **R**-exprs do not have a *return value* but instead return the multiplicity, which indicates that the assignment is "*true*". Therefore, we must assign the return value of the Dyna expression to a given variable when converted into **R**-exprs.[161] Asts that are not expressions, such as top-level asts and declaring rules, are made into expressions by defining them to return the dummy value of *true*. The third argument is a map from string variable names, as used in the source code, to the value object types: $\mathit{VariableNameMap} : \mathit{String} \to \mathcal{V}$. The **R**-expr is not required to use the same identifier as the source code; hence, the map tracks the association between names and variables. For example, $\{\text{"}X\text{"} \mapsto \texttt{VarX72}, \text{"}Y\text{"} \mapsto 7\}$.

The `string_to_ast` **R**-expr can be considered a "proper" **R**-expr, in that it can be completely defined as a relation on its arguments. On the other hand, `ast_to_rexpr` is not proper. When `ast_to_rexpr` is rewritten, it modifies the global[162] state of the Dyna system. This includes changing the definition of user-defined **R**-exprs

[161]This is similar to representing the return value as an argument in a Prolog program. For example, `append([1,2], [3], [1,2,3])`.

[162]Throughout this chapter, I mention different global variables which are used to track the state of the system. More precisely, the global state is actually thread-local and specific to the current invocation of Simplify running. In this dissertation, the system is currently single-threaded, so there is no meaningful distinction.

(section §5.2.2.11).

Once the Dyna system, which is controlled by the API (section §2.3.1), is finished evaluating all queries and updates provided as a string, the results from queries are returned to the user. If the resulting **R**-expr from a query is “*sufficiently simple*”, then Dyna automatically extracts the value from the **R**-expr representation so it is more easily interpretable by the driver program. This includes ground values such as integers and strings, as well as arrays and hash maps. If the result from a query is not a simple **R**-expr, then the **R**-expr itself will be returned to the driver program. Users are expected to write programs that result in simple **R**-exprs and values, therefore an **R**-expr being return generally indicates that “something has gone wrong[163]” when evaluating a query. Hence, users are not expected to build driver programs that understand **R**-exprs. The Dyna REPL, which is built upon the Dyna API, will attempt to pretty-print **R**-exprs.

A high-level overview of the front-end is shown in algorithm 6.

11.4 Realistic Rewriting, Part 1—Redesigning SIMPLIFY(NORMALIZE)

In this chapter, we are trying to make our implementation of rewriting *faster*. As such, we are going to make some changes to the approach described in chapter §8

[163]In the same way that a computer algebra system may return a large, complicated expression.

[164]A guess will initially made with the `0` **R**-expr, which may cause the result from the query to be `0` also. Once the guess converges, the query may have a non `0` result. This was previously shown in section §10.5.

```
1:  function DynaFrontEnd(Source)
2:      R ←proj(A,string_to_ast(Source, A)*ast_to_rexpr(A,true, {}))
3:      repeat
4:          ProcessUpdates( )      ▷ Call pending updates on the update queue, algorithm 4
5:          global[162] QueryResult ← ∅
6:          R2 ← SimplifyNormalize(R) ▷ Evaluate the program, modifying global as needed
7:          assert R2 == 1          ▷ The top-level AST rewrites as multiplicity 1 upon success
8:      until isQuery(Source) and isEmpty(UpdateQueue)   ▷ If pending updates, run
    updates and redo queries, algorithm 4
9:      ▷ Query results are saved into the QueryResult buffer by ast_to_rexpr
10:     if QueryResult matches (ResultVariable=Value) :
11:         return Value      ▷ Avoid returning the R-expr if the R-expr is "sufficiently simple"
12:     else
13:         return QueryResult            ▷ Return the entire R-expr produced by the query
```

Algorithm 6. The front-end for Dyna is called on a single update or query at a time. An update/query is wrapped in a standard **R**-expr (line 2) and is passed to SimplifyNormalize (line 6) where all execution happens via rewrites. If a query is performed that causes guessing (which can cause pending updates on the update queue), then those updates will be processed, and the queries will be redone (line 8).[164]Once the query has converged, the result is returned to the user, with sufficiently simple values being cast into a usable format.

in hopes of making execution run faster.

Previously in section §8.2.1, we had the functions SIMPLIFY (section §8.A) and SIMPLIFYNORMALIZE (algorithm 1). The function SIMPLIFY recursively invokes itself on the **R**-expr performing *any* rewrites it can apply. To find relevant rewrites, SIMPLIFY matches the current **R**-expr and uses the context $\mathcal{C}$ to identify conjuncts without rearranging the **R**-expr. SIMPLIFYNORMALIZE invokes SIMPLIFY in a loop until no more rewrites can be applied.

The first problem with this design we will address is that SIMPLIFY chooses among *all* possible rewrites equally. However, not all rewrites are equally *useful* or can be matched *quickly*. To manage this, we separate our rewrites into three separate categories, depending on what information is needed to match the rewrite and what priority the rewrite should run at.

1. The first category we call *standard* rewrites. These rewrites only need a local view of the **R**-expr and knowledge about bindings to variables, which is available in the context $\mathcal{C}$. These rewrites correspond to common operations such as performing addition between values (e.g. rewrite rule 28, `plus(1,2,X)`$\to$`(X=3)`).

   These rewrites are classified as *standard* as they are the most common kind of "*useful*" rewrites that we have. Furthermore, variable bindings can be efficiently tracked using an associative map (hash map) inside the context $\mathcal{C}$. This simplifies the context as it does not require *complete knowledge* about all conjunctive **R**-exprs, which means that maintaining the context is efficient.

2. The second category of rewrites are *inference* rewrites. These rewrites require the *complete* context with all conjunctive **R**-exprs. Because all conjunctive **R**-exprs are tracked in the context, this is more "*heavy*" in that **R**-exprs and indexes on **R**-exprs are required. Furthermore, matching against the context is much more expensive. When the system only checks variable assignments, it can perform an $O(1)$ time lookup in a map. However, when matching with **R**-expr in the context, it must scan through many potential matches.

   Because these rewrites are expensive to match, they are run with the lowest priority.

   The reason these rewrites have been named *inference* rewrites is that they usually *infer* the existence of new constraints using propagation. For example, rewrite rule 34 combines two `lessthan` constraints, e.g. `lessthan(A,B)*lessthan(B,C)` $\xrightarrow{34}$`lessthan(A,B)*lessthan(B,C)*lessthan(A,C)`. In fact, other rewrite systems sometimes call these kinds of rewrites *propagators* [72].

3. The third category of rewrites are *constructor* rewrites. These rewrites are run with the highest priority. They are run immediately when an **R**-expr is constructed. These rewrites are mostly used to keep the **R**-expr tidy. For example, rewrite rules 13 to 15 are the identities involving `0` and `1`, and will be removed from a conjunction immediately rather than being kept around in

the **R**-expr representation (e.g. `0*R→0`).

With this division of rewrite methods, we have two different versions of the SIMPLIFY function: SIMPLIFYONLYFAST and SIMPLIFYALLREWRITES. These new functions behave the same as the SIMPLIFY function from chapter §8, in that they take a context and an **R**-expr and rewrite it as another **R**-expr, but SIMPLIFYONLYFAST only invoke the *faster-to-match* standard rewrites and SIMPLIFYALLREWRITES invokes both standard and inference rewrites. The constructor rewrites are invoked immediately upon the construction of an **R**-expr, and therefore are "*hidden*" and do not require a "visually obvious" call to SIMPLIFY in the code. The function SIMPLIFY refers to whichever version of SIMPLIFY is currently "active".

The SIMPLIFYNORMALIZE function is redesigned to first complete all of the *fast* rewrites by fixed-pointing the SIMPLIFYONLYFAST function and then calling the SIMPLIFYALLREWRITES function once to check if there are any inference rewrites that can be performed. It will then go back to running only the fast rewrites until all of those are completed again. This modified version of SIMPLIFYNORMALIZE is shown in algorithm 7.

11.5 Declaration of R-exprs and Rewrites

As hinted at in the overview (section §11.2), we are going to make heavy use of Clojure macros in the implementation of Dyna. The reason for this is that Dyna,

```
1:  function SimplifyNormalize(R)
2:      $\mathcal{C} \leftarrow \langle\{\} : \texttt{Var} \mapsto \texttt{val}\}, \{\texttt{R}\}\rangle$  ▷ Initialize empty context, ⟨ variables to values, R-exprs ⟩
3:      repeat
4:          repeat
5:              $R_{old} \leftarrow R$
6:              $R \leftarrow$ SimplifyOnlyFast($R$, $\mathcal{C}$)      ▷ Perform only rewrites that are “fast”
7:          until $R_{old} = R$           ▷ Reached a fixed-point, no more fast rewrites
8:          $R \leftarrow$ SimplifyAllRewrites($R$, $\mathcal{C}$)        ▷ Perform all rewrites
9:      until $R_{old} = R$               ▷ Reached a fixed-point, no more rewrites
10:     return R
```

Algorithm 7. Simplify Normalize with different priorities for fast rewrites.

built on **R**-exprs, is made up of hundreds of different **R**-expr kinds and rewrite rules. Hence, we want to minimize the amount of code we need to write for each **R**-expr and rewrite rule.

11.5.1 Declaration of R-expr Kinds

To define **R**-expr kind in Clojure, the implementation provides the `def-base-rexpr` macro. This macro creates a class[165] that is specialized for the implementation of a particular **R**-expr kind. It also sets up internal data structures associated with the **R**-expr kind that are used for relevant meta-data, registering the **R**-expr kind, and registering rewrites for the **R**-expr kind.

Using `def-base-rexpr` we can define some core **R**-expr kinds as follows:

```
(def-base-rexpr conjunct [:rexpr-list args])
(def-base-rexpr unify [:var a :var b])
(def-base-rexpr proj [:hidden-var var :rexpr body])
```

[165]This is a Clojure `deftype` which gets compiled into a Java class file.

The first argument is the name of the **R**-expr, and the second argument is the names of fields associated with each **R**-expr, along with types for each field denoted using a Clojure keyword. The type annotation on the fields automatically configures how different methods on the **R**-expr behave. For example, `:var` represents a field that holds value types, which can be variables or constant values. This automatically configures the "*rename variables*"[166] and "*list all variable*"[167] functions to include the `:var` annotated fields. Fields marked `:hidden-var` contain variables that are projected out, and these variables are removed from the set of variables returned by "list all variables". Similarly, `:rexpr` represents a single **R**-expr, and `:rexpr-list` represents an array of **R**-exprs.

This annotation using keywords handles around 90% of all cases that we need when defining **R**-exprs. We can also override the definition of any method in the case that we need to change something that only applies to one **R**-expr kind.

For example, we override the `is-constraint?`[168] method to check that all of the sub-**R**-exprs contained in a conjunction are constraints, in which case their conjunction is also a constraint (line 479):

```
477 (def-base-rexpr conjunct [:rexpr-list args]
478   (is-constraint? [this]
479     (every? is-constraint? args)))
480 (def-base-rexpr unify [:var a :var b]
481   (is-constraint? [this]
482     true))
```

[166]"Renaming variables" was previously denoted with the $R\{X \mapsto Y\}$ notation.
[167]The "list all variables" was previously denoted with $\text{vars}(R)$.
[168]Recall that constraints are **R**-exprs whose multiplicity is at most 1. (section §5.2.2.4)

11.5.2 Declaration of Built-Ins

Most of the **R**-expr kinds and rewrite rules that we have to define correspond to built-in **R**-exprs (sections § 5.2.2.3 and 6.3). Built-ins define numerical operations (`plus`, `times`, `cosine`, etc.), logical operations (`lessthan`, `lessthaneq`, etc.), and other primitive operations such as string concatenation.

To help define all of these **R**-expr kinds, we have another special macro: `def-builtin-rexpr`, which calls `def-base-rexpr` and `def-rewrite`, which will be defined shortly.

As an example, the *complete* definition of the `plus(·,·,·)` **R**-expr is as follows:

```
483 (def-builtin-rexpr plus 3
484   (:allground (= v2 (+ v0 v1)))
485   (v2 (+ v0 v1))
486   (v1 (- v2 v0))
487   (v0 (- v2 v1)))
488
489 (def-user-term "+" 2 (make-plus v0 v1 v2))
490 (def-user-term "-" 2 (make-plus v2 v1 v0))
```

The `def-builtin-rexpr` macro will define a `plus` **R**-expr which contains 3 variable fields which are automatically named `v0`, `v1` and `v2`. Lines 484 to 487 define the different rewrite rules we have for `plus`. Line 484 corresponds with rewrite rules 26 and 27 where all the arguments are ground (e.g. `plus(1,2,3)`$\rightarrow$`1`). Similarly, lines 485 to 487 corresponds with rewrite rules 28 to 30 which define the rewrites for performing computation using `plus`. The variable that appears first in the parentheses is the variable that is assigned by the computation. The second expression in the parentheses defines how the computation is performed. For example, when the system rewrites `plus(1,2,X)`, it will have `v0 = 1` and `v1 = 2`. The variable `v2` corresponds with the variable `X` and is assigned the value returned from `(+ 1 2)`. Hence, this is equivalent to rewriting `plus(1,2,X)`$\to$`(X=3)`.

Lines 489 and 490 define the `plus` **R**-expr under the names accessible from the Dyna source code. The '+' function (in the Dyna source) takes two arguments and returns the third argument. Represented as `v0,v1` and `v2` respectively. Recall from section §7.1.4, I mentioned that we do not have a subtraction **R**-expr, but instead use the `plus(·,·,·)` **R**-expr with its arguments rearranged. This is what is happening on line 490, with the `v0` and `v2` arguments switched.

The function `make-plus` is the constructor for the `plus` **R**-expr. It is defined by the `def-base-rexpr` macro. The `make-NAME` functions internally run any constructor rewrites that are marked to run when an **R**-expr is constructed.

11.5.3 Declaration of Rewrite Rules

One thing I attempted to make easy is the declaration of rewrite rules. We have literally *hundreds* of rewrite rules that need to be implemented. In previous rewrite-based prototypes of Dyna that I developed, rewrites with large implementations frequently resulted in difficult-to-find bugs.[169] Hence, having a short implementa-

[169]A bug might only be obvious after many subsequent steps of rewriting have been performed, which makes it very difficult to locate a bug

tion for each rewrite makes it easier to visually inspect[170] that a rewrite is correct.

The core mechanism for defining rewrites is the `def-rewrite` macro. This macro takes a number of different keyword arguments and a Clojure expression, which computes the result of a rewrite. For the remainder of this section, I will illustrate how the `def-rewrite` macro is used, to give an idea of what features were necessary to concisely define **R**-expr rewrite rules.

11.5.3.1 First Declaration of a Rewrite Rule

We will start by looking at the `def-rewrite` definition for rewrite rule 28, which performs the computation for built-in `plus` (e.g. `plus(1,2,X)`→`(X=3)`). Note: these rewrites on `plus` are automatically generated by the `def-builtin-rexpr` defined above on line 485 and do not appear directly in the implementation. I have chosen to present rewrite rule 28 because it is a simple rewrite which is useful for illustration purposes.

```
(def-rewrite
  :match (plus (:ground A) (:ground B) (:free C))
  (make-unify C (make-constant (+ (get-value A) (get-value B)))))
```

Figure 11-3. Basic version of `def-rewrite` for rewrite rule 28, e.g. `plus(1,2,X)`→`(X=3)`. Prolog would notate this as `plus(+A,+B,-C)`.

In figure 11-3, the `:match` keyword argument specifies what **R**-expr this rewrite

[170]At this time, there is no automatic checking that a rewrite is implemented correctly. This would require some executable representation of the semantic definitions (section §5.2) other than the rewrite rules, which we want to check.

rules will match against. The arguments `A, B` and `C` are matched to the variables on the `plus(A,B,C)` **R**-expr. The notation used by `:match` is *positional*, in that it is written in the order in which fields on the **R**-expr were defined. Internally, this generates code that directly accesses the relevant field on the `plus` **R**-expr class.[171] The keywords `:ground` and `:free` are annotations that are used to match against the variable. `:ground` and `:free` cause the **R**-expr matcher to check the context $\mathcal{C}$ to see if there is an assignment to a variable. We have more keywords defined to match more complex scenarios.

Rewrites return an **R**-expr that will replace the **R**-expr in the newly generated **R**-expr data structure returned by SIMPLIFY. This is illustrated by line 493, which returns the equality constraint (e.g. `(X=Y)`), written as `unify` in the Dyna implementation (section §5.2.2.1). The variable `C` already is a variable[172] so it can go into the first argument of `unify`, which expects a value type. For the second value, the system will compute $A+B$, which will be a numerical value. First, it needs to read the value of `A` and `B`. This is done using `get-value`, which will either get the value directly out in the case that it is a constant or read from the context $\mathcal{C}$ in the case that it is variable. The context $\mathcal{C}$ is globally[162] accessible, hence usable by `get-value`. The `make-constant` constructor returns a value type, which is a constant. Hence, the value assigned to `C` is embedded directly inside the `unify`

[171]This is the same as doing `RexprObject.fieldName` in Java. Coding this so that there is no indirection through method calls turned out to be very important for making this matching fast.

[172]Because `C` is matched as `:free`, we know it must be a variable. In general, the **R**-expr `plus` allows `C` to be a value type that can be a constant or a variable.

R-expr.

11.5.3.2 Assignment Rewrites

The above definition of rewrite rules is very general in that it matches **R**-exprs and returns an **R**-expr. However, we have certain rewrite patterns that are sufficiently frequent that we provide special handling. For example, we can assign a value to a variable directly as in figure 11-4.

```
494 (def-rewrite
495   :match (plus (:ground A) (:ground B) C)
496   :assigns-variable C
497   (+ (get-value A) (get-value B)))
```

Figure 11-4. Rewrite rule defined which returns a value to be assigned to the variable C, instead of returning an **R**-expr.

On line 497, we return the *value* to assign to the variable C instead of an **R**-expr which represents the assignment.

The definition in figure 11-4 is *conceptually* equivalent to figure 11-3. However, it is preferable to use figure 11-4. The reason is that figure 11-4 provides the system with additional metadata on what the rewrite does. In this case, the system can analyze the rewrite in figure 11-4 to identify that C is assigned a value after this rewrite runs. Trying to extract this information from figure 11-3 would require analysis of the Clojure source code that defines the rewrite. This information will be used when compiling a sequence of rewrites in chapter §12.

Furthermore, the rewrite in figure 11-4 handles two rewrites at once. Observe that on line 495, I have removed the `:free` annotation for the variable `C`. If the variable `C` has already been assigned some value, the `:assigns-variable` behaves as an equality check (rewrite rules 26 and 27). If not, it assigns a value to `C` (rewrite rule 28).

11.5.3.3 Handling Invalid Inputs

In the above definition given in figure 11-4, we no longer return an **R**-expr. This means that we cannot represent a failure or an invalid input by returning the **R**-expr `0`.[173]

To handle this, we have *two* ways to represent failure. The first is returning the `0` **R**-expr, and the second is throwing a `UnificationFailure`[174] exception[175] as in figure 11-5.

[173]Recall that an **R**-expr corresponds with a bag relation where 1 is returned when the assignment to the variables is contained in the bag (e.g. section §5.2.2.3). Hence, if we pass values to the `plus(·,·,·)` **R**-expr which are *not supported*, then this corresponds with multiplicity `0` (e.g. `plus("hello", foo[1,2], X)`$\to$`0`).

[174]The name `UnificationFailure` was chosen as it aligns with the idea of unification failure in Prolog or CLP, which causes the system to backtrack and try another branch of a disjunction.

[175]Dyna runs on top of Java, and throwing an exception in Java is fast. This design may not work as well if Dyna is implemented with another programming language.

```
498 (def-rewrite
499   :match (plus (:ground A) (:ground B) C)
500   :assigns-variable C
501   (let [val-A (get-value A)
502         val-B (get-value B)]
503     (when (or (not (number? val-A))
504               (not (number? val-B)))
505       (throw (UnificationFailure.)))
506     (+ val-A val-B)))
```

Figure 11-5. Rewrite rule checking inputs and throwing a `UnificationFailure` on line 505. Admittedly, this rewrite is checking the *type* of the value assigned to the variables `A` and `B` (lines 503 and 504) before it throws the exception, but recall that Dyna handles type errors by rewriting as `0`, and it is up to the system to detect this and report the `0` to the user as a type error (rewrite rule 26 and section §2.8.3).

Throwing a `UnificationFailure` and returning the `0` multiplicity are used interchangeably. However, there are some advantages to throwing the `UnificationFailure` exception. The exception stops rewriting the current **R**-expr and parent **R**-expr that would also be rewritten as `0` as a result. For example, suppose that the system is rewriting an **R**-expr like `S+proj(X,R1*R2*R3*`$\cdots$`*Rn))`, if `R1` is rewritten as `0` (`R1`$\rightarrow$`0`), then we would like the system to immediately stop rewriting the conjunction (`R1*R2*`$\cdots$) and the projection, and returns control flow to rewriting the disjunction.

11.5.3.4 Modifying the Context

Rewrite rules can access the context $\mathcal{C}$ as it is tracked with a global variable.[176] We have already seen `get-value`, which can read the value of a variable from the context. We can also modify the context in place using methods like `set-value!`, which tracks the assignment of a variable directly into the context.

Figure 11-6 shows the value assigned to the variable `C` being set directly into the context. The rewrite rule can therefore return the multiplicity `1`, instead of assignment to the variable `C`. This is fine, as we do *not* require a representation for `C`'s value inside of the **R**-expr itself. Using the context in this way enables us to track the assignments to variables and avoid having to shuffle around equality constraints inside of the **R**-expr.

```
507 (def-rewrite
508   :match (plus (:ground A) (:ground B) (:free C))
509   (let [val-A (get-value A)
510         val-B (get-value B)]
511     (when (or (not (number? val-A))
512               (not (number? val-B)))
513       (throw (UnificationFailure.)))
514     (set-value! C (+ val-A val-B))
515     (make-multiplicity 1)))
```

Figure 11-6. The variable `C` is set directly into the context $\mathcal{C}$ (line 514) without a corresponding representation in the **R**-expr, as was done in figure 11-3.

[176] Previously in chapter §8 I denoted the context as being passed as an extra argument to *all* functions. This quickly becomes cumbersome, so it is easier to reference the context using a global (thread-local) variable.

11.5.3.5 Rewrite Priorities

As stated in section §11.4, we have three different priorities for which rewrites can run at: *standard*, *construction*, and *inference*. By default, a rewrite is created with the standard priority. A rewrite's priority is set using the `:run-at` keyword argument as in figure 11-7.

```
516 (def-rewrite
517   :match {:rexpr (unify (:free A) (:ground B))
518           :check (has-context?)}
519   :run-at [:construction :standard]
520   (do
521     (set-value! A (get-value B))
522     (make-multiplicity 1)))
```

Figure 11-7. A rewrite on the unify/equality constraint, which is run when it is first constructed. This rewrite directly adds the assignment of a variable in the context using `set-value!`, as described above. With this rewrite included and running when `unify` is constructed, this makes the rewrites defined in figure 11-3 and figure 11-6 equivalent. Note that construction rewrites run when the **R**-expr is constructed, which means there might be an active context that can track the assignment to a variable. Hence, line 518 defines a side condition as executable Clojure code that checks that there exists an active context before this rewrite matches.

Construction rewrites like figure 11-7 are used to keep the **R**-expr tidy. In this case, we do not want to have equality **R**-exprs that assigns a value to a variable represented in the **R**-expr. Instead, we prefer to track assignments using the context. Hence, this rewrite "eliminates" the equality constraint from the **R**-expr and instead tracks the assignments to variables using the context.

11.5.3.6 Inference Rewrites

Inference rewrites are the lowest-priority rewrites as they take the most effort to match. These rewrites have access to the *complete* context that tracks conjunctive **R**-exprs in addition to assignments to variables. Figure 11-8 shows rewrite rule 34, which combines two `lessthan` constraints to infer a third `lessthan` constraint.

```
523 (def-rewrite
524   :match {:rexpr (lessthan A B (is-true? _))[177]
525           :context (lessthan B C (is-true? _))}
526   :run-at :inference
527   :infers (make-lessthan A C (make-constant true)))
```

Figure 11-8. Implementation for rewrite rule 34, which combines two `lessthan` constraints together to infer a third. The `:run-at` (line 526) specifies that this runs during inference with the complete context. The `:context` expression in the `:match` checks in the context for a matching **R**-expr. The variable `B` used on lines 524 and 525 is *required* to be the same variable.

Note that the match expression on lines 524 and 525 only lists *one* **R**-expr under the `:rexpr` matcher, and the second under the `:context` matcher. The reason is that Simplify recurses through the **R**-expr and performs matches quickly against the **R**-exprs it encounters. In other words, once the first **R**-expr on line 524 is matched, Simplify *then* checks the context for a matching `lessthan` constraint. As such, a more *accurate* way to write rewrite rule 34 would be:

[177] Note, that `(lessthan A B (is-true? _))` is the trinary version of `lessthan` discussed in section §7.1.4. The `is-true?` annotation is similar to `:ground`. However, it *also* checks value is *true*. The underscore `_` is a placeholder variable.

```
(def-rewrite
  :match-combines [(lessthan A B (is-true? _))
                   (lessthan B C (is-true? _))]
  :run-at :inference
  :infers (make-lessthan A C (make-constant true))
```

Figure 11-9. `:match-combines` expanded this rewrite rule into multiple rewrite rules of the form shown in figure 11-8. For each defined rewrite, one **R**-expr will be matched in the local context, and the other **R**-exprs will be matched using the context.

$$\texttt{lessthan(A,B)} \xrightarrow{34} \texttt{lessthan(A,B)*lessthan(A,C)} \quad \begin{array}{l} \textbf{if } \texttt{lessthan(B,C)} \in \mathcal{C} \\ \textbf{and } \texttt{lessthan(A,C)} \notin \mathcal{C} \end{array}$$

where `lessthan(B,C)` is checked if it is contained in the context *as a side condition*.

Additionally, observe that rewrite rule 34 checks that the newly inferred `lessthan(A,C)` does not already exist. This is done to avoid inferring the same constraint multiple times. This check is handled by the `:infers` keyword on line 527.

11.5.3.7 Combining R-exprs for Inference Rewrites

Often times when running inference rewrites that check the context, we do not care about which **R**-expr appears in the context and which **R**-expr was directly matched against. To handle this, we have the `:match-combines` keyword argument, which takes a list of **R**-exprs that must be matched. The rewrite is expanded into several simpler rewrites of the form in figure 11-8.

11.5.3.8 Recursive Rewrites

As stated in section §8.2.2, we have special "rewrite rules" for recursive **R**-exprs (such as conjunction) that recursively invoke SIMPLIFY. To this end, we can use the function `simplify` as in figure 11-10, which automatically references the current simplify function (either SIMPLIFYONLYFAST or SIMPLIFYALLREWRITES).

```
533 (def-rewrite
534   :match (conjunct (:rexpr-list Rs))
535   (make-conjunct (map simplify Rs)))[178]
```

Figure 11-10. Recursive rewrite rules call `simplify` recursively applies rewrites to sub-**R**-exprs.

Recall that in section §11.5.3.3, I said that we throw a `UnificationFailure` exception instead of returning a `0` multiplicity **R**-expr. Throwing an exception immediately aborts the loop on line 535 without any special handling.

11.5.4 Conclusion of R-exprs and Rewrites Declarations

This section has served both as documentation for our Dyna implementation and as inspiration for anyone developing their own rewrite-based system. Given the number of rewrite rules required to implement a rewrite-based language, such as Dyna, I believe that it is critical to have a concise way to define rewrite rules. From experience, I can say that more than 90% of the rewrite rules can be handled

[178] The method map in Clojure returns a lazy sequence. We need to force the sequence to evaluate immediately using another Clojure function like vec or `doall`. I have omitted this in this example as it does not contribute to understanding the rewrite definition.

succinctly, given the right abstraction. Allowing the remaining 10% of rewrite rules to fall back on the host language works well to ensure that the rewrite system is "*sufficiently powerful*".

11.6 Efficient R-exprs Kinds

At the beginning of this chapter, I gave a definition for an "efficient" program (section §11.1.1). So far, we have only seen the same "inefficient" **R**-exprs we have had since chapter §5. To make **R**-exprs efficient, we are going to replace some **R**-expr kinds with **R**-exprs that are conceptually equivalent but have *more* efficient implementations. The way we accomplish this is by *combining* two or more **R**-expr kinds into a single larger **R**-expr. This will allow us to handle frequent cases with specialized code.

The two efficient **R**-expr kinds we currently have are *efficient disjunction*, which merge disjunctions and equality constraints, and *efficient aggregation*, which merge aggregation and projection. Future work should consider the implementation of additional efficient **R**-expr kinds.[179]

11.6.1 Efficient Disjunctions

The first efficient **R**-expr kind that we will look at is the efficient disjunction. We care a lot about the efficiency of disjunctions as they occur very frequently when

[179]Some suggestions for efficient **R**-expr kinds would be *dense matrix* representations or *GPU backed* **R**-exprs and rewrites. (sections § 16.1 and 16.7)

```
a(0,0) = 1.
a(0,1) = 2.
a(1,0) = 3.
a(1,1) = 4.
```

(a) Dyna

```
a(X,Y, Val) → (Val=only(Inp,
  (X=0)*(Y=0)*(Inp=1)+
  (X=0)*(Y=1)*(Inp=2)+
  (X=1)*(Y=0)*(Inp=3)+
  (X=1)*(Y=1)*(Inp=4)))
```

(b) R-expr

Figure 11-11. Example Dyna program translated into an **R**-expr showing how ground values can be a frequent **R**-expr we must handle. Each rule in the Dyna program is converted into a branch of the disjunction, with the ground values present in the rules present in the **R**-expr as ground assignments to variables.

translating a Dyna program into **R**-exprs, e.g., figure 11-11.

To start, the standard "inefficient" disjunction is defined as follows:

```
(def-base-rexpr disjunct [:rexpr-list args]
  ;; overridden functions on disjunction omitted
  )
```

Figure 11-12. Standard "inefficient" implementation of disjunction.

The declaration in figure 11-12 matches the disjunction we have used since chapter §5, with a disjunction of the form `R1+R2+R3+R4` being represented as a list of `[R1,R2,R3,R4]` inside of a disjunction **R**-expr kind. Unfortunately, this design has a fatal flaw. Finding the right disjunct requires scanning the *entire* list of disjunctions and applying SIMPLIFY to each sub-**R**-expr. For example, if we have the disjunct `(X=1)*R1+(X=2)*R2 +(X=3)*R3+(X=4)*R4`, then finding a particular value of `X` requires us to perform a linear time scan through all of the disjunctive **R**-exprs.

To make the disjunction more efficient, we create an *index* for the assignment to the variable `X`. When `X` has a known value,[180] we can access the relevant sub-**R**-expr of the disjunct in $O(1)$ time by finding it using a data structure such as a hash table.

11.6.1.1 Requirements on the Efficient Disjunction Data Structure

To implement an efficient disjunction, which merges disjunction and ground assignments to variables, there are a few requirements that must be met. Currently, the implementation only has hash-table-based tries as the only data structure backing the efficient disjunction (section §11.6.1.2). Future work should consider adding alternate implementations for disjunctions (section §16.1).

The data structure backing disjunctions combined with ground equality constraints needs to support the following:

1. In tracking the value assigned to a variable, the data structure needs to *allow* any ground value. For example, suppose we have an array containing disjunctive **R**-exprs, where the index in the array corresponds with the assignment to a variable `X`. This would be able to represent the cases where `X` is assigned a small integer, such as `(X=0)*R1+(X=2)*R2+`$\cdots$. However, the data structure also needs to handle the cases where `X` is not a small integer, such as `(X="hello")*R3`. This can be done by combining the array with some kind of auxiliary data structure to handle the cases where `X` is not assigned an integer.

[180] Meaning that there is a sub-**R**-expr of the form `(X=7)`, where 7 is the known value.

2. There needs to be some way to represent a "FREE VALUE". The reason is that we can have a disjunct like (X=1)*R1+R2 where there is no known value of X on the second branch of the disjunct. One solution is to represent the "FREE VALUE" internally using a special value that does not appear in $\mathcal{G}$ and, therefore, can be distinctly recognized. I will represent the "FREE VALUE" case using a question mark "?". This means that the previous **R**-expr can be "*normalized*" into the disjunct (X=1)*R1+(X=?)*R2.[181]

3. There needs to be some way to efficiently make queries against the data structure that include the "FREE VALUE". This means queries of the form:

   $$\big((X_1{=}1)\ \textbf{or}\ (X_1{=}?)\big)\ \textbf{and}\ \big((X_2{=}2)\ \textbf{or}\ (X_2{=}?)\big)\ \textbf{and}\ \cdots\ \textbf{and}\ \big((X_n{=}\text{"N"})\ \textbf{or}\ (X_n{=}?)\big)$$

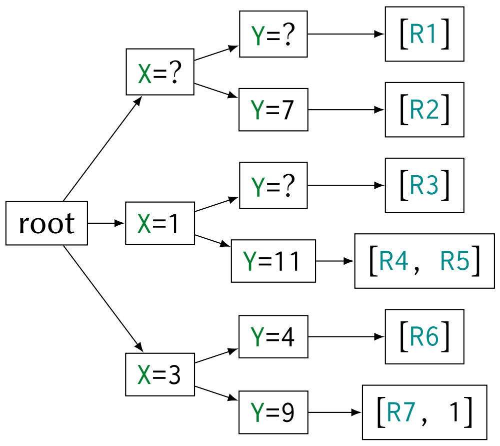
   The reason is that given an assignment $(X_1{=}1){*}(X_2{=}2){*}\cdots(X_n{=}\text{"N"})$, we need to match *all* disjunctive branches that unify with this assignment. This includes branches where the assignment to a variable is known (e.g. $(X_1{=}1){*}(X_1{=}1)$), and branches where there is no assignment to a variable (e.g. $(X_1{=}?)$). The *only* case that can be eliminated is when the assignment to a variable *differs* from the value we are querying for (e.g., $(X_1{=}1){*}(X_1{=}\text{"not 1"}){\to}0$).

4. There needs to be a way to "store" the **R**-expr in the data structure. In our case, the data structure is in memory, so we can store an opaque pointer to the

[181]The (X=?) is *not* an equality constraint, as we say that $(X{=}?){\to}1$ in all cases. This means that we can have something like $(X{=}7){*}(X{=}?){\to}(X{=}7)$. I have chosen to write this with an explicit value "?" rather than using an **R**-expr like (X=X) as this more closely matches the implementation.

```
R1 +
(Y=7)*R2 +
(X=1)*R3 +
(X=1)*(Y=11)*R4 +
(X=1)*(Y=11)*R5 +
(X=3)*(Y=4)*R6 +
(X=3)*(Y=9)*R7 +
(X=3)*(Y=9)
```

(a) Standard **R**-expr



(b) Efficient Disjunctive Trie **R**-expr

Figure 11-13. A disjunctive **R**-expr in **(a)** is converted into a trie in (b). Each level of the trie is associated with one of the variables (either X or Y in this example). Each level of the trie is an immutable hash map. This allows for efficient $O(1)$ retrievals in the case where a variable's value is known. A special value is associated with the case where the value of the variable is unknown, shown here as a question mark '?'. The leaf of the trie is a list that contains one or more disjunctive **R**-exprs.

R-expr. Future work could implement disjunctions with a disk-backed data structure or SQL database-backed disjunctions, in which case, they will have to implement serialization of **R**-exprs.

11.6.1.2 Efficient Disjunction using a Trie

The only data structure that we currently have for efficient disjunction is trie. Each level of the trie corresponds to a different variable.[182] The leaf of the trie represents a disjunction of one or more sub-**R**-exprs. The **R**-exprs in the leaves are treated as opaque pointers to **R**-exprs. This is shown in figure 11-13.

[182]The variable order is chosen arbitrarily. Future work should consider developing a better heuristic for this.

By having lists at the leaf of the trie instead of an arbitrary **R**-expr we can completely replace the standard disjunction with the trie-based disjunction. This is useful as it allows us to integrate the trie disjunction into the **R**-expr rewrite with a single rewrite rule that runs at the construction of the standard disjunction. This automatically convert from the standard disjunction to the trie-backed disjunction, as shown in figure 11-14.

```
543 (def-base-rexpr disjunct-efficient [:var-list trie-variables
544                                     :prefix-trie trie])
545 (def-rewrite
546   :match (disjunct (:rexpr-list Rs))
547   :run-at :construction
548   (let [variable-order (vec (exposed-variables[183] rexpr))]
549     (make-disjunct-efficient variable-order
550                              (convert-to-prefix-trie variable-order Rs))))
```

Figure 11-14. Declaration of the efficient disjunct, and a rewrite rule which converts from the standard disjunct to the efficient disjunct upon constructions. The variables on the trie are all exposed variables from the sub-**R**-exprs. The variable order is chosen arbitrarily (line 548). Future work should consider developing a variable ordering heuristic.

The path from the root of the trie to the leaves will have one level for each exposed variable in $\text{vars}(\cdot)$. This allows **R**-exprs to be placed onto the *most informative* path of the trie without having secondary variable assignments contained in the **R**-exprs in the leaves. To determine the assignments to the variables, recall that all the assignments to variables are in the context $\mathcal{C}$ and eliminated from the **R**-expr itself

[183] `exposed-variables` is the name in the implementation for $\text{vars}(\cdot)$ from chapter §5.

(figure 11-7). This means that after invoking SIMPLIFY on each **R**-expr in the leaves, we are guaranteed that there are no assignments **R**-exprs contained in the resulting **R**-expr, and that all assignments to variables can be found using context $\mathcal{C}$. Any variable without a known assignment, as tracked by the context, is assumed to not have a value, and is placed under the "`FREE VALUE`", '?' branch.

11.6.2 Efficient Memoization uses Tries

In chapter §10, I introduced memoization using only abstract **R**-exprs. In chapter §10, I said that when an upstream dependency is changed, the system *recomputes* the **R**-expr using its original definition. If the system recomputes the entire **R**-expr, this can be quite inefficient. The memoized **R**-expr frequently represents hundreds of different disjunctive **R**-exprs (e.g. `(X=1)*R1+(X=2)*R2+···+(X=n)*Rn`).

To make memoization realistic, we will only recompute some subset of the memoized **R**-exprs. In other words, if the currently memoized **R**-expr is `(X=1)*R1+(X=2)*R2+···+(X=n)*Rn`, then we might say that values which are memoized for `(X=2)` are invalid and need to be recomputed. In which case, the system will modify the memoized **R**-expr so that it gets `(X=1)*R1+(X=2)*R2new+···+(X=n)*Rn`, where `R2` is replaced with `R2new`.

To implement this, we no longer treat the memoized **R**-expr as a generic **R**-expr. Instead, we explicitly use the efficient disjunctive trie (section §11.6.1) as the memoized **R**-expr. The trie supports "special rewrites" that implement efficient modification of the trie. These rewrites are exemplified by rewrite 86 and 87, and allow us to delete entries matching a particular key[184] from the trie, getting a new trie, or add **R**-exprs to the trie.

R_{trie} `* if((X=1)*(Y=2), 0, 1)` $\xrightarrow{86}$ $\mathsf{R}_{new\ trie}$ ▷*Delete from trie*

R_{trie} `+ (X=2)*(Y=3)*S` $\xrightarrow{87}$ $\mathsf{R}_{new\ trie}$ ▷*Add to trie*

These "rewrite rules" make use of the trie implementation to share the internal structure with the existing trie as much as possible. These rewrites are invoked explicitly through special methods that are currently only supported by the trie.

11.6.3 Efficient Aggregation & Projection

The second efficient **R**-expr kind we have is for aggregation and projection. So far, our rewrite strategy for projection and aggregation requires many steps of rewriting to rearrange them into a form that can be matched by the appropriate rewrite rules (rewrite rules 49 and 55). When considering aggregation **R**-exprs, there are often nested disjunctions and projections that need to be rewritten away before we can obtain the result of aggregation. For example, consider the program in figure 11-15.

```
a += b(X).
a += c(X,Y)*d(Y,Z).
```

(a) Dyna

```
a(Val) → (Val=sum(Inp,
  proj(X, b(X, Inp))+
  proj(X,proj(Y,proj(Z,proj(Tmp1,proj(Tmp2,
    c(X,Y,Tmp1)*d(Y,Z,Tmp2)*
    times(Tmp1,Tmp2,Inp))))))))
```

(b) R-expr

Figure 11-15. Dyna program with two contributions to the rule 'a'.

[184]The key does not have to specify all of the variables.

This program has two contributions on lines 551 and 552 with each line having variables that are projected out. With the rewrite rules we have so far, we have to expand the projections using all possible assignments to the variables to solve this **R**-expr. (E.g. `proj(X,((X=1)+(X=2))*R)`$\rightarrow$`proj(X,(X=1)*R)+proj(X,(X=2)*R)`$\rightarrow$ (`R`$\{$`X`$\mapsto 1\}$) + (`R`$\{$`X`$\mapsto 2\}$))

Ideally, the system would execute the **R**-expr from figure 11-15 in a manner similar to the procedural implementation of the matrix-vector product in algorithm 5. Conceptually, the nested loops for figure 11-15 look something like the code in figure 11-16.

```
1:  $a \leftarrow 0$                                  ▷ Initialize a container to hold the result of aggregation
2:  for $x \in$ DOMAIN(X, b(X)) :  ▷ Loop over the domain of projected variables from line 551
3:      (Inp=$i$) ← "SIMPLIFY((X=$x$)*b(X, Inp))"        ▷ Evaluate called user-defined rule
4:      $a \leftarrow a + i$                       ▷ Record the contribution directly into the result container
5:  for $x \in$ DOMAIN(X, c(X,·)) :  ▷ eval line 552, start enumerating proj-ed var X's domain
6:      for $y \in$ DOMAIN(Y, c(X,Y)) :                    ▷ Enumerate projected var Y's domain
7:          ▷ Evaluate a R-expr once enough arguments are known to trigger a rewrite
8:          (Tmp1=$t_1$) ← "SIMPLIFY((X=$x$)*(Y=$y$)*c(X,Y, Tmp1))"
9:          for $z \in$ DOMAIN(Z, d(Y,Z)) :                ▷ Enumerate projected var Z's domain
10:             (Tmp2=$t_2$) ← "SIMPLIFY((Y=$y$)*(Z=$z$)*d(Y,Z, Tmp2))"
11:             $i \leftarrow t_1$ "*" $t_2$                      ▷ Evaluate times(Tmp1,Tmp2,Inp)
12:             $a \leftarrow a + i$          ▷ Once all R-exprs are evaluated, record the contribution
13: return $a$
```

Figure 11-16. An informal pseudocode example of the *idealized* evaluation of the rule 'a' from figure 11-15. First, a container is initialized to hold the result of aggregation on line 1. Then, the system sequentially runs the sub-**R**-exprs of the disjunction. For the first disjunct, the system loops over the entire domain of the projected variable X, line 2. How the system figures out the domain of the variable will be discussed in section §11.7. During each iteration, the **R**-expr that represents a rule is SIMPLIFied. The resulting value is directly saved directly into the aggregator's accumulator variable a, line 4.

For the second disjunct, we have multiple variables projected out, X, Y and Z. The system loops through the domains of the first two variables on lines 5 and 6. Once it has *enough* information to evaluate an **R**-expr, it is *immediately* evaluated, as on line 8. The remaining variables are still looped over (line 9) with the remaining **R**-exprs also being evaluated (line 10). Just like before, the contribution to aggregation is saved directly in the accumulator variable a (line 12).

Finally, after evaluating the entire body of the aggregator, the result of the aggregation is returned (line 13).

To make the execution with **R**-exprs like what is shown in figure 11-16, we will introduce two different efficient **R**-expr kinds. The first **R**-expr kind will be the *outer aggregator*, which initializes the container for aggregation and returns the

```
a(Val) → (Val=sum_aggregator_outer(
  sum_aggregator_inner([X], Inp,          ▷List of projected variables and input variable
    b(X, Inp)     )+                                  ▷Body of first inner aggregator
  sum_aggregator_inner([X,Y,Z,Tmp1,Tmp2], Inp,
    c(X,Y,Tmp1)*d(Y,Z,Tmp2)*                         ▷Body of second inner aggregator
    times(Tmp1,Tmp2,Inp))    ))
```

Figure 11-17. The **R**-expr from figure 11-15 with its aggregator and projections replaced with the efficient aggregator.

final result of the aggregation. The outer aggregator corresponds with a *term*, and will combines the result from multiple rules. This corresponds with the solid red lines in figure 11-16 and is represented by lines 1 and 13. The second **R**-expr kind will be the *inner aggregator*, which simultaneously handles projection and nested loops over variables and will also compute the contribution to the aggregator. A single outer aggregator can have multiple inner aggregators internally, as in this example. The inner aggregator corresponds with a single *rule*, and terms can be defined using multiple rules. The inner aggregators correspond with the dashed blue lines in figure 11-16 with lines 2-4 and lines 5-12.

The **R**-expr representation for figure 11-15 with an efficient aggregator is shown in figure 11-17.

Note that in figure 11-16, the aggregator accumulator variable a is initialized by the outer aggregator but used by the inner aggregator. The accumulator variable is passed between the outer and inner aggregator through a “globally”[162] accessible pointer. Hence, the inner aggregator does not have well-defined **R**-expr semantics on its own. Instead, the inner aggregator is only well defined when it is nested inside an outer aggregator as in figure 11-17.

The implementation of the efficient aggregator is similar to the efficient disjunction. We define new **R**-expr kinds that hold the necessary details for the aggregator. A rewrite rule that runs at construction matches the standard aggregator and converts it into the efficient representation (figures 11-18 and 11-19).

```
(def-base-rexpr aggregator-outer [:string operator[185]
                                  :var result
                                  :rexpr body])
(def-base-rexpr aggregator-inner [:hidden-var incoming
                                  :hidden-var-list projected
                                  :rexpr body])
(def-rewrite
  :match (aggregator operator[185] (:var result) (:var incoming)
                     (is-true? is-conjunctive) ;; Recall §6.5.1
                     (:rexpr body))
  :run-at :construction
  (make-aggregator-outer operator result
    (make-aggregator-inner incoming [] body)))
```

Figure 11-18. Declaration of the outer and inner efficient aggregator **R**-expr, and rewrite rule which converts from the standard aggregator into the efficient aggregator.

The inner aggregator also uses several rewrite rules that match the body of the inner aggregator to move the inner aggregator closer to the computation performed by the **R**-expr. The inner and outer aggregators can be separated by disjunctions

[185] In writing **R**-exprs, we have been writing aggregators as `(A=sum(X,R))` where each aggregator is its own **R**-expr kind. There is a single aggregator implementation in the implementation, with different aggregators being distinguished by their name, represented here as `operator`.

```
566 (def-rewrite
567   :match (aggregator-inner (:var incoming) (:var-list projected)
568                            (proj (:var projv) (:rexpr body)))
569   :run-at :construction
570   (make-aggregator-inner incoming (cons projv projected) body))
571 (def-rewrite
572   :match (aggregator-inner (:var incoming) (:var-list projected)
573                            (disjunct Rs))[186]
574   :run-at :construction
575   (make-disjunct
576     (map (fn [R] (make-aggregator-inner incoming projected R))
577           Rs)))
```

Figure 11-19. The inner aggregator rewrites its body to *embeds* itself in the **R**-expr as close as possible to the **R**-expr body which corresponds with a single Dyna rule. It matches against projection (line 567) and merges those variables into the inner aggregator's own projected variable list. For disjunction, it moves towards the leaves of the disjunct (line 572).

and `if`-expressions (memoization). An inner aggregator nested under a disjunction corresponds to a single Dyna rule that partially defines a user-defined term. An example of rewrites that embed the inner aggregator is shown in figure 11-19.

When running the rewrites for the inner and outer aggregators, the outer aggregator sets up the aggregation accumulator where the intermediate result of

[186]There is a similar rewrite rule for `aggregator-inner` which matches against the efficient disjunct. The efficient disjunct rule is a bit more complicated as it has to handle variables that are assigned, as well as the fact that some of those variable assignments might be projected out by the `aggregator-inner`.

[187]The delayed aggregator is an **R**-expr like `ResultVar=sum_aggregator_outer(R+sum_aggregator_inner([], Inp, (Inp=`*accumulator*`)))`, where `R` is the partially rewritten aggregator body as an **R**-expr.

[188]The + on line 604 corresponds with running the addition between the current aggregated value and the value which has been computed from the inner aggregator. In the actual implementation, this would consult the aggregator `operator` rather than being hardcoded for += as done here to simplify this example.

```
578 (def-rewrite
579   :match (aggregator-outer operator result body)
580   (let [;; stash parent frame aggregation accumulator container
581         parent-accumulator (get-global *aggregator-accumulator*)]
582      ;; initialize aggregation container into global
583     (set-global *aggregator-accumulator* nil)
584     (let [new-body (simplify body)           ;; run rewrites on body
585           accumulator (get-global *aggregator-accumulator*)]
586       ;; restore the parent frame aggregation accumulator into
    global
587       (set-global *aggregator-accumulator* parent-accumulator)
588       (if (= new-body (make-multiplicity 0))
589         (do ;; done running the inner aggregators, return the value
590           (make-unify result (make-constant accumulator)))
591         (do ;; inner aggregators not finished, return delayed
592           (make-aggregator-outer[187] operator result
593             (if (= nil accumulator)
594               new-body
595               (make-disjunct [new-body
596                 (make-aggregator-inner (make-constant accumulator)
597                                   [] (make-multiplicity 1))]))))))))
598 (def-rewrite
599   ;; match R-expr like `sum_aggregator_inner([], Inp, (Inp=7))`
600   :match (aggregator-inner incoming []
601                            (unify incoming (:ground value)))
602   (do
603     (set-global *aggregator-accumulator*
604       (+[188] (get-global *aggregator-accumulator*)
605          (get-value value)))
606     (make-multiplicity 0)))
```

Figure 11-20. These example rewrites are only intended to be *illustrative* of how the inner and outer aggregators work together. Many details have been omitted. There is a single global `*aggregator-accumulator*` variable. This works fine, as the inner aggregator is always associated with the nearest outer aggregator. The aggregator accumulator from a higher scope is stashed in the local variable `parent-accumulator` as to not interfere with the parent's partially computed aggregation (line 581). Simplify is recursively called on the body **R**-expr (line 584). When an inner aggregator has been completely rewritten by Simplify, the resulting contribution is added to the global accumulator variable (line 603). The inner aggregator will rewrite itself as `0` (line 606). Here, rewriting as `0` indicates that there are no delayed **R**-exprs, and that the result of the aggregation has been handled. This is conceptually similar to when the unify **R**-expr rewrites as 1 once the assignment to a variable is tracked via the context (figure 11-7).[189]

aggregation is stored. The inner aggregator is only responsible for handling a single rule body. When the inner aggregator is *completely* rewritten its body and added its contribution to the outer aggregator's accumulator, it will be rewritten as `0`. Rewriting as `0` indicates that no delayed **R**-exprs still need to be evaluated.[189] A conceptual example of the rewrite rules that handle the inner and outer aggregator is shown in figure 11-20.

In the next section, I will define iterators, which is how the "Domain(`X`, `b(X)`)" calls in figure 11-16 are implemented.

11.7 Iterators

In chapter §9, I defined the optional constraint **R**-expr, which lifts useful conjunctions out of nested **R**-exprs. However, recall that optional constraints are only used to *license* various rewrites that are useful for rearranging the **R**-expr into something that can be rewritten. Iterators, as described in this section, are an implementation of optional constraints that work with variables that have a known finite domain. For example, iterators correspond with optional constraints of the form `opt((X=1)+(X=2)+···+(X=1000))`, where this is a disjunction that represents 1000 different assignments to the variable `X`. Iterators are *more limited* than the optional constraints from chapter §9. However, by limiting iterators to *only* focus

[189] The outer aggregator will have a *disjunction* of multiple inner aggregators (which represent each rule). The `0` is the identity element of disjunctions. This is conceptually similar to the `unify` **R**-expr rewriting as 1 when its value has been saved in the context (figure 11-7). In that case, 1 is the identity element of the *conjunction* that contains the `unify` **R**-expr.

on ground assignments to variables, we can build efficient class abstractions and implementations. For example, an iterator can be backed by the efficient disjunctive trie (section §11.6.1.2), in which case it will stream the values assigned to a variable, as known by the trie. This is akin to having an iterator over the keys in a hash-table.

In addition to iterators having efficient implementation, iterators allow us to generate JIT-compiled code that is conceptually similar to the example code in figure 11-16. This is tractable as iterators and assignments to variables can be held in local variables of generated code at runtime, in the same way that is done when writing a procedural program. Compilation of **R**-expr and will be discussed extensively in chapter §12.

11.7.1 Iterator Interface

Every **R**-expr defines its own GETITERABLES method. The GETITERABLES returns a set of `IterablesWithMetadata` that are supported by the **R**-expr (figure 11-21). Calling GETITERABLES on an **R**-expr `R` is equivalent to rewriting `R` as `R*opt(···)*opt(···)`, using the rewrites defined in chapter §9, where `opt(···)` will represent a single iterator. The GETITERABLES method's definition is informed by the **R**-expr's behavior with respect to rewriting optional constraints. For example, the GETITERABLES on a conjunction simply returns the union of all iterables. This follows as `(R*opt(Ropt))*(S*opt(Sopt))`$\rightarrow$`(R*S)*opt(Ropt)*opt(Sopt)` is a valid rewrite which simply rearrange optional constraints inside of the conjunction.

```
607 interface IterableWithMetadata {
608  List<Variables> VariableOrder();          // order variables are assigned
609  Set<Variables> VariablesCanEnumerate();  // variables with finite domains
610  Iterator iterator();                                      // The iterator
611 }
612 interface Iterator {
613  LazySeq<IterInstance> Run();                      // Run without duplication
614  LazySeq<IterInstance> RunNotFinite();                   // Allow duplication
615  Iterator BindValue(Object value);  // directly assign value, no iteration
616  int estimateCardinality();       // estimate of number of values returned
617 }
618 interface IterInstance {
619  Object value();                  // Value assigned to the iterated variable
620  Iterator continuation();                       // Iterator for next variable
621 }
622 interface LazySeq<IterInstance> {        // LazySeq is provided by Clojure
623  IterInstance head();                             // Return the current value
624  LazySeq<IterInstance> tail();    // Return tail, which contains next value
625 }
```

Figure 11-21. Java Interfaces for iterables and iterators.

The iterator interface is designed to be roughly comparable to iterating over tries,[190] (figure 11-21). The `IterableWithMetadata` interface provides information about which variables the iterable supports. When the iterable is started, using the `iterator` method on line 610, the `Iterator` returned *only* iterates over the domain of a *single* variable (line 612). This corresponds to iterating over all of the value bindings to a variable at the first level of the trie. If we use the trie from figure 11-13b as an example, the first iterator will iterate over the bindings for the variable `X`, which are $\{?, 1, 3\}$. To obtain an iterator for a second variable, the `continuation`

[190]Such as the trie in section §11.6.1.2 figure 11-13.

method on line 620 is used. The second iterator is *conditioned* on the current value returned by the first iterator. To continue using the trie from figure 11-13b as an example, when the first iterator has `X=3`, the continuation iterator over `Y` will loop over the values $\{4, 9\}$, which correspond to the set of possible values that `Y` can be assigned condition on `(X=3)`.

The order in which iterables loop over variables is indicated by the `VariableOrder` method on line 608. The variable order typically corresponds to the order in which variables appear on the underlying data structure (which is usually a trie). In the case that the variable order supported by an `IterableWithMetadata` is not the desired variable order, temporary materialized tables can be created to rearrange iteration order of variables.

When using the iterator, we ideally want to only iterate over variables that have a known *finite* domain. Recall that this corresponds to an optional constraint of the form `opt((X=1)+(X=2)+(X=3))`. The `VariablesCanEnumerate` method (line 609) returns the set of variables that have known finite domains. When a variable does not have a finite domain, the corresponding optional constraint has the form `opt((X=1)+(X=2)+1)`, which is not useful for splitting an **R**-expr into smaller *non-overlapping* **R**-exprs (previously discussed in section §9.2.2). When a variable does not have a finite domain, the iterator interface still *allows* the variable to be iterated. However, special care is required to handle the "`FREE VALUE`", which will be returned (indicating that any value is allowed to be assigned to the variable). For the purposes of implementation, and in an attempt to limit potential bugs, the Run method on line 613 *cannot* be used when a variable does not have a finite domain. Instead, a second method `RunNotFinite` on line 614 is used instead. Furthermore, the Run method *guarantees* that a value will not be repeated, whereas no guarantee is provided by the `RunNotFinite` method.

This design of iterating over a single variable at a time was chosen so that the system can interleave rewriting with SIMPLIFY with iteration. This was hinted at in figure 11-16, where some simplification on line 8 was performed *before* the variable `Z` was iterated.

Finally, an iterator can also be used as a *filters*. This is done using the `BindValue` method on line 615. The `BindValue` function allows us to avoid iterating through the domain of a variable at a particular level of the trie and directly go to the next variable. For example, if we have the **R**-expr `(X=3)*R`$_{trie}$, we do not need to use `R`$_{trie}$ to iterate over values of `X`. Furthermore, if `R`$_{trie}$ does not contain a branch for `(X=3)` (meaning `(X=3)*R`$_{trie}$$\to$`0`), then `BindValue` will return `null`, which can be used for filtering impossible values. Calling `BindValue` is generally more *light weight* than rewriting the **R**-expr with SIMPLIFY. Hence, when we have a conjunction of multiple iterators, the `BindValue` function can be used to filter out impossible values before rewriting is even attempted.

11.7.2 Different Kinds of Iterators

Inside of the Dyna implementation, there are many different classes that implement iterators (figure 11-21 only shows *interface*). Some of these iterators come from built-ins. For example, the integer range built-in's iterator enumerates integers. In this case, the iterator has its own class implementation, which efficiently represents the current integer as well as the min/max integer it is iterating to.

Outside of the built-ins, some core **R**-expr kinds also interact with the iterators. For example, projection and aggregation will project out variables from an iterator when the variable is projected out in the **R**-expr representation. In other words, if we have an **R**-expr like in figure 11-22:

```
proj(X, (X=1)*((Y=1)+(Y=2)+(Y=3))+
        (X=2)*((Y=7)+(Y=2))+
        (X=3)*((Y=9)+(Y=1)))
```

Figure 11-22. Example **R**-expr with projection of `X`

When the system calls getIterables on the above **R**-expr in figure 11-22, it will only get an iterable over the variable `Y`. This corresponds to the optional constraint `opt((Y=1)+(Y=2)+(Y=3)+ (Y=7)+(Y=9))` pulled through the projection to the top of the **R**-expr. This is handled by the implementation of getIterables for projection and has wrapping iterable, and iterator classes that modify the iterables returned from the body of the projection. The wrapping iterator will attempt to stream the iterated values of `Y`, instead of materializing the values of `Y` into a data structure.

11.7.3 Iterators *Attempt* to Stream Values

The intention of iterators is to be able to make use of the underlying data structures without having to materialize everything as an **R**-expr. Much like streaming APIs and libraries, the iterator module has a number of different kinds of iterator types that wrap an iterator and modify it as needed. As already discussed for figure 11-22, the Dyna implementation has an iterator that will project out a variable from a wrapped iterator. The wrapper modifies the return values from the `VariableOrder` and `VariablesCanEnumerate` method from the `IterableWithMetadata` (figure 11-21) such that the projected out variable is not visible outside of the projection. In the case that an iterator's variable order is "unfavorable" to the projection (such as the variable order is `[X,Y,Z]` and we are projecting out the variable `X`), then the wrapper will consolidate the values of `Y,Z` using a temporary data structure *when* the `Run` method is invoked (line 613). Alternately, if the `RunNotFinite` method is used, then no consolidation is required (line 614).

Conjunction of iterators

In some cases, we might want to combine two or more iterators on the same variable. For example, with a conjunction of two iterators from different sources, we might use one of the iterators as a filter as in figure 11-23. Checking that an iterator can bind to a particular value can be much faster than using SIMPLIFY to rewrite the **R**-expr.

```
1: for ⟨$x$, continuation_1⟩ ∈ iterator_1.Run() :
2:     if iterator_2.BindValue($x$) ≠ null :
3:         yield value $x$
```

Figure 11-23. Merging conjunctive iterators using `iterator_2` to filter out values which will not work (would cause the **R**-expr to rewrite as `0`).

We can also use a conjunction of iterators in the case that we do not have a single iterator that can iterate over all of the variables that we are interested in. In this case, a second iterator can be used as in figure 11-24.

```
1: for ⟨$x$,continuation_1⟩ ∈ iterator_1.Run() :
2:     if continuation_2 = iterator_2.BindValue(x) :
3:         for ⟨$y$, _⟩ ∈ continuation_1.Run() :
4:             for ⟨$z$, _⟩ ∈ continuation_2.Run() :
5:                 yield values $x$ & $y$ & $z$
```

Figure 11-24. If we have `iterator_1` over `X` & `Y`, and `iterator_2` over `Y` & `Z`, then we can iterate over the variables `X`, `Y` & `Z` by combining these two iterators.

Disjunction of iterators

To handle a disjunction like

$$\Big(\texttt{(X=1)+(X=2)+(X=3)}\Big)\texttt{+}\Big(\texttt{(X=1)+(X=7)+(X=11)}\Big)$$

we can have iterators from each branch of the disjunction. These iterators can be merged by first iterating through the domain of the first iterator and then iterating through the domain of the second iterator, using the first iterator to filter out values already seen. Assuming that the iterator efficiently implements the BindValue method, then this will be more efficient than creating an intermediate data structure for the values of X. This is shown in figure 11-25.

```
1: for X ∈ iterator_1.Run() :
2:     yield value X
3: for X ∈ iterator_2.Run() :
4:     if not iterator_1.BindValue(X) :          ▷ Filter out values from iterator_1
5:         yield value X
```

Figure 11-25. Iterating through a disjunction of iterator_1 and iterator_2. The first iterator is used to filter out values of X that have already been seen on line 4.

Note that the filtering in figure 11-25 corresponds to the more generally applicable if-expression rewrite rule 72. In this case, given the constraint if(Q1+Q2,1,0) (which can come from an optional constraint), is rewritten as if(Q1+Q2,1,0)→if(Q1,1,0) + if(Q2,1,0)*if(Q1,0,1), which shows that we can split the disjunctive constraint (conditional of the if-expression) if we use if(Q1,0,1) to filter out redundant values.

11.7.4 Using Iterators

The iterator subsystem provides a run-iterators macro as an internal API that automatically determines how to use the available iterables to subdivide the **R**-expr. The run-iterators macro takes a list of variables that the system wants grounded (such as the list of projected variables in the case of aggregation), as well as a set of iterables returned by the GETITERABLES method. The run-iterators macro automatically chooses which iterators to use. `Run-iterators` can either select a single iterator or combine multiple iterators in the case where no single iterator will work. When there are multiple iterators that can iterate the same variable,[191] the `run-iterators` macro makes use of the `estimateCardinality` method (line 616) to select the iterator that reports the smallest estimated cardinality. When there does not exist an iterables that guarantees that some requested variable is grounded, then `run-iterators` can attempt a *best effort* iteration binding other variables. For example, if `run-iterators` is requested to ground the variables `X`, `Y`, and `Z`, where there only exists an iterator over `Y` and `Z`, the system is allowed to iterate over `Y` and `Z` in hopes there is a constraint like `plus(X,Y,Z)` contained in the **R**-expr that is capable of determining the value of `X` from an assignment to `Y` and `Z`.

11.7.5 Efficient Aggregation uses Iterators

In figure 11-16, the aggregator makes use of a function of the form Domain(`X`, `b(X)`) to loop over the domain of a variable. This comes from the iterator subsystem and is the `run-iterators` macro in the implementation. When `run-iterators` chooses the order in which variables will be bound, it is informed by the iterators available. (The iterator's variable order is informed by the order in the data structure's store data, such as in the trie.)

[191]For example, if we have `((X=1)+(X=2))*((X=1)+(X=2)+(X=3)+(X=4))`, then the first iterator iterates over $\{1,2\}$ and reports an estimated carnality of 2, and the second iterator iterates over $\{1,2,3,4\}$ and reports an estimated carnality of 4.

When a variable `X`'s value is assigned, the aggregator will rewrite the **R**-expr using the information (`X`=x). If the **R**-expr is rewritten as `0`, then this means that the current assignments of variables are inconsistent with the **R**-expr, and the aggregator will skip the value x and try the next possible assignment to `X` without having run more deeply nested loops over variables. Attempting to rewrite the **R**-expr before all variables have been bound is akin to the optimization of lifting operations outside of loops or propagating new bindings to variables once available in constraint logic programming through the constraint store.

The inner aggregator is responsible for performing the aggregation inside of the loop once all variables have been bound and the **R**-expr is completely rewritten. If the **R**-expr body of the aggregator is not rewritten completely, then the partially rewritten **R**-expr is preserved by the aggregator in the hope of being rewritten later.

Chapter 12

Compilation of R-expr Rewrite Strategies

One unfortunate thing about research sometimes is that despite making significant progress, we can still find ourselves with an unsatisfactory system. In this case, the realistic implementation described in chapter §11 is easily 100 to 1000 times faster[192] than the pure, minimal implementation presented in chapter §8. However, despite that, the "*realistic implementation*" is still *too slow* for the use cases that we care about.[193]

In hindsight, this is not terribly surprising. We can think of the **R**-expr as

[192]This observation was made on Python prototypes of term-rewriting. The code at `https://github.com/argolab/dyna-R/blob/backend-v2/dyna_match_paper/rexprs.py` attempts to be a pure term rewriting-based system that closely matches chapter §8. `https://github.com/argolab/dyna-R` was a prototype of the **R**-expr based system described in chapter §11, although it lacks several features added in the implementation for this dissertation: `https://github.com/argolab/dyna3`.

[193]I note that the efficient disjunctions are asymptotically inefficient in some query modes. This asymptotic inefficiency was accounted for when making the judgment call to prioritize the work in this chapter over adding indices to the tries.

akin to an internal representation inside a compiler. Our rewrites are, therefore, akin to optimization passes. For example, a rewrite like 28, `plus(1,2,X)`$\to$`(X=3)` is essentially constant propagation. Unsurprisingly, an optimizing compiler being used to evaluate the entire program is *much* slower than the equivalent procedural instructions.

As such, in this chapter, I will discuss my efforts to compile **R**-expr rewriting. When rewriting **R**-exprs that come from Dyna programs, there will be many similar **R**-expr "shapes" and sequences of rewrites performed. Hence, it is reasonable to believe that we can speed up the Dyna implementation by compiling **R**-expr rewrites.

The goal of our implementation is to maintain the flexibility of **R**-exprs while running faster. The work presented here focuses on removing the overhead of the rewrite system and is not a "*final stage*" compiler. Instead, our goal is to generate procedural code instead of individual rewrites that should be comparable to what a user would write in a procedural language.[194] Before we start looking at *how* we generate code, let us briefly discuss what some of the *overheads* are that we are aiming to eliminate.

[194]Our compilation targets Clojure, which is compiled into Java bytecode and eventually machine code by the Java virtual machine (JVM).

```
a = b * c
```

(a) Python

```
LOAD_GLOBAL    0 (b)
LOAD_GLOBAL    2 (c)
BINARY_OP      5 (*)
STORE_FAST     0 (a)
```

(b) Python Bytecode

Figure 12-1. Example Python with procedural instructions

12.1 What is Overhead?

Let us start by defining what "overhead" means in the context of **R**-exprs and executing Dyna programs. Consider the line of code in a procedural programming language in figure 12-1a. To evaluate figure 12-1, we are reading from two variables and calling the multiplication operation. If this was converted into efficient assembly instructions, we might have two read-from-memory-into-register instructions, followed by a multiplication instruction and then a final store instruction. These four instructions in figure 12-1b represent the *required* operations to evaluate this line of code.

Now, suppose that all variables are stored in a hash-map. Instead of performing a single read to load the variable, the system indirects through multiple operations performed by the hash-map: reading the base pointer of the hash-map, computing the hash code of the string "b", reading the size of the hash-map, computing the `hash("b")` mod the size of the hash-map, reading an entry in an array, checking equality between the string "b" and the entry found in the array. In this case, every operation the hash-map performs can be considered overhead. Now, *overhead* does not mean wasted. In this case, using a hash-map makes Python's implementation simpler. However, it does not contribute to the "useful work" of multiplying numbers.

12.1.1 Overhead with **R**-exprs Rewriting

When running a Dyna program represented as **R**-exprs, there are similarly some operations that correspond to manipulations of the data that we are interested in and other operations that correspond to the overhead of **R**-exprs. For example, some overheads we have with **R**-exprs are:

1. Manipulations and accesses of the context $\mathcal{C}$. The context contains both variable bindings and other conjunctive **R**-exprs. The assignments to variables are stored in a hash-map.[195] This means that matching the `:free` and `:ground` preconditions requires accessing the context's hash-map. Similarly, when the context is used to find conjunctive **R**-exprs, those **R**-exprs are stored in sets. Finding the right conjunctive **R**-expr requires searching through any relevant set. Additionally, conjunctive **R**-exprs are added to these sets when encountered.

[195] Variable identifiers are generic objects. Converting all variables to small integers that could be used as array indices is non-trivial because **R**-expr objects are immutable and reused in multiple places.

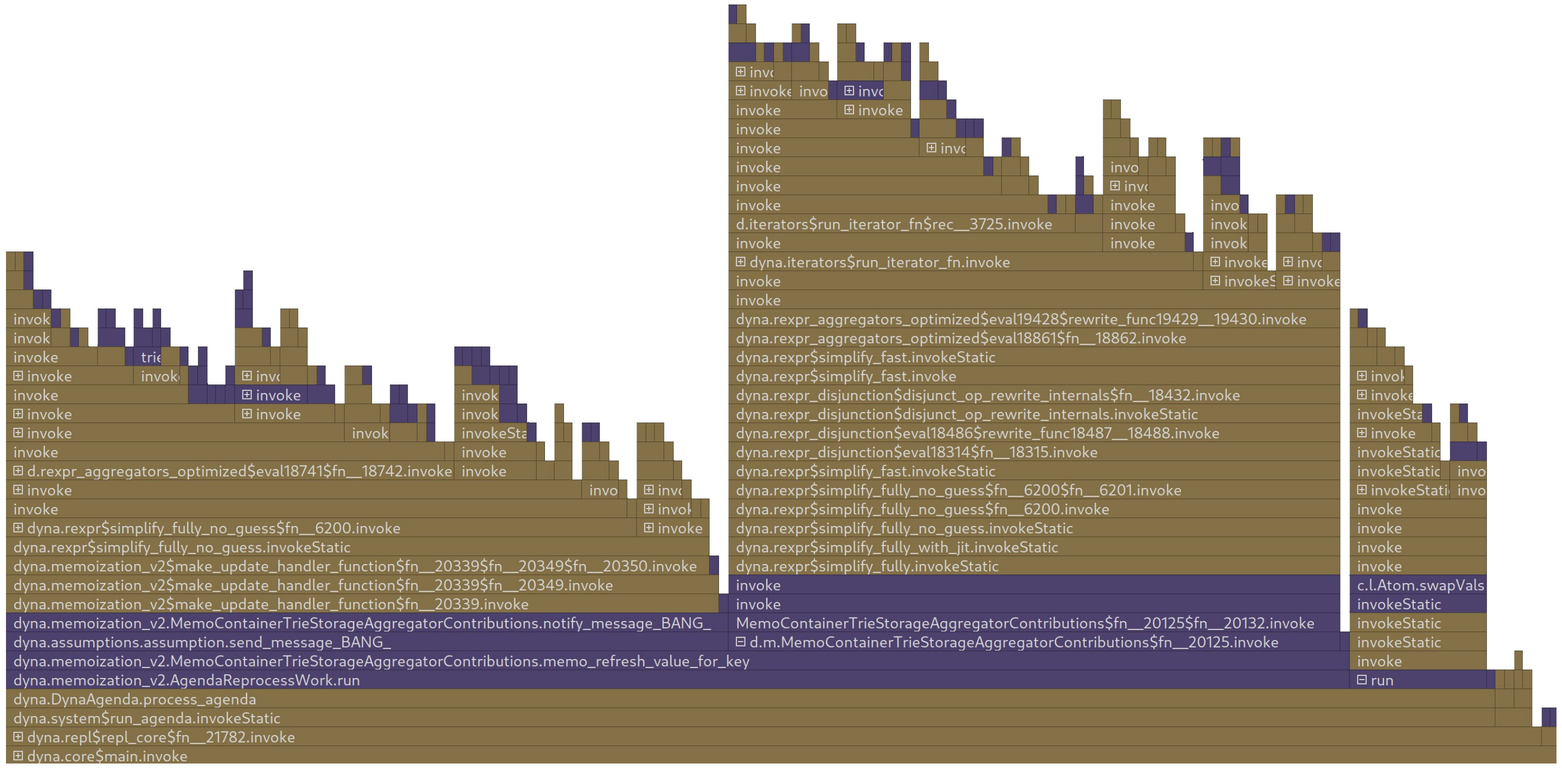


Figure 12-2. This is a flame graph for CKY parsing using memoization and forward chaining of a small sentence without using the compiler. From this graph, we can observe which functions take the most time to run. We can observe two operations used by memoization (chapter §10): sending messages for updates and recomputation of values. Both of these operations use Simplify to rewrite specially crafted **R**-exprs to perform the bulk of the work. We can observe that the **R**-expr rewriting occupies 98% of the runtime.

2. To identify a rewrite, the system checks a list of potential rewrites.[196] Rewrites are segmented by the *kind* of **R**-expr, but operations such as checking the ground/free state of variables use the context. Conjunctive **R**-exprs require scanning through the context.

3. The results of rewrites are either **R**-exprs or assignments to variables that are tracked by modifying the context (such as in figure 11-6). In both cases, this means creating many new Java objects.

4. Variable renaming and other similar manipulations of **R**-exprs require scanning the entire **R**-expr to perform the necessary rewrites. A new **R**-expr is created and returned.

5. Finding iterators is a recursive function that returns a set of iterators. This creates many intermediate sets and iterator types. This happens every time we evaluate any disjunction under an aggregation.

6. Memoization and update messages are handled via a priority queue.[197] Therefore, the push and pop operations take $O(\log n)$ time where n is the number of updates messages pending. Comparably, a dynamic program with a fixed execution order does not require an agenda at all and entirely avoids this

[196] Recall that the pseudocode in section §8.A first matched against the type of the **R**-expr, and then checked each individual rewrite to see if any could apply.

[197] `$priority` returns a floating point value that is used to sort update messages (section §10.8.5).

overhead.[198]

This is not intended as an exhaustive list of the sources of overhead. Furthermore, not all cases of overhead are “bad” or something *we* need to deal with. For example, if we allocate and deallocate a Java object within the same compilation unit, then the Java compiler can eliminate the allocation altogether. Unfortunately, given the design of SIMPLIFY from chapter §11, this optimization is unlikely to happen, as compilation units are likely only able to get as large as a single rewrite rule.

12.2 Compilation Overview

In the remainder of this chapter, I will detail how we compile **R**-exprs and their rewrites.

Our goal in compiling is to make the rewriting **R**-exprs run faster. To make compilation work, we need *compilable units* and *reentrancy* so that the compiled units are reused. The most obvious approach would be to compile the SIMPLIFY function, which is similar to a bytecode interpreter where **R**-exprs are the “bytecode”. However, this has the same problems we had with memoization (section §10.2). It requires matching against a large part of the **R**-expr to figure out when there is reentrancy. Instead, we will take an approach similar to the one taken with memoization. We will replace a sub-**R**-expr with a *JIT-generated* **R**-expr kind that

[198]Admittedly, this is not addressed by JIT compilation in this dissertation, though future work should consider looking into this problem. Section §16.6

represents a compiled unit, such as in figure 12-3.

Generated **R**-exprs and generated **R**-expr rewrites will be generated using a *just-in-time* compiler (JIT) and are made up of multiple *primitive* **R**-exprs and **R**-expr rewrites (the **R**-exprs and rewrites defined previously in chapters 5 and 6).

As we will see, our approach can be broken down into these two major components, which work in tandem:

1. Generating new JIT-generated **R**-expr kinds, which represent larger **R**-expr expressions *state*. These JIT-generated **R**-exprs correspond to compilable units of code. This allows us to match with a larger **R**-expr unit all at once. Therefore, we can save time by not having to match each **R**-expr individually. When a particular **R**-expr *state* is reencountered, we will reuse the generated **R**-expr kind, enabling reentrancy (section §12.4.2).

2. Generating rewrites for the JIT-generated **R**-expr kinds. The **R**-expr kind on its own does not represent execution but rather a state of the program. Hence, we need to generate rewrites for the new **R**-expr kinds. These rewrites correspond with multiple *primitive rewrites* being performed against the underlying **R**-expr state. Performing a single match allows us to avoid the overhead of performing each match individually and serializing the state as an **R**-expr between rewrites.

12.3 Generating New JITted **R**-expr Kinds

The mechanism for generating a new **R**-expr kind is conceptually straightforward. Every JIT-generated **R**-expr kind is given a name like `state1234`,[199] and corresponds to some *primitive* **R**-expr state, which is tracked via external metadata. Primitive **R**-expr kinds are those previously defined in chapter §5. The generated **R**-expr will have **holes** corresponding to variables and **R**-exprs not statically knowable. For example, exposed variables that are referenced by other **R**-exprs may have different names in different contexts. Therefore, the generated **R**-expr will have a hole representing a variable. Variables that are only used internally and projected out do not require a hole in the JITted **R**-expr because they only represent internal data. For example, the variable `X` in `proj(X, times(X,Y,Z)*⋯)` is only used locally, but the variables `Y` and `Z` are in the externally visible vars(·) of this expression, therefore, will be represented with holes.

Holes are typed as we saw in section §11.5.1, and can include sub-**R**-exprs, value types, and hidden variables.[200] This is very important as it allows a generated **R**-expr to represent *part* of a larger **R**-expr and does not require that the *entire* **R**-expr is represented at the same time. In other words, if we think of an **R**-expr as a recursive tree data structure, then the generated **R**-expr is allowed to represent

[199] Names are generated randomly using the gensym method in Clojure, which generates a unique name using a global counter.

[200] Hidden variables are how variables on projection are annotated, section §11.5.1.

a sub-tree all the way down to the leaves, and it is also allowed to represent a sub-tree that stops before it gets all the way to a leaf. This allows us to 1) create smaller JIT-generated units that are more likely to be reusable and 2) we can exclude **R**-exprs that we do not want to support in the JIT (I will revisit this point in section §12.5).

An example of creating a JIT-generated **R**-expr is shown in figure 12-3.

12.4 Generating New Rewrites

The second part of JIT-compiling an **R**-expr is to generate the corresponding rewrites. The high-level approach is inspired by tracing JIT compilers ([31, 74, 75], section §3.6). A tracing JIT works by compiling a *trace* of operations that are performed rather than generating using the definition of a particular method. In the case of rewriting, a trace will correspond to a sequence of one or more rewrites applied to the state represented by the JIT-generated **R**-expr.

To identify which rewrites can be applied, the JIT uses the same information that SIMPLIFY uses to match a rewrite. This includes the context $\mathcal{C}$, which contains information about conjunctive **R**-exprs and the values assigned to the variables. To gain access to the context, we invoke the JIT-compiler from SIMPLIFY. In other words, when SIMPLIFY encounters a JIT **R**-expr kind that does not have a generated

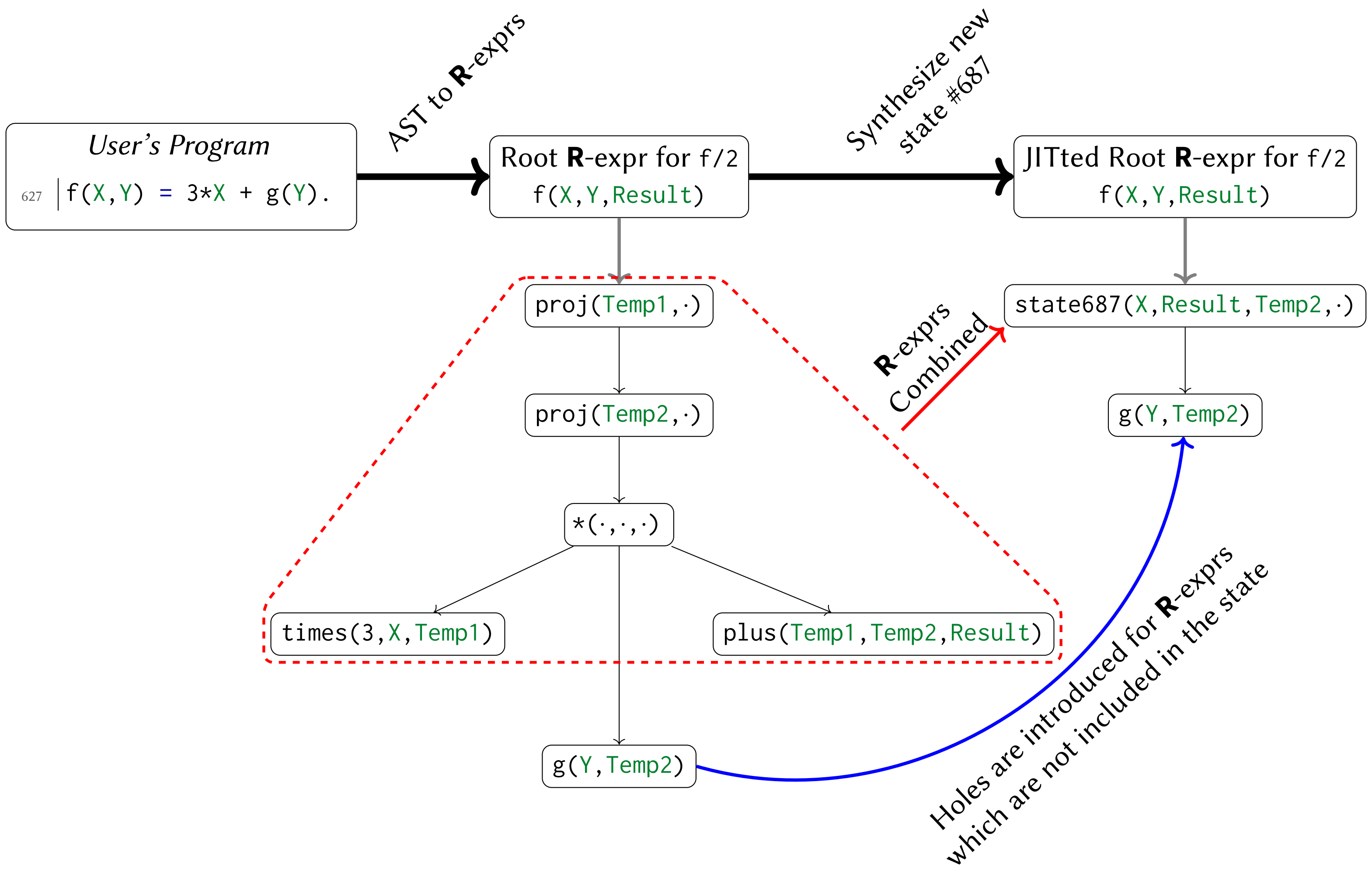


Figure 12-3. This figure shows the first step in JIT compiling an **R**-expr. The center-dot (·) represents a pointer to another **R**-expr. We identify a subset of an **R**-expr (represented by the red dashed box in the middle) that we want to compile into a single state. The selected **R**-expr does not have to include all children **R**-exprs, as represented by '`g(Y,Temp2)`', which is not included in `state687`. This allows us to avoid including parts of the **R**-expr that are unlikely to benefit from being compiled into the state. Variables such as `X`,`Result` are referenced by the external environment, and the variable `Temp2` is hidden and referenced by the **R**-expr in the hole. Variables that are only internal, such as `Temp1`, do not have variable slots to track their name.

rewrite,[201] it invokes the JIT compiler with the **R**-expr and the current context. The process of creating JIT-generated rewrites therefore happens *lazily*—as is typically done when JIT compiling a program.

Any rewrite generated by the JIT compiler is immediately applied to the **R**-expr, in the usual way that SIMPLIFY applies rewrites to **R**-exprs. However, the newly generated rewrite will also be added to the collection of available rewrites that the system has. This means that the same rewrite can be applied in the future without requiring additional code generation. An example of this is shown in figure 12-4 with a rewrite being generated for the **R**-expr from figure 12-3 and applied immediately.

12.4.1 Combing Multiple Rewrites Into One

One of the core features of the JITted rewrites is that we can combine multiple steps of rewriting together into a single JIT-generated rewrite. This limits the overhead of matching and allows us to pass values between primitive rewrites using local variables.

The way the JIT compiler generates rewrites differs from what we have seen with the primitive rewrites from chapter §11. First, with the primitive rewrites, there

[201]This means that there are *zero* rewrites for a given **R**-expr kind. Not that there is no rewrite that can be applied. If there is a rewrite whose preconditions are not satisfied, then this will not invoke the JIT compiler. To allow JIT **R**-expr kinds to have more than one rewrite, the JIT compiler is still invoked with a *low priority* if there are no other rewrites that can be found to rewrite the top level **R**-expr.

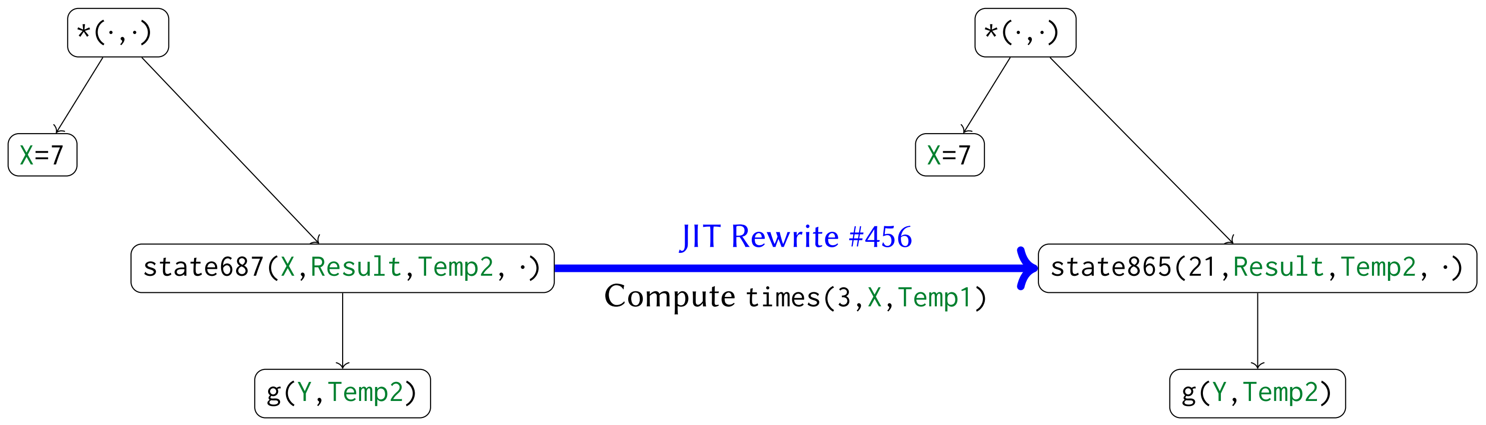


Figure 12-4. A new JIT rewrite is generated for `state687(···)` from figure 12-3. The JIT is invoked by the SIMPLIFY method, which recursively walks the **R**-expr tree. When the JIT is invoked by SIMPLIFY, it has access to the current **R**-expr `state687(···)` and the context $\mathcal{C}$. Hence, the JIT can see that `X` is assigned the value `(X=7)`. The JIT looks for rewrites that can be applied to the "primitive **R**-expr state" represented by `state687(···)`. In this case, the only rewrite that can be applied is to rewrite `times(3,X,Temp1)`.

Observe that the result of this rewrite, `state865(···)`, no longer has a reference to the variable `X`. The reason is that there are no **R**-exprs in `state865(···)` that still refer to the variable `X`. Furthermore, we previously had that `Temp1` was a hidden variable (due to the projection), but now that it has a known value that is not statically knowable, that value needs to be saved somewhere. Therefore, `state865(···)` has an argument (hole) for the value assigned to `Temp1`, in this case that value is 21.

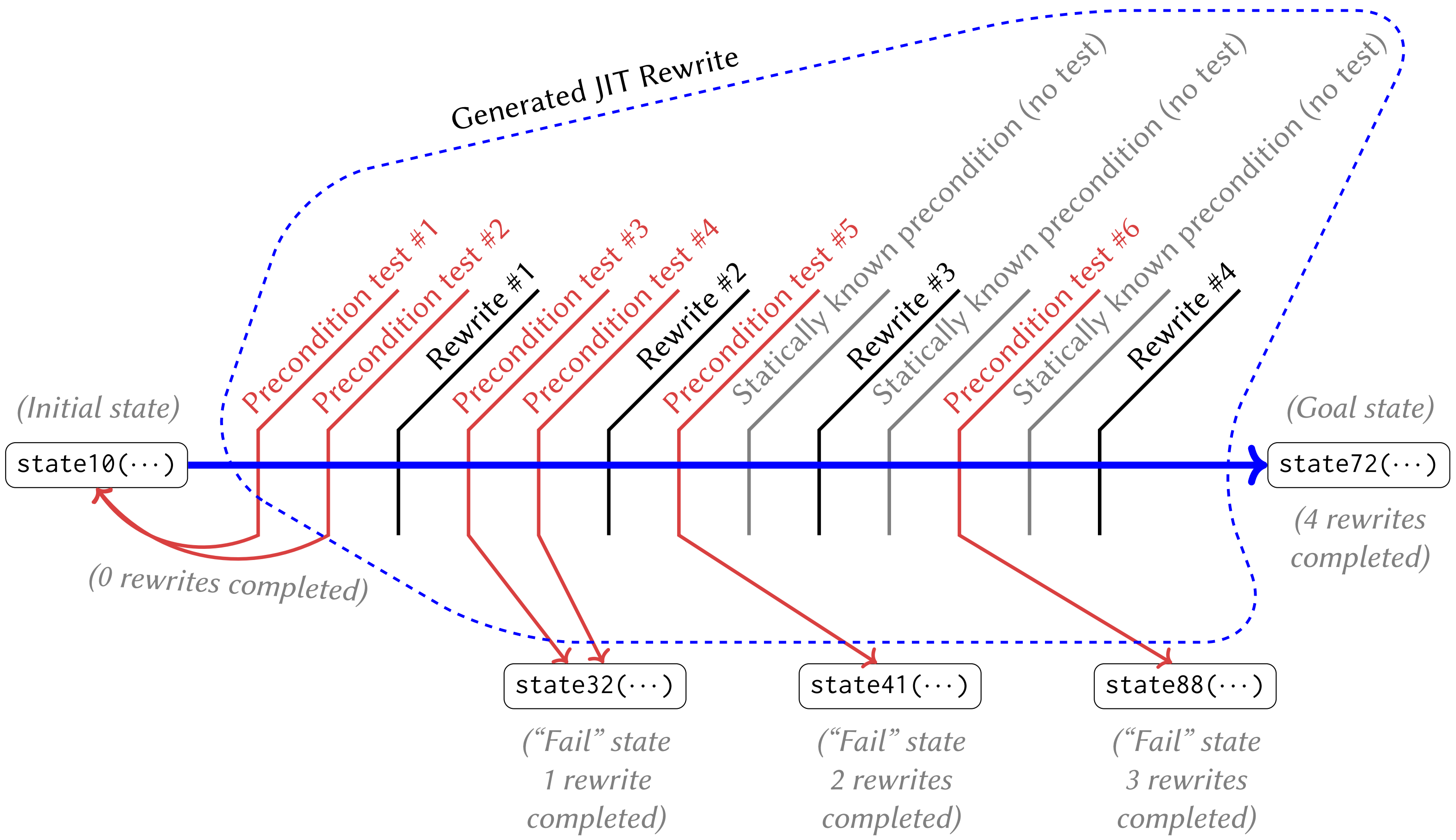


Figure 12-5. This represents a single rewrite. The blue arrow in the middle is the *trace* of 4 different rewrites. The trace will include all possible rewrites from the time of compilation, starting from the initial state. The preconditions or match statements for a rewrite are represented as red lines and are checked before the rewrite is applied. If a precondition fails, then it takes branches to a generated **R**-expr state. JIT-generated **R**-expr states such as states 32, 41, 88, and 72 are generated on demand at the same time that the rewrite was generated. If an existing state matches, then it will be reused rather than generating a new state. Preconditions that can be statically resolved to be true, either because they were already checked by another rewrite or because they consume statically known output from an earlier rewrite, do not generate any code in the generated rewrite (represented in gray).

is a single `:match` statement (section §11.5.3). The match statement is matched *in full* before the rewrite is applied. If we were to apply this same design to the JIT-generated rewrites, then we would end up with rewrite rules that are not widely applicable, as it would require matching all of the preconditions before anything can be rewritten. For example, suppose that we combine a sequence of 20 rewrite rules together. If we have to match against all 20 different conditions first, then it is likely that one of the rewrite's preconditions will not match, and the JIT-generated rewrite will get poor reuse.

To get around this issue, we *alternate* between checking preconditions and performing rewrites. A rewrite's preconditions are checked right before the rewrite is applied. If one of the rewrite's preconditions has been previously checked or can be statically determined, then it is not rechecked, and no code is generated for the precondition check, as is shown in figure 12-5.

12.4.1.1 What Happens If Only *Some* Rewrites Match?

Interlacing the checking of rewrite preconditions and applying rewrites means that we can end up in a scenario where only some of the rewrites have been applied. In this case, we need to generate code to handle the *failed* match. To accomplish this, we take inspiration from tracing. In tracing, when there is a conditional branch, a check is inserted to check that the branch goes the same way as when the JITted code was originally generated. If the conditional test branches the other way, the

JITted code falls back to the new code and only then will generate the code under that branch [31, 74, 75]. An example of this was shown in the related work chapter with a small procedural program in section §3.6.

Adapting this idea to **R**-exprs, we can observe that when a precondition fails to match, we need to *represent* the state of the primitive **R**-expr (that is being "conceptually" rewritten) at this point. Furthermore, we want to avoid generating all possible rewrites of what *can* happen upfront (e.g. generating a sequence of 20 rewrites without knowing that it can be used at least once is unproductive). First, to represent the state, we can use the generate-a-new-**R**-expr mechanism from section §12.3. An example of a single rewrite that interlaces checks and rewriting is shown in figure 12-5.

To complete the analogy to tracing, we need to generate code for branches that were previously not taken. Recall from section §12.4, I mentioned that when a JIT-generated **R**-expr is Simplified for the first time, we will attempt to generate a new rewrite. This new rewrite corresponds to generating the code for a branch that was not previously encountered when running. The end result of generating new **R**-expr kinds and generating rewrites for those **R**-exprs is a "flow graph" that intermixes **R**-expr states as nodes with rewrites as transitions. This is shown in figure 12-6.

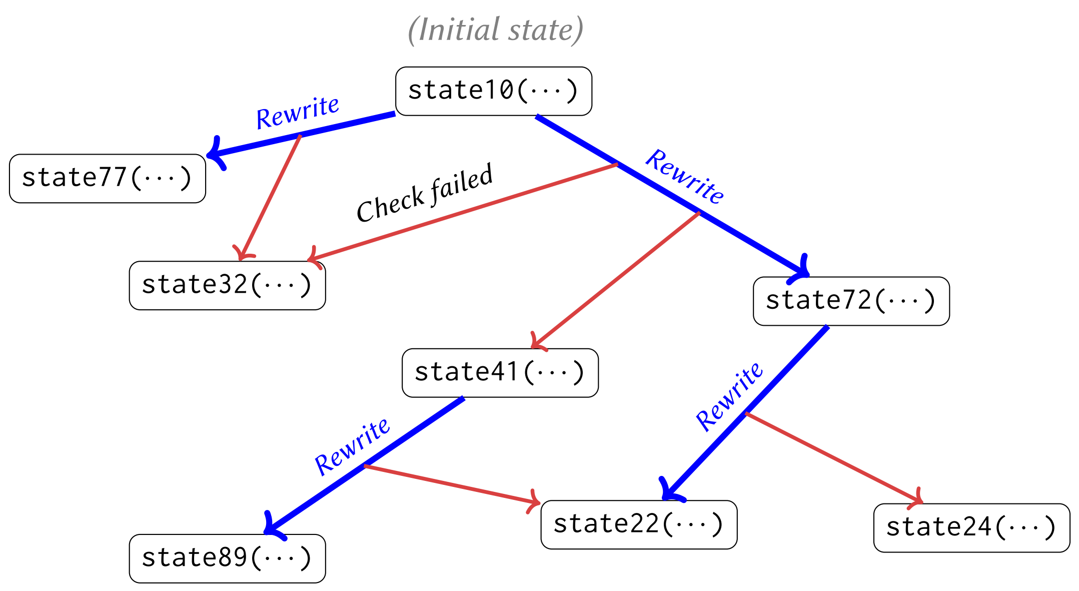


Figure 12-6. An example graph of JIT states and rewrites which go between the states. Rewrites are shown in blue and always start at an **R**-expr state. A rewrite can go to different **R**-expr states depending on how many primitive rewrites match. If a precondition fails, then it takes one of the red edges and immediately jumps to a JIT-generated **R**-expr kind. These failure edges always start from a rewrite and go directly to a state that represents the **R**-expr at this point. The JIT-generated **R**-expr states are reentrant (e.g. states 32 and 22) in the case the same state is encountered multiple times.

12.4.2 How to Reenter JITted R-exprs

The generated **R**-expr kinds represent the state of the program when we are rewriting. Hence, reusing a JIT state is how we get reentrancy into compiled code (as in figure 12-6).

To implement reuse of JIT-generated **R**-expr kinds, whenever we are tasked with generating a new JIT-generated **R**-expr kind, we first check through the previously generated **R**-expr kinds and see if there is anything applicable.

To check for equivalent JIT states, there are a few things that we need to handle. Namely, there are parts of the **R**-expr that we do not care about. First, different variable names can be used in equivalent **R**-exprs. Because we are tracking the variable names using *holes* when a variable is renamed, this just results in a different variable name being placed into the hole. Second, some **R**-exprs can be rearranged. For example, conjunctions are associative and commutative, so `R*S` and `S*R` are *equivalent*.

As a more concrete example, the following two **R**-exprs are equivalent:

```
lessthan(A,B)*lessthan(A,C)*lessthan(C,D)*lessthan(A,E)
lessthan(X4,X0)*lessthan(X2,X4)*lessthan(X2,X1)*lessthan(X2,X7)
```

Figure 12-7. Equivalent **R**-exprs with different variables and ordering[202]

Checking for equivalent states uses a *more relaxed* definition of equality compared to what is used by Simplify from chapters 8 and 11. The first step is to identify a set of potential JIT states by computing a JIT state-specific hash-code. The hash-code ignores the value types (variables names and constants values) in the **R**-expr and the order that sub-**R**-exprs are in associative and commutative **R**-exprs. The hash-code also ignores **R**-exprs kinds that will likely be holes in the JIT-generated state (section §12.5). The implementation of this is done by having a second hashCode method call `jitHashCode` on all **R**-expr kinds. Using the JIT's hash-code, the system will quickly identifies a small set of potential JIT-generated

[202] `A=X2, B=X7, E=X1, C=X4, D=X0.`

R-expr kinds to consider.

Once we have identified a set of possible JIT **R**-expr kinds into which an **R**-expr can be converted, we need to identify if there is a way that an existing generated **R**-expr kind *can* be equivalent to the **R**-expr state that the rewrite is attempting to serialize (during the compilation of a rewrite). This is implemented using a Prolog-style backtracking and unification, which searches through different ways in which the holes on the JIT-generated **R**-expr kind can be filled in such that the JIT-generated state is equivalent to the **R**-expr being converted to a JIT state.

12.4.3 Abstract Evaluation of Primitive Rewrites

So far, I have only talked abstractly about "finding primitive rewrites that can be applied", and "checking preconditions". While this description is the *high-level* idea of the JIT compiler, this level of detail does leave a large glaring hole that needs to be addressed, in how the rewrites are actually generated.

When the JIT compiler is called from SIMPLIFY, it has access to the current JIT-generated **R**-expr state, the corresponding primitive **R**-expr that state represents, context $\mathcal{C}$, and the Clojure source code for all primitive rewrites from chapter §6 that were defined using the `def-rewrite` macro (section §11.5.3). To merge rewrites and speed up the rewrites themselves, we use partial evaluation to convert the Clojure source code that defines a rewrite into a format that can be manipulated inside of the JIT compiler.

Partial evaluation has been well studied in the JIT compilation literature [42, 57, 73, 116, 149]. Partial evaluation is a process in which we evaluate the program before we know *all* the information, which will only happen when it is running. For example, if we have the expression $1+2+x$, we can partially evaluate this expression into $3+x$. The information about 1 and 2 was sufficient to determine the result of $+$. Conceptually, this is very similar to what we have already been doing with **R**-exprs, in that we allow **R**-expr to be *partially rewritten* when there is insufficient information to completely rewrite the **R**-expr. The difference this time is that we are partially evaluating *Clojure code* instead of **R**-exprs.

To implement partial evaluation rewrites defined as Clojure code, we have a `partial-evaluate` function that takes as arguments a Clojure expression represented as an AST (which is a recursive list data structure as is typical of LISP-like languages) and the information that is globally accessible to the rewrite, such as the context $\mathcal{C}$. The Clojure source code that `partial-evaluate` is passed as an argument is from a single primitive rewrite defined using the `def-rewrite` macro.[203] The `partial-evaluate` function returns a data structure that includes any statically knowable values, the current value conditioned on the current context $\mathcal{C}$, and statically known type information. The exact information returned varies by the value's type that the Clojure expression returns, as will be described shortly. Currently, in our JIT compiler, we only distinguish between a few types that are

[203]Recall that last argument to `def-rewrite` is a Clojure expression that when evaluates returns an **R**-expr or value depending on the keyword arguments passed to `def-rewrite`.

useful for compiling **R**-*expr rewrites*. These types are: **R**-exprs, an array of **R**-exprs, value types (variables or constants), an array of value types, “primitive” value (any ground value such as integers or strings contained in $\mathcal{G}$, but also any Java value, some of which may not be contained in $\mathcal{G}$), and function types that are callable. The `partial-evaluate` function does not distinguish the type of a “primitive” value, such as floating point vs integers. This would typically be necessary when doing *low-level* code generation. However, this is not a problem for us, as we are not directly generating a low-level representation but instead generating Clojure, which is sufficiently high-level and does not require this level of detail.

When `partial-evaluate` evaluates an expression that returns either a primitive value type (like an int or string), or a **R**-expr value type (a constant or variable), the `partial-evaluate` function returns the *currently* known value, with respect to the context C that the rewrite is being generated for, and an optional *statically* known value. It also returns a fragment of Clojure code that can be embedded in the generated rewrite to access the value at runtime.

Functions such as ‘`get-value`’ and ‘`is-ground?`’ receive special handling in the JIT-compiler. These functions cache their returned value in a local variable in the generated rewrite. This means that if a rewrite performs `(get-value X)` twice on the same variable, the second time, the generated code will return the local variable instead of calling `(get-value X)` in the generated rewrite. This is useful, as when generating the code for multiple rewrites sequentially, there are often multiple calls to `get-value` for the exact same variable.

The way `partial-evaluate` handles **R**-expr typed variables and functions that return **R**-exprs is a bit more interesting. **R**-expr typed variables are tracked *statically* and have no runtime representation.[204] For example, in figure 12-5, as the rewrite progresses from the initial state on the left to the final state on the right, there are no **R**-exprs objects serialized inside of the rewrite. This means that if we write `(make-lessthan A B (make-constant true))`[205] in the `def-rewrite` for a rewrite rule, the **R**-expr is only tracked statically and there is no corresponding code in the generated rewrite that needs to be executed at runtime. To make this work, we must slightly expand the **R**-expr representations with additional classes. Namely, we define new *value types*[206] to allow general Clojure expressions and variable names that are local to the rewrite in addition to the named variables, which are looked up in the context $\mathcal{C}$, and constants. These new value types are only allowed within a single compilation unit that will generate a single rewrite. Therefore, when the **R**-expr state is serialized into a JIT-generated **R**-expr type, these new value types are saved into a hole for future use. For example, in figure 12-4, the number 21 on `state865(···)` would have been held in a local variable that was referenced by the **R**-expr state. Therefore, the value 21 is saved in the JIT-generated

[204]**R**-exprs that are contained in holes will have a variable which is treated as an opaque pointer to an **R**-expr.

[205]For example, the function `make-lessthan` is used by rewrite rule 34 and in the `def-rewrite` example figure 11-8.

[206]The "value type" is a Java interface that is implemented by both constants and variables. Here, we are simply defining new classes that also implement the value type interface.

R-expr kind so that the remaining **R**-exprs can still refer to the value 21.

This design of not representing **R**-exprs in the compiled code was inspired by the Truffle project [149][207] which is a framework for making JIT compilers by defining an AST interpreter. The AST is represented using standard Java objects. When the AST is compiled, the AST does not have a representation in the generated code outside of the program counter, just like we are doing with the **R**-expr typed variables.

12.4.4 Structure of Generated Rewrites

The way in which the generated code is structured is heavily influenced by the fact that we are generating Clojure code, which is translated into Java bytecode.[209] This means that we cannot generate JUMP instructions in the code. Instead, we use Java exceptions and try/catch blocks to emulate JUMPS. These operations can be converted into JUMP instructions when translated into machine code by the Java Virtual Machine (JVM). A conceptual example of this is shown in figure 12-8 with a rewrite actually generated by the JIT compiler shown in figure 12-9.

[207]The Truffle project is built on and closely integrated with the Graal compiler [56, 57].
[208]Some manual cleanup has been done to make the code presentable.
[209]Even if we generated Java bytecode directly, there are limitations on the JUMP instructions that can be generated, which is enforced by the Java Virtual Machine.

```
try:
    if not CHECK(⋯) :                              ▷ Check precondition 1 for first rewrite
        throw CheckFailure      ▷ throw/catch can be converted to JUMPs by the Java VM
    if not CHECK(⋯) :                              ▷ Check precondition 2 for first rewrite
        throw CheckFailure
      ⋮      ⋮      ⋮                              ▷ More preconditions checked (omitted)
                                         ▷ Do Rewrite once all preconditions are checked
    local_y ← GETVALUE(Y)    ▷ Values assigned to variables are cached into local variables
    local_z ← GETVALUE(Z)
    local_x ← local_y + local_z              ▷ Only the “work” of the rewrite is generated
    try:                                                         ▷ Start Second rewrite
        ⋮      ⋮      ⋮                                    ▷ Second rewrite’s body omitted
    catch  CheckFailure:                ▷ JUMP target for failed matches of second rewrite
        ▷ Local variables used by state731 are passed as arguments
        return state731(⋯, local_x, local_z, ⋯)
catch  CheckFailure:                     ▷ JUMP target for failed matches of first rewrite
    return null                  ▷ Return null to indicate that no rewrite were performed
```

Figure 12-8. High-level example of how generated code nests try/catch blocks to represent each rewrite. The catch blocks are the JUMP targets for when the match fails. Subsequent rewrites become more deeply nested under the try blocks.

```
(fn [rexpr simplify]    ; Rewrite function takes R-expr & active simplify func
 (StatusCounters/match_attempt)
 (let [**context** (ContextHandle/get)
       **threadvar** (ThreadVar/get)]
  (try
   (do           ; rewrites that modify the R-expr may not generate operations
    (try
     (try
      (do  ; status counter incremented after 1 rewrite is done.  performing a
           ;  rewrite does not have "code" sometimes (e.g. rearranging R-expr)
        (StatusCounters/jit_rewrite_performed)
        (try
         (do           ; perform a test, otherwise throw JITRuntimeCheckFailed
          (jit-precondition-to-check (is-bound-in-context? (. rexpr jv23463)
    **context**))
          (do
           (jit-precondition-to-check
            (let* []
             (jit-precondition-to-check (is-variable? (. rexpr jv23461)))
             (not (is-bound-in-context? (. rexpr jv23461) **context**))))
           (let* [local-cache23852 (get-value-in-context (. rexpr jv23463) **
    context**)]
            (do (set-value-in-context! (. rexpr jv23461) **context**
    local-cache23852)
              (try
               (try
                (try
                                                    ; the final returned R-expr
                 (make-jit-rexpr24202 (. rexpr jv23462) (make-constant
    local-cache23852) (. rexpr jv23464))
                 (catch JITRuntimeCheckFailed _
                                   ; Failure branch, return a different R-expr
                  (simplify (make-jit-rexpr24046 (. rexpr jv23462) (
    make-constant local-cache23852) (. rexpr jv23464)))))
                (catch UnificationFailure _
         ;    due to disjunctions, a unification failure might not result in 0
         ; so we catch this exception is handle explicitly instead of bubbling
                 (make-multiplicity 0)))
               (catch JITRuntimeCheckFailed _
                     ; Second failure branch, returns another different R-expr
                 (simplify (make-jit-rexpr23857 (. rexpr jv23464) (
    make-constant local-cache23852) (. rexpr jv23462))))))))))
         (catch UnificationFailure _
          (make-multiplicity 0))))
      (catch JITRuntimeCheckFailed _
       (simplify (make-jit-rexpr23624 (. rexpr jv23462) (. rexpr jv23461) (.
    rexpr jv23463) (. rexpr jv23464)))))
     (catch UnificationFailure _ (make-multiplicity 0))))
  ; the highest level runtime check returns nil to indicate no change/rewrites
   (catch JITRuntimeCheckFailed _ nil))))
```

Figure 12-9. Example generated Rewrite from JIT compiler.[208]

12.4.5 Generating Aggregators in the JIT-Generated Rewrites

So far, the generated operations that we have talked about have only been about generating *straight-line* code that corresponds to a sequence of rewrites performed against an **R**-expr. As discussed previously in sections § 6.5 and 11.6.3, aggregation requires combining the results of multiple contributions. This requires splitting the **R**-expr into smaller disjunctive **R**-exprs, which can each be processed individually. This is done through the use of iterators, which are designed to loop over the domain of a variable (section §11.7). By having iterators split the **R**-expr using assignments to variables, rather than returning arbitrary **R**-exprs, we can easily represent this as a new JIT-generated state (line 8 of figure 12-10). The current value of a variable can be stored with a local variable at runtime. Conversely, an arbitrary **R**-expr cannot be efficiently represented at runtime given the design of the JIT compiler.

When getting iterators from an **R**-expr, we create a precondition on finding an iterator that behaves the way that we need when considering the code that we have generated in the rewrite (lines 3 to 6 of figure 12-10). In the event that we cannot find an iterator that can support our compiled subroutine, the rewrite will be aborted with a precondition failure, just like in section §12.4.4 (line 6 of figure 12-10). To implement this behavior inside of the JIT-compiler, there is special code for handing rewrites on the efficient inner and outer aggregator kinds (section §11.6.3), and the primitive rewrites for aggregation are not at all used by the JIT-compiler.

To accumulate the aggregated values, we use a globally accessible container that holds the current aggregated value. This is the same as was done in section §11.6.3 figure 11-20.

A conceptual example of an aggregator in a JIT-generated rewrite is shown in figure 12-10.

12.5 What is JITable?

Most of the primitive **R**-expr kinds can be included in a JIT-generated **R**-expr state, however there are some **R**-exprs that we exclude.

From section §12.4, we can observe that the JIT rewrite compiler requires that the **R**-expr after a rewrite has been applied is *predictable*. If a rewrite does not result in a predictable **R**-expr, then the JIT compiler would be required to generate many different branches for the different states that could result from a particular rewrite. If there are too many JIT-generated states, the JIT compilation will become inefficient, as it is difficult to benefit from reuse of existing JIT-generated states. Therefore, we exclude **R**-exprs that are frequently rewritten in difficult-to-predict ways. To handle these **R**-expr kinds, they are represented using holes in the JIT-generated **R**-expr; therefore, they are passed around and simplified as opaque

[210]The JIT compiler also generates the necessary code to implement the GETITERABLES for all JIT-generated **R**-expr kinds.

```
1:  global aggregationResult ← agg_null ▷ Where the resulting value of aggregation is stored
2:  aggregationResiduals ← 0                    ▷ R-expr of still un-evaluatable R-exprs
3:  allIterators ← GETITERABLES(⋯)[210]         ▷ Get iterators from R-expr, section §11.7
4:  iterator ← STARTITERATORWITHVARORDER(allIterators, [X, Y, Z])
5:  if iterator == null : ▷ If there is not the right iterator, the precondition for the rewrite fails
6:      throw CheckFailure                      ▷ Abort rewrite, as described in section §12.4.4
7:  for ⟨$x$, continuation⟩ ∈ iterator.Run() :                        ▷ Run iterator
8:      R ← state941($x$, continuation, ⋯) ▷ JIT-generated R-expr representing agg body
9:      R' ← SIMPLIFY(R, $\mathcal{C}$) ▷ Evaluate the aggregator's body in its own R-expr rewrite "unit"
10:     if R' $\neq$ 0 :                            ▷ Check if there is an R-expr residual
11:         aggregationResiduals ← aggregationResiduals + R'
12: if aggregationResiduals == 0 :                       ▷ if no R-expr residual
13:     ▷ The aggregation is done, and the result is in aggregationResult
14:     if aggregationResult == agg_null :
15:         "return" 0
16:     else
17:         "return" (A=aggregationResult)
18: else
19:     ▷ Return the aggregator with the delayed residual R-expr
20:     "return" (A=sum(X,(X=aggregationResult)+aggregationResiduals))
```

Figure 12-10. Conceptual example of aggregation performed by a JIT-generated rewrite on the outer aggregator. The accumulator variable is initialized on line 1 and will be modified inside of SIMPLIFY, called on line 9. Like with efficient aggregators in section §11.6.3, the aggregation is done when the residual **R**-expr is rewritten as 0 (line 12).

R-expr pointers and not otherwise handled by the JIT.

The **R**-exprs kinds which we do not support are as follows:

1. An obvious case is the **R**-expr that converts from the Dyna AST into **R**-exprs (section §11.3). This **R**-expr can result in *any* possible **R**-expr depending on the shape of the program. Furthermore, these **R**-exprs are only used when translating the Dyna source. This rarely happens in the running of the program. Hence, we are unlikely to benefit from making it run faster.

2. Another case of unpredictable **R**-exprs are disjunctions. For example, suppose that we have $\mathtt{R}_1$+$\mathtt{R}_2$+$\mathtt{R}_3$+$\cdots$+$\mathtt{R}_n$, if we consider that each $\mathtt{R}_i$ could be independently rewritten as `0`, then there are potentially 2^n possible states in which this **R**-expr could be rewritten. This means that we could potentially generate these 2^n states and the corresponding rewrites between them. To limit this, I only allow *small*[211] disjunctions to be generated in the JIT. The idea here is that a small disjunction likely represents a function that would benefit from compilation. For example, the rectified linear function $f(x) = \max\{0, x\}$ would be represented as the **R**-expr `(A=max(Y,(Y=0)+(Y=X)))` which includes a disjunction. In contrast, a large disjunction might represent a table of values or a memo. Hence, retrieving data from the hash table trie (section §11.6.1)

[211]Small disjunctions are currently set as a disjunction with fewer than 8 children. The limit of 8 was chosen without any data to inform this choice. In the future, one may wish to reconsider this limit and experimental with different configurations of the JIT compiler.

would be more efficient than a chain of if-expressions.

3. Memo read **R**-exprs are also excluded for the same reason as disjunctions.

4. We also do not include user-defined **R**-expr kinds in the JIT-generated **R**-expr kinds. The reason is that user-defined **R**-exprs can be redefined at the REPL or by guessing with memoization. This means that any JIT state that included a user-defined **R**-expr (and rewrites on those states) would have to be invalidated. Furthermore, user-defined **R**-exprs often indirect through memo tables and include a disjunction inside of their definition. Hence, a generated **R**-expr that includes a user-defined **R**-expr would not be able to represent that much in the first place.

While limiting disjunctions, memos, and user-defined **R**-exprs seems like a big limitation, we can still work around it. One way to work around this is by *merging* a JIT state with the **R**-expr, which is held in as an opaque pointer inside of the JIT hole. This is accomplished by allowing JIT to create a rewrite that matches the **R**-expr contained in the hole and add it to the current JIT state. This means that once the **R**-expr contained in a hole has been sufficiently rewritten, it can be merged to create a larger JIT-generated **R**-expr state.

12.6 Starting the JIT Compiler

So far, we have that we can take an **R**-expr and represent it as a JIT-generated **R**-expr state and then create rewrites on those states. However, this leaves the question of how to create the initial *seed* JIT-generated states.

The way that I do this is by starting from 1) user-defined **R**-exprs (as in figure 12-3) and 2) **R**-exprs that are nested under disjunctions and turning these into JIT-generated states. The reason to use user-defined **R**-exprs is that these are likely to reappear a lot and be used in the same way. Hence, using a user-defined **R**-expr as a seed is akin to compiling a user-defined method. The second case for an **R**-expr under a disjunction is that the disjunction itself does not get compiled. However, the leaves of the disjunction might still contain code.

This choice of seed is, of course, a *heuristic* and something that future work may wish to revisit.

12.7 Experiments: Benchmarks

Our ability to run real-world programs is still limited by the runtime performance of the implementation on the whole. In chapter §16, I discuss possible directions to further improve the runtime of Dyna. So, unfortunately, at this time, I do not have benchmarks on "real-world" problems. However, we can still benchmark a small Dyna program to measure how the JIT and non-JIT rewrites compare. Additionally,

```
f(X) += I for range(0, X, I).[212]
```

(a) Dyna

```
def f(X):
  return sum(range(X))
```

(b) Python

```
(defn f [X]
  (reduce + (range X)))
```

(c) Clojure

```
int f(int X) {
  return IntStream.range(0,X).sum();
}
```

(d) Java using Streams

Figure 12-11. Simple program that computes $\sum_{i=0}^{x} i$. To measure the *overhead* vs time doing the additions, we can vary the value of `X` to increase the size of the problem.

we can compare again other programming languages running the same program to observe how more the Dyna implementation needs to be improved.

The program we will use to compare computes the sum of integers between 0 and x. I am running this program without special optimizations (such as recognizing that $\sum_{i=0}^{x} i = x * (x+1)/2$), so this test is an evaluation of how fast we can run a loop over the values of x and perform a small numerical processing task during each step. Figure 12-11 shows what this program looks like in a number of different programming languages.

First, we can observe in figure 12-12, the ratio of the runtimes between the

[212]The `range(0,X,I)` is a builtin that loops over integers between `0` and `X`. The built-in can be written explicitly, as done here, or inferred through a conjunction of an `int` constraint and two `lessthan` constraints that provides both an upper and a lower bound. The `range(0,X,I)` constraint provides an iterator over the domain of the variable `I`, which is used to drive the loop.

[213]The run for the first value has been removed from the graph. The reason is that it included the time for the JIT compiler to generate the rewrites and compile the Clojure code. For the initial run that was eliminated, the JIT compiled version was 20 times slower than the non-JIT version. For all subsequent runs shown in figure 12-12, the JIT-compiled was usually much faster.

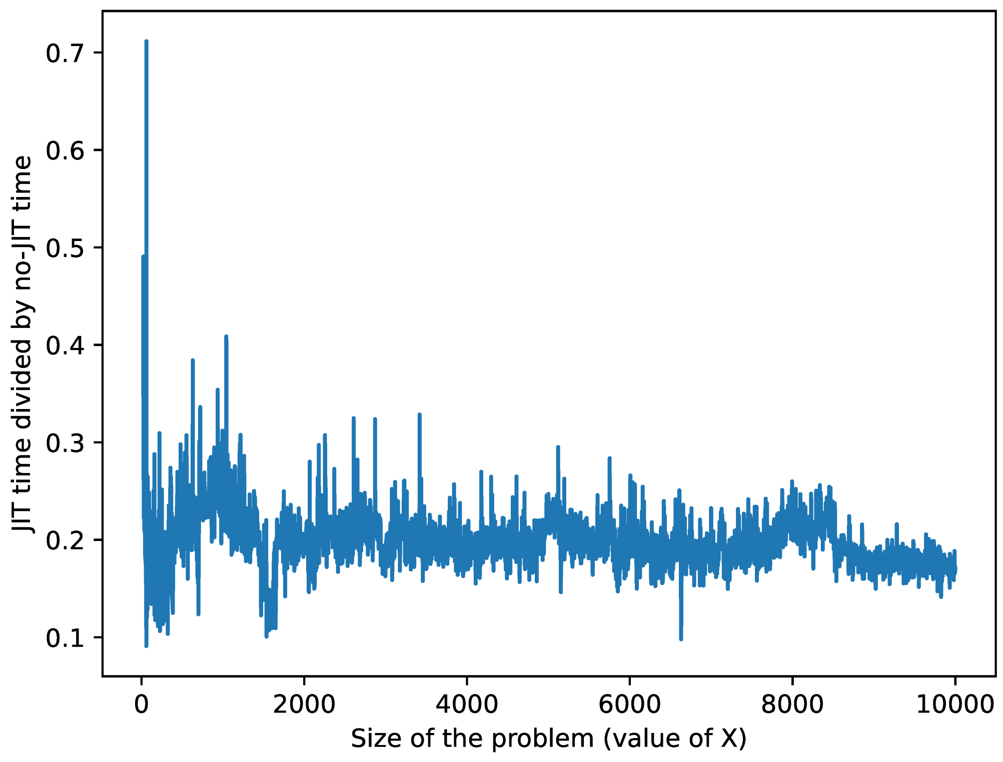
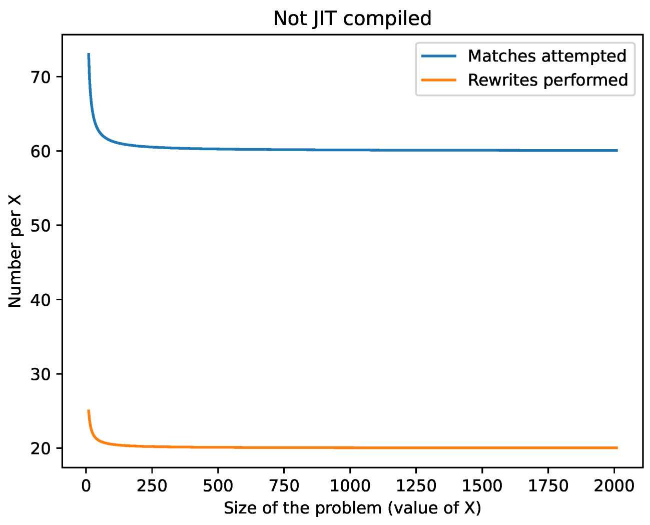


Figure 12-12. The y-axis is the ratio between the JIT compiled version and the non-JIT compiled version (chapter §11). We can see that the JIT-compiled version runs roughly about five as fast as the non-JIT-compiled version (yay!).[213]

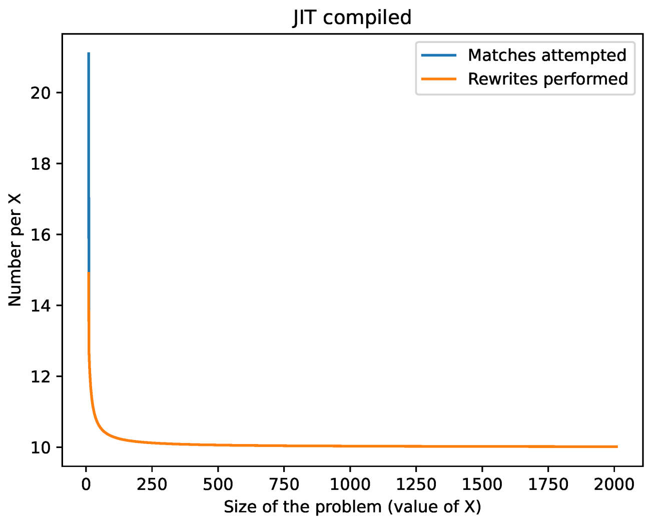
non-JIT-compiled code and the JIT-compiled code. The JIT makes this program run roughly five times faster. In making the JIT-generated version, the initial seed state (section §12.6) was created before the program started running, but no rewrites were created. The first *warm up* run has been removed from the graph. The first run, which triggers the generation of rewrites, is 20 times slower than the first run of the non-JIT-compiled version. The first run being slower is consistent with what our expectations as the first run is performs extra work generating the rewrites.

To try and understand why the JIT-compiled version is faster, we turn our attention to figure 12-13. Figure 12-13 shows the number of times that SIMPLIFY *attempted* to match a rewrite vs. the number of times a rewrite was applied. If you recall from section §8.A, the attempted matches come from the design of SIMPLIFY where it will dispatch against the **R**-expr kind, and then sequentially check if any of the rewrites can be applied by checking preconditions. For the non JIT-compiled version, we first observe that SIMPLIFY attempts 60 matches per number summed. Without JIT-compilation, we observe that only $\frac{1}{3}$ of the rewrites match. This means $\frac{2}{3}$ of the time is wasted on checking if "useless" rewrites could apply. Conversely, after generating new **R**-expr states and making JIT-generated rewrites, we see that 100% of the attempted matches match and run rewrites successfully. The JIT *only* generates rewrites for the scenarios that we have been previously encountered. Because this program is repeating the same operation multiple times (summing an integer), there are no unnecessary rewrites loaded into the system, and therefore no wasted effort in searching through matching rewrites. This is similar to the kinds of numerical programs that are typical of machine learning algorithms

In addition to observing that the JIT does help the runtime, we also compare it against other programming languages. In figure 12-14, I show the runtime per number summed for Dyna with and without JIT-compilation, Python, Clojure, and Java. We can observe that the JIT-compiled **R**-exprs rewrites are 120 times slower



(a) This shows the number of rewrites attempted and the number of rewrites that were successfully matched and applied. Here, there are 60 rewrites attempted per number summed, with 130 rewrites as a constant overhead. There are 20 rewrites that are successfully applied per summed number and 50 that are a constant overhead.



(b) This shows the number of rewrites attempted and successfully applied when the JIT is used to generate **R**-expr kinds and rewrites. Here we can observe that *all* rewrites attempted are successfully matched. Furthermore, there are only 10 additional rewrites per number summed (vs. 20 without the JIT), and there is a baseline of 40 rewrites of constant overhead. This shows that the JIT performs the same amount of work using fewer rewrites.

Figure 12-13. Creating JIT-generated **R**-expr kinds significantly reduces the number of attempted rewrite matches, and the number of rewrites required to do productive work, in terms of rewrites matched.

than Python (6010 times slower than Java). Further delving into this in figure 12-15, we can profile where the JIT-compiled version is spending its time. It appears that much of the time spent is on non-**R**-expr rewriting activities, such as initializing hash-tables used for the context's internal data structures, computing hash-codes for **R**-exprs, and accessing globally accessible objects such as the context. The *good* news from figure 12-15 is that there is a lot of opportunity to make the system run faster through careful engineering to avoid these unnecessary overheads. The bad news from figure 12-14 is that Dyna is 120 times slower than Python and likely still too slow for "real-world" problems.

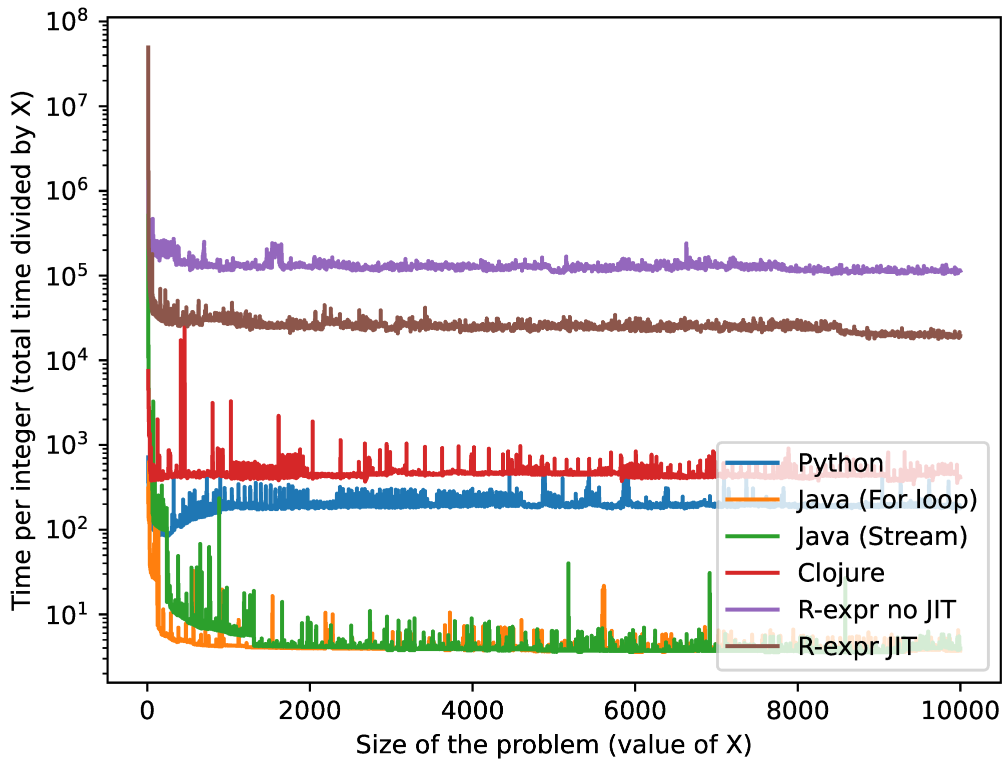


Figure 12-14. Results of running the small benchmark programs from figure 12-11. The y-axis is the total time of running the benchmark 10 times and dividing it by the value of x, which is the number that is being summed to. Hence, this plot shows the scaling of time per summed integer. The non-JIT **R**-expr rewriting (chapter §11) is 31168 times slower than the fastest runtime, which is the Java program using a for-loop. The JIT compiled **R**-exprs (this chapter) is 6010 times slower than Java.

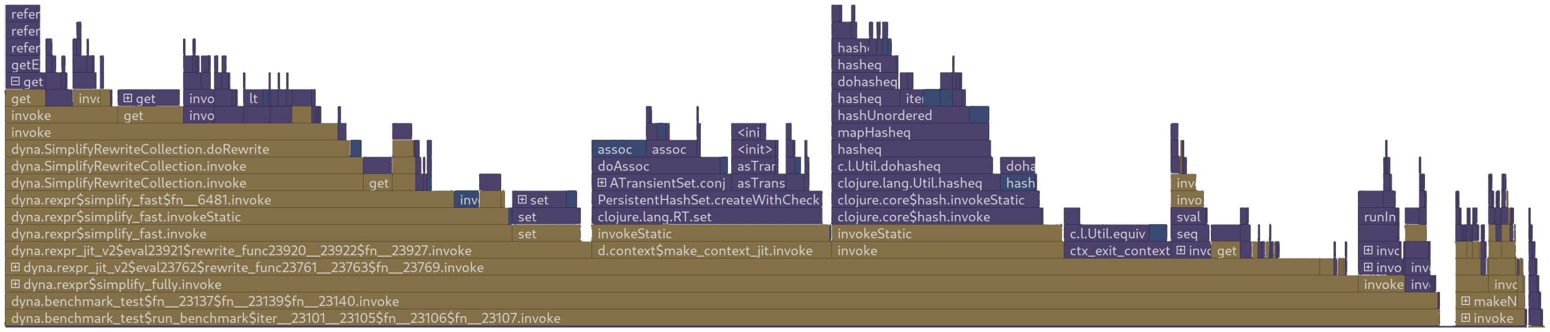


Figure 12-15. This is a flame graph for the benchmark from figure 12-11a running with JIT-generated rewrites. We can observe that 85.19% of the runtime is taken up by a JIT-generated rewrite that represents the outer part of the aggregation (conceptually what is in figure 12-10). This rewrite internally calls simplify and performs other rewrites. We can observe that 23% of the overall time (27% of the rewrite) is spent rewriting the JIT-generated **R**-expr that represents the body of aggregation (line 9 in figure 12-10). The creation and manipulation of nested contexts account for 22% of the overall runtime (26% of the rewrite). 15% of the overall runtime is spent creating **R**-exprs, more specially the profile seems to indicate that creating cached hash codes is somehow the expensive part of this. Approximately 10% of the overall runtime is spent reading and writing from global variables. This is, of course, subject to the profiler's ability to measure these typically fast functions, but accessing the global context does happen frequently, so it is conceivable that this is a correct measurement.

Chapter 13

Object Oriented Programming in a Pure Declarative Language

13.1 Dynabases

Object oriented design in Dyna with dynabases was originally *proposed* by Eisner and Filardo in [59] with an expanded discussion in Filardo's dissertation [66]. Unfortunately, the prior work did not include an implementation or complete details. This dissertation contributes for the first time working details of the syntax for dynabases (section §2.9) and a working implementation of dynabases. The design of dynabases in this dissertation has some differences from the one proposed in [59, 66], though these differences will be immaterial to most users.

Dynabases provide prototype-based inheritance [49, 59, 133, 136] for logic programming. A dynabase object can be extended by adding rules to a dynabase after it has been “created”. Dynabases are *open*, in that additional rules can be added to the dynabase’s “class definition” at any point, even after “instances” of the class have been instantiated. Allowing rules to be added to a dynabase is consistent with Dyna’s out-of-order execution, where the program is a collection of rules, and the system is looking for a consistent assignment to all terms defined by the program, regardless of the order in which they were defined.

To implement this behavior, dynabases are not allocated objects, as is typical in a procedural programming language. Rather a dynabase is represented using an immutable dynabase term identifier, as will be described in section §13.2. The dynabase identifier can be used to call terms defined on the dynabase, much the same way that a structured-term (section §2.1.1) can be passed as an argument to a term. As such, we will see in section §13.3 that dynabases can be implemented as *almost* a purely syntactic transformation before the program is converted to **R**-exprs (chapter §7). This is conceptually similar to what appeared in previous publications about Dyna’s object-oriented design [59, 66] as well for prototype-based inheritance in Prolog with the LogTalk project [49].

13.2 Dynabase Object Representation

Dynabases allow closures capturing *all* variables present when they are instantiated. Dynabases also support single inheritance. Conceptually, we can think of a dynabase’s identifier as an immutable list of dynabases types that the dynabase instance inherited from. If we are to use the Dyna notation from section §2.1.1, then a dynabase object would look something like:

```
679 [dynabase_123[CapturedVar1, CapturedVar2], dynabase_789[X, Y]]
```

Here, the numerical identifiers 123 and 789 on line 679 identify the dynabase class types that this object inherits from. The variables CapturedVar1, CapturedVar2, X, Y are the captured variables that were present when the dynabase was instantiated. The presentation on line 679 is conceptually similar to what was presented previously in Eisner and Filardo [59].

While the array of identifiers on line 679 is *conceptually* correct, we instead going to use a slightly different design in the implementation for efficiency reasons. Instead, we are going to group the dynabases by dynabase class identifier into a map and then have a list of lists to track the captured variables. As a Java type, this looks like line 680:

```
680 Map<DynabaseTypeID, List<List<𝒢>>>
```

Here, DynabaseTypeID is an opaque class identifier. The identifier is generated to be unique when the Dyna program is converted from the Dyna AST into **R**-exprs. The DynabaseTypeID identifier corresponds with the dynabase's "*class*" rather than an instance of the dynabase. For the presentation in this chapter, I will use the line number on which a dynabase is defined on as the identifier.

The value associated with each dynabase class identifier is List<List<$\mathcal{G}$>>. The inner List<$\mathcal{G}$> contains the values of the captured variables when the dynabase is created. The order in which the values are held in `List<𝒢>` is chosen arbitrarily, but the generated code will consistently reference the variables in the slots.

The second list of lists is to support the scenario where a dynabase type can appear *more than once*. This happens when *self-inheritance* is used. This will be discussed in more detail in section §13.5.

13.2.1 Why Capture *All* Variables?

Dynabases capture all local variables when created. The reason why it needs to capture all of the variables in the current rule is that a dynabase might be modified or referenced from places other than where it was constructed. For example, consider the dynabase on line 681.

```
681 f(X) = new {}.
682
683 db_a = f(1).
684 db_b = f(2).
685
686 db_a.a = 123.
687 db_b.a = 456.
688
689 assert db_a.a == 123.
690 assert db_b.a == 456.
691 assert f(1).a == 123.
```

Here the functor `f(X)` has one variable `X` and returns an "empty" dynabase. On lines 683 and 684, we construct two different dynabases. We then proceed to modify the dynabases by setting the "field" 'a' on lines 686 and 687. The values we have assigned in each of these dynabases are distinct from each other (lines 689 and 690).

We, therefore, must have some way to track this distinction between dynabases.

If this were a procedural language, then we would use the fact that we are *creating* two different instances of an object with two different locations in memory as a result of the two different function calls on lines 683 and 684. However, by the nature of being declarative and functional, we require that *every* time that we call a function with the same arguments (Dyna term), we get back the same result. Hence, we can get db_a again by calling f(1) a second time as on line 691.

Therefore, we would either need a design that is *required* to store any dynabase that is returned from a user-defined function or "recreate" the same dynabase multiple times. We have opted to recreate the same dynabase object multiple times. This is done by returning an immutable identifier from f(1) rather than a mutable object. Furthermore, f(·) defined on line 681 must return *different* dynabase identifiers when f(1) is called on line 683, and when f(2) is called on line 684. To make this work, there needs to be a distinguishing feature between the call to f(1) and f(2). The only distinguishing feature is the assignment to variables present in the rule. (In this case, the variable X.) Hence, we *capture* the value of all variables when creating a dynabase.

Consequence of this design: An consequence of this design is that we cannot construct two (or more) dynabases at the same time in a single rule.

For example, the following program does not work:

```
692 does_not_work(X) = (A = new {}), (B = new {}), A.
```

The reason for this is that both the A dynabase and the B dynabase need to capture a ground value reference of the other (we do not allow for cycles in the ground values). In my opinion, this is not an issue, as it can easily be worked around by ensuring that there is at most one dynabase constructed per functor:

```
693 make_B(X) = new {}.
694 works(X) = (A = new {}), (B = make_B(X)), A.
```

If a user writes a rule like line 692, this is detected and reported as an error to the user.

Future work may consider adding an automatic transformation to the AST-to-**R**-expr front-end to perform a transformation like line 693 to allow multiple dynabases to be defined in a single Dyna rule.

13.3 Desugared Dynabases

In converting a Dyna program with dynabases into an **R**-expr, there are two cases that we have to handle: constructing a dynabase and calling methods (accessing fields) on a dynabase. Both of these cases are handled by introducing *new* **R**-expr types, which are called $\texttt{dynabaseCreate}_{\texttt{dynabaseTypeID}}$(SuperDynabase, Var1, ⋯, VarN, Result) and $\texttt{dynabaseAccess}_{\texttt{dynabaseTypeID}}$(Dynabase, Var1, ⋯, VarN).

The dynabaseCreate **R**-expr is annotated with an opaque symbol dynabaseTypeID,[214]

[214]There is one **R**-expr class that implements the dynabaseCreate **R**-expr, and dynabaseTypeID is held in a metadata field on the **R**-expr class. This is the same as with structural-equality constraints that track the name of the structural-term through a metadata field, section §8.1.

which is picked to be unique when the program is translated from the Dyna source code. The variable SuperDynabase is the dynabase inherited from. The variables Var1, . . ., VarN are the captured local variables (section §13.2.1).

To illustrate the translation process, consider the following Dyna program on lines 695 to 704 with dynabases. To make the translation easier to read, I will show the translation using "*pseudo Dyna*", which is an intermediate step in translating from Dyna into **R**-exprs.

```
695 e = new {
696   val = 123.
697 }.
698 f(X,Y) = new {
699   val += X.
700   val += Y.
701   func(X) = $self.val * X.
702 }.
703
704 g = f(1,2).val.
```

With dynabases desugared:

```
705 e = dynabaseCreate_695($nil, Result), Result.            % From line 695
706 val(Dynabase) = dynabaseAccess_695(Dynabase), 123.       % From line 696
707 f(X,Y) = dynabaseCreate_698($nil, X, Y, Result), Result. % From line 698
708 val(Dynabase) += dynabaseAccess_698(Dynabase, X, Y), X.  % From line 699
709 val(Dynabase) += dynabaseAccess_698(Dynabase, X, Y), Y.  % From line 700
710 func(DBase, X) = dynabaseAccess_698(DBase,_,Y), val(DBase)*X. %From L701
711
712 g = val(f(1,2)).                                        %  From line 704
```

Note that the rule val appears on both dynabase_695 and dynabase_698. The distinguishing factor between the val rules are the dynabaseAccess **R**-exprs. The reason for this is that dynabases inherit from each other, and as such, there are rules that are defined in different dynabases that need to be combined. This means that a "compile-time" function dispatch only considers the functor name (`val` in this example). A secondary "runtime" dispatch selects the relevant rules that define the `val` terms from lines 706, 708 and 709. This is done by using the value assigned to the `Dynabase` variable to rewrite the `dynabaseAccess` **R**-expr.

The approach of grouping by functor name and using `dynabaseAccess_698` is *necessary* to allow modifications to the dynabase after it has been initially defined. For example, consider line 715 where we define a new value `v` on a dynabase.

```
713 db(X) = new {a = 1.} for X > 0.
714 db(X) = new {b = 2.} for X <= 0.
715 db(X).v = 2 + X.
```

On line 715 we do not know the dynabase *type* we are modifying. In fact, we are actually defining 'v' on two different dynabase types at the same time. When we desugar line 715, we get something like line 716 below.

```
716 v(Dynabase) = (Dynabase = db(X)), 2 + X.                  % From line 715
```

Here, we have that the dynabase is represented using the variable `Dynabase`. We only need to find a value of `X` such that the expression `Dynabase = db(X)`, is true. This can be handled using rewrites on `dynabaseCreate`.

13.4 Dynabase Rewrite Rules

The dynabase rewrite rules are fairly straightforward. The inheritance relations between dynabase types are tracked via a global data structure and updated whenever a new dynabase instance is created by the `dynabaseCreate` rewrite rules. Some of the rewrite rules for dynabases are conditionally enabled and disabled depending on the inheritance between different dynabase types.

$$\texttt{dynabaseCreate\_698(SuperDB, X, Y, Result)} \xrightarrow{88} (\texttt{Result}=\textit{dynabase object}) \quad \textbf{if } \texttt{SuperDB, X, Y} \in \mathcal{G}$$

$$\texttt{dynabaseCreate\_698(SuperDB, X, Y, Result)} \xrightarrow{89} (\texttt{SuperDB}=\textit{dbase object})*(\texttt{X}=x)*(\texttt{Y}=y) \quad \textbf{if } \texttt{Result} \in \mathcal{G} \textbf{ and } \texttt{dynabase\_698} \text{ does not self-inherit}$$

$$\texttt{dynabaseAccess\_698(Dynabase, X, Y)} \xrightarrow{90} ((\texttt{X}=x_1)*(\texttt{Y}=y_1) + (\texttt{X}=x_2)*(\texttt{Y}=y_2) + (\texttt{X}=x_3)*(\texttt{Y}=y_3) + \cdots) \textbf{ if } \texttt{Dynabase} \in \mathcal{G} \quad \triangleright \textit{allow self-inherit}$$

$$\texttt{dynabaseAccess\_698(Dynabase, X, Y)*dynabaseAccess\_695(Dynabase)} \xrightarrow{91} \texttt{0} \textbf{ if } (\texttt{dynabase\_698} \textbf{ and } \texttt{dynabase\_695} \text{ are incompatible (w.r.t. inheritance)})$$

$$(\texttt{dynabaseCreate\_698(SuperDynabase, X, Y, Dynabase)} * \texttt{dynabaseCreate\_695(SuperDynabase2, Dynabase)}) \xrightarrow{92} \texttt{0} \textbf{ if } (\texttt{dynabase\_698} \textbf{ and } \texttt{dynabase\_695} \text{ are incompatible (w.r.t. inheritance)})$$

$$(\texttt{dynabaseCreate\_698(SuperDynabase, X, Y, Dynabase)}* \texttt{dynabaseAccess\_695(Dynabase)}) \xrightarrow{93} \texttt{0} \textbf{ if } (\texttt{dynabase\_698} \textbf{ and } \texttt{dynabase\_695} \text{ are incompatible (w.r.t. inheritance)})$$

Rewrite rule 88 creates a dynabase object in the case that all of the arguments are ground. The `SuperDB` variable is the dynabase object from which the created dynabase inherits. In the case where there is no parent dynabase, then this variable is set to the value `$nil`.

Like with structured-terms, the `dynabaseCreate` **R**-expr both *creates* dynabase objects and destructure dynabase objects, getting access to the variables captured by the dynabase closure. This is done using rewrite rule 89, which requires that the `Result` variable is assigned dynabase that inherits from dynabase_698. Rewrite rule 89 is used when evaluating `Dynabase = db(X)` from line 715. Further note that rewrite rule 89 is *disabled* in the case that a dynabase type self-inherits. I will discuss the reason further in section §13.5.

Rewrite rule 90 is used to match a dynabase object for rules that were defined inside of the dynabase's definition. This was used by lines 706 and 708 to 710. Observe that rewrite rule 90 rewrites as a *disjunction* of bindings to the variables `X` and `Y`. This is a disjunction over *all* of the times that a dynabase type has been inherited from in constructing a dynabase object. A single dynabase type appearing more than once in the case of self-inheritance, which will be discussed in section §13.5. Note, in the case that a dynabase type does *not* self-inherit, rewrite rule 90 will only rewrite a single assignment to the variables `X` and `Y` for a given assignment to the `Dynabase` variable.

Rewrite rules 91 to 93 implement "*type checks*" that cause an **R**-expr to be rewritten as `0`, thus eliminating dead code. Rewrite rules 91 to 93 do not require that the value of the `Dynabase` is known for these rewrites to apply. However, these rewrites depend on the information about how dynabases inherit from each other. This inheritance information can be updated after the system has started running. As such, applications of rewrite rules 91 to 93 depend on the state of dynabase inheritance, and usage of these rewrite rules is tracked using the assumption mechanism previously discussed in section §10.4.1.

13.5 Self-Inheritance

As hinted at in the previous section, self-inheritance with dynabases allows for some strange interaction and causes rewrite rule 89 to become disabled in the case there is self-inheritance. So what is self-inheritance, why is it interesting, and why does it cause things to break? To answer this, let us work through a small example of self-inheritance to see what it is, how it works, and how we can build towards a logical inconsistency that prevents us from using rewrite rule 89 in the case of self-inheritance.

We define dynabase_717 on line 717, which inherits from the argument `X`.[215] By passing in a dynabase that has already inherited from dynabase_717, we can self-inherit, as is done on lines 722 and 723.

```
717 f(X) = new (X) {
718   z += 1.
719 }.
720 a = new {}.
721 af = f(a).
722 aff = f(af).
723 afff = f(aff).
724 assert af.z == 1
725 assert aff.z == 2.
726 assert afff.z == 3.
```

Observe that we have defined ‘z’ on line 718 to essentially *count* the number of times that dynabase_717 has been inherited from. The way this works is that ‘z’

[215] An alternate way to get self-inheritance is by defining a term in terms of itself using recursion. Something like

```
f(N) := new f(N-1) { z += N. }.
f(0) := new {}.
```

is desugared to use the dynabaseAccess_717 **R**-expr:

```
z(Dynabase) += dynabaseAccess_717(Dynabase, X), 1. % From line 718
```

And then the dynabaseAccess_717 **R**-expr will rewrite using rewrite rule 90 into a disjunction of the different values of `X` that contributed to creating the ‘`afff`’ dynabase object:

$$\texttt{dynabaseAccess\_717(afff, X)} \xrightarrow{90} \texttt{(X=a) + (X=af) + (X=aff)}$$

Now, ‘`afff.z`’ returning the value `3` is exactly what we want to happen using the `dynabaseAccess`, which is used for rules defined inside of the dynabase’s initial definition. Now, let us consider a scenario where `dynabaseCreate`, which is disabled, is used instead.

Before we do work through the `dynabaseCreate` scenario, let us build up to this case with a by defining ‘`w`’ as a standard, non-dynabase, Dyna term:

```
w(D) += 1.
assert w(a) == 1.
assert w(af) == 1.
assert w(aff) == 1.
assert w(afff) == 1.
```

There are no tricks here in defining ‘`w`’ in that we sum the value `1` for every unique term passed as an argument to ‘`w`’. Now, let us define `w2` with a constraint on the variable ‘`D`’.

```
w2(D) += (D = f(X)), 1.
assert is_null(w2(a)).
assert w2(af) == 1.
assert w2(aff) == 1.
assert w2(afff) == 1.
```

Here '`(D = f(X))`' requires that there exists some value for `X`, such that the value held in the variable `D` is returned. As already discussed in section §13.3, line 733 is equivalent to line 738.

```
738 f(X).w2 += 1.
```

Now we can begin to see the problem. The interpretation of line 738 should be that the value of '`w2`' defined on the dynabase returned from the function `f(X)` should be equivalent to the value of '`z`'. Hence, it should be counted *multiple* times. However, this is different from the interpretation of '`w2`' as defined on line 733.

Rather than trying to reconcile the two ways in which '`w2`' could be interpreted, we instead *disable* rewrite rule 89. This means that if a user writes a rule like line 733 or line 738 they get back unrewritten **R**-expr instead of the value 1 or 3. Admittedly, an unrewritten **R**-expr is rarely the desired result, but at least this does not give an incorrect answer. Furthermore, this issue can be avoided by defining rules inside of the dynabases definition, as with the definition of '`z`' on line 718.

Finally, I want to reiterate that without self-inheritance, dynabases are conceptually *very* simple. The reason for the additional complexity of dynabases largely has to deal with self-inheritance, and for rewrites on dynabases to work as "compile time" rewrites when possible.

13.6 Comparison with 2011 Proposal for Dynabases

The design presented in this chapter is my own design for dynabases. However, it is not the only possible design for dynabases. When dealing with *simple* dynabases that do not use self-inheritance or "external modifications" of the dynabase (such as line 715), then all proposed implementations of dynabases exhibit similar behavior. However, how self-inheritance (section §13.5) and external modification (e.g. line 715) are handled does differ between the proposed implementations.

In Eisner and Filardo [59], the authors proposed "collect[ing] the relevant rules by recursively traversing [the dynabase's] parent [reference[216]]". This is, of course, the right *high-level* idea, but as we saw in section §13.5, there are many details that are needed to make this operational. Furthermore, I note that Eisner and Filardo [59] did not distinguish between `dynabaseCreate` and `dynabaseAccess` **R**-exprs as was done here. Instead, they only use Dyna-to-Dyna translations like `(Dynabase=db(X))` (e.g. line 716), which depends on the interpretation of their "`dynabaseCreate`" operation. The issue of self-inheritance (the disjunction in rewrite rule 89) was not addressed in [59].

Filardo did expand on the [59] design in his dissertation [66] section 6.2. In section 6.2.2 "Rule-Collection Semantics With Recursive Owner Writes", Filardo also recognized the issue of self-inheritance. Filardo's section 6.2.2 attempted to fix

[216]The dynabase's parent reference is akin to the `SuperDynabase` variable in section §13.3, which could be stored in a list data structure, as discussed at the beginning of section §13.2.

the issue of self-inheritance by introducing a concept he called “focused dynabase” into the “collect-relevant-rules” function. Unfortunately, Filardo did not work out the details of how the “focused dynabase” would integrate into the execution of the Dyna program that had dynabases, and how it works with calls to other terms defined on a dynabase. Although the design from [59] and [66] does not have an implementation and the description in [59, 66] is partially incomplete, I believe that regardless of how the design from [59, 66] is completed, it would run into the same kinds of issues that we have in the case of self-inheritance (section §13.5).

To illustrate some of the complications with dynabases, and how the way this chapter handles dynabases vs [59, 66], let us work through the following example program on lines 739 to 753.

```
739 f(X) = new (X) {
740   q += 1.
741   r += $self.q.
742   s += $self.u.
743   t += $self.ss.s.
744 }.
745 a = new {
746   ss = $self.
747 }.
748 af = f(a).
749 aff = f(af).
750 afff = f(aff).
751 afffb = new afff {
752   u += 1.
753 }.
```

Here, I am defining dynabase_739 as a dynabase that self-inherits. Observe that the ‘q’ term on line 740 is the same as the ‘z’ from section §13.5 in that it counts the number of times that dynabase_739 is inherited from. However, this time, we additionally have the 'r' term that sums the value of 'q' every time it is added. The question we need to answer to understand how the system works is how **$self** on line 741 is defined as rules are added by recursively traversing the parent references.

Before delving into the discussion about [59, 66], I will note that for this example running under the design presented in this chapter, we have `afffb.r == 9` and `afffb.t == 9`, as **$self** always reference the dynabase that is being called (in this case the value `afffb`). Hence, **$self**`.q` on line 741 returns the value `3`. Furthermore, the call to **$self**`.u` on line 742 will refer to the 'u' defined on line 752, even though 'u' is defined in a descendant dynabase. This approach of making **$self** reference the dynabase object that and calling functions that have been overridden is consistent with the behavior that is typically used when doing object-oriented programming.

Now, we turn our attention to the design from [59, 66]. When evaluating `afffb.r`, I assume that the "focused dynabase" was intended to be `afffb`, which would have allowed the collect-relevant-rules function to identify that 'r' from line 741 needs to have 3 contributions. Next, we need to consider how **$self** references the relevant dynabase.

A first possible interpretation of **$self** would be to define **$self** as only referencing the current dynabase (this would have allowed for values computed in parent dynabases to be cached using memoization and modified by the children), then this would mean that **$self** is not able to reference functions on children dynabases.

This is problematic for compositionality. This interpretation of `$self` would mean that `afffb.t` is undefined as the `$self` on line 742 would be unable to reference the '`u`' defined on line 752. I note this design as [59] proposed caching the computation performed on parent dynabases. Filardo [66] rejects this idea and instead makes `$self` a dynamic variable that includes references to the rules defined on children dynabases.

Going to the second interpretation of `$self` as a dynamic variable that references the "focused dynabase", this still leaves the question of how `$self` and the `($self=db(X))` constraint from the Dyna-to-Dyna translation interact. Unfortunately, these details were glossed over. Presumably, had these details been added, it would have resulted in something like the `dynabaseAccess` **R**-expr that can handle self-inheritance and match in the case where it is a parent of the current dynabase.

Finally, I will note that [59, 66] proposed an access control mechanism that was introduced through an additional hidden variable that tracked the *owner* dynabase. I have chosen to do away with the owner concept for dynabases. Instead, anyone who has a pointer to a dynabase can modify it. This simplifies the design of dynabases and the amount of information one has to learn to use Dyna.

Chapter 14

Folding and Speculative Rewrites for Recursive Programs

When rewriting calls to user-defined **R**-expr (section §5.2.2.11), our general approach is to expand the user definition (rewrite rule 74). This is sufficient for non-recursive programs, as, without recursion, we can expand the **R**-expr to the maximum depth and have a finite-sized **R**-expr where rewrites can be applied arbitrarily. Unfortunately, when there is recursion, we cannot expand the **R**-expr all of the way.[217]

One way we can handle recursive programs while still being able to leverage the power of **R**-exprs is by using fold/unfold transformations. [22, 25, 58, 80, 97, 141] This essentially creates new automatically defined functions which are specialized

[217] As the **R**-expr will keep growing in size every time it is expanded. In other words, the program has a "*data dependency*" to ensure that the recursion eventually reaches a base case.

to a specific case. This can allow us to make inferences for recursive programs, which may be useful in some cases.

The unfold transform is analogous to rewrite rule 74 that we have already seen. A user-defined **R**-expr is expanded to its **R**-expr definition, with variables renamed as needed.

The fold transform is the reverse of unfolding an **R**-expr. Folding generates a new, automatically named **R**-expr. This can be written as rewrite rule 94, where ‘f’ is a newly chosen name.

$$\mathtt{R}_{\mathtt{f}} \xrightarrow{94} \mathtt{f}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3,\ldots,\mathtt{X}_n) \qquad \textbf{where}\ \{\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3,\ldots,\mathtt{X}_n\} = \text{vars}(\mathtt{R})$$

14.1 Why Fold a Program

The reason why one might want to fold a program is that, in the case of a recursive program, a folded definition can depend on itself. This allows us to perform rewrites that optimize recursive functions before it is unfolded in the context of the input to the Dyna program is processed. For example, if we have a higher-order function that performs an indirect function, we can create a specialized version of the function that does not perform an indirect call and can enable other rewrites the opportunity to optimize the **R**-expr. For example consider the `map` function below in figure 14-1 that applies a function to a list:

```
754 map(F, []) = [].
755 map(F, [Head|Tail]) =
756   [F(Head)|
757    map(F,Tail)].
```

(a) Dyna

```
map(F, L, Res) → (Res=only(Inp,
  (L=[])*(Inp=[])+
  proj(Head,proj(Tail,proj(Tmp1,proj(Tmp2,
    (L=[Head|Tail])*
    indirect_call(F, Head, Tmp1)*
    map(F, Tail, Tmp2)*
    (Inp=[Tmp1|Tmp2])))))))
```

(b) R-expr

Figure 14-1. A recursive function that makes an indirect call. Here the map function (lines 754 to 757) applies the function F to all elements in the list. The indirect call (line 756) is compiled into the indirect_call(F,···) **R**-expr.

The indirect_call **R**-expr makes the map function in figure 14-1 difficult to analyze and compile. Ideally, we want to eliminate all indirect_calls. In the case of the map function, this can accomplish by creating a *specialized* version as in figure 14-2.[218]

The remainder of this chapter is not intended to be a complete discussion about why one wants to fold programs or what kinds of programs folding can help solve. To find some discussion around why one wants to fold, how folding can solve certain kinds of programs, and how folding can be used to change the asymptotic runtime of a program, I suggest that the reader look into prior work such as Eisner and Blatz [58] and Vieira [141], both of which discuss folding on Dyna programs that have a single semiring. More generally, [22, 25, 80, 97] discuss folding on

[218]In this example, folding away the indirect_call is similar to using a template in C++ which allows function calls to be static (rather than require a runtime indirection through a function pointer).

```
times2(X) = X*2.
b(L) = map(
  times2[], L).
assert b([1,2,3])
    == [2,4,6].
```

(a) Dyna

```
b(L,Res) → (Res=only(Inp,map_times_2(L, Inp)))
proj(F, map(F,L,Res)*(F=times2[])) → map_times_2
    (L, Res)
map_times_2(L, Res) → (Res=only(Inp,
  (L=[])*(Inp=[])+
  proj(Head,proj(Tail,proj(Tmp1,proj(Tmp2,
    (L=[Head|Tail])*
    times(Head, 2, Tmp1)*
    map_times_2(Tail, Tmp2)*
    (Inp=[Tmp1|Tmp2])))))))
```

(b) R-expr with $\texttt{times}_2$ folded into the `map` **R**-expr.

Figure 14-2. Folding with `map` eliminates the `indirect_call` which can block other rewrites.

functional programming and logic programming.

14.2 How to Fold

Folding a program represented as a collection of **R**-exprs can be completed in a few steps.[219]

First, observe that folding is *only* useful in the case of a recursive function. Non-recursive functions can be expanded entirely. This means that we first identify if a user-defined **R**-expr is recursive, or depends on a recursive **R**-expr, and will target those user-defined **R**-exprs only. These functions are identified by constructing a call graph and identifying which user-defined **R**-exprs are depended on when

[219]Folding on **R**-expr was implemented on the **R**-expr prototype https://github.com/argolab/dyna-R, and has yet to be reimplemented on the Clojure implementation from chapter §11.

expanding user-defined rewrites (rewrite rule 74). For example, if we have a user-defined **R**-expr `f(⋯)`, which calls itself directly, then its rewrite rule will include a `f(⋯)` **R**-expr in the result of the rewrite:

$$\texttt{f}(\cdots) \xrightarrow{74} \cdots\texttt{*f}(\cdots)\texttt{*}\cdots$$

Or in the case of indirect calls such as with `g(⋯)` and `h(⋯)` that require computing the call graph:

$$\begin{aligned}\texttt{g}(\cdots) &\xrightarrow{74} \cdots\texttt{*h}(\cdots)\texttt{*}\cdots\\ \texttt{h}(\cdots) &\xrightarrow{74} \cdots\texttt{*g}(\cdots)\texttt{*}\cdots\end{aligned}$$

Once the system has identified a user-defined **R**-expr that is recursive, it unfolds the user-defined **R**-expr so that it has a single large **R**-expr that represents two or more steps of the recursive call. If the recursion happens indirectly (such as with `g(⋯)` and `h(⋯)`), then the system unfolds the user-defined **R**-exprs until it expands all the way to the recursive calls.

Once the recursive user-defined **R**-exprs has been identified, the system will use folding to attempt to create new user-defined **R**-exprs. These **R**-exprs are identified as a conjunction of **R**-exprs that are folded together. This is implemented by normalizing the variable names on the **R**-expr and then using the fact that **R**-exprs supports hashing and equality checks (chapter §8), and therefore can be used as keys in a hash-table.

If the newly folded **R**-expr is different from the existing user-defined **R**-exprs and previously folded **R**-exprs, it is added to the set of automatically defined **R**-exprs.

If folded **R**-expr already exists, then the existing **R**-expr is referenced rather than creating another identical named **R**-expr.

The newly created **R**-expr will contain a call to a recursive user-defined **R**-expr. Hence, it will be checked for additional folding opportunities within itself.

An example of this process is shown in the next section.

14.3 Example Using Folding to Solve a Recursive Program

To see how folding a user-defined **R**-expr works, let us work through an example of intersecting an even and odd length list in figure 14-3. Here, both `even_list` and `odd_list` are recursive functions. Independently, for these recursive functions, there is nothing that the system can do. However, once we take a *conjunction* of these on line 766, then there are rewrites that can identify that this term is impossible for all inputs provided we can work around the recursion using folding.

```
762 even_list([]).
763 even_list([X,Y|Tail]) :- even_list(Tail).
764 odd_list([X]).
765 odd_list([X,Y|Tail]) :- odd_list(Tail).
766 no_possible_list(List) :- even_list(List), odd_list(List).
```

Figure 14-3. Even and odd list length example. There is no list whose length is both even and odd, so the `no_possible_list` rule on line 766 as a null value for every list.

The **R**-expr for figure 14-3 is shown in figure 14-4:

```
even_list(L) → (true=exists(true,
  (L=[])+
  proj(X,proj(Y,proj(Tail,
    (L=[X,Y|Tail])*
    even_list(Tail))))))
odd_list(L) → (true=exists(true,
  proj(X, (L=[X]))+
  proj(X,proj(Y,proj(Tail,
    (L=[X,Y|Tail])*
    odd_list(Tail))))))
no_possible_list(List) → (true=exists(true,
  even_list(List)*odd_list(List)))
```

Figure 14-4. Even and Odd List translated into an **R**-expr. I am simplifying the presentation by omitting the `Result` variable as it would simply take on the value *true* at all times.

First, we can identify which user-defined **R**-exprs are recursive in this program. We will identify both `even_list`, `odd_list` as recursive and determine that `no_possible_list` depends on them. Therefore, all of these **R**-exprs are candidates for folding.

When the system tries folding `even_list` and `odd_list`, no interesting/new **R**-exprs are found. The resulting **R**-exprs are equivalent to the existing `even_list`, and `odd_list`, so nothing is done.[220]

However, when the system attempts to process `no_possible_list`, it unfolds both the `even_list` and `odd_list` and finds that it can use folding to create a new **R**-expr that represents the first step of `even_list` and `odd_list`, as shown in figure 14-5:

[220]If a fold ends up getting created, then it would be equivalent to the existing **R**-exprs, hence unproductive. An unproductive fold is not *invalid*, just not useful.

```
even_list(L)*odd_list(L) $\xrightarrow{95}$ even_and_odd_list(L)
even_and_odd_list(L) $\xrightarrow{96}$
  (true=exists(true,
    (L=[])+
    proj(X,proj(Y,proj(Tail,
      (L=[X,Y|Tail])*
      even_list(Tail))))))*
  (true=exists(true,
    proj(X, (L=[X]))+
    proj(X,proj(Y,proj(Tail,
      (L=[X,Y|Tail])*
      odd_list(Tail))))))
no_possible_list(List) $\rightarrow$ (true=exists(true,
  even_and_odd_list(List)))
```

Figure 14-5. The `no_possible_list` user-defined **R**-expr is defined in terms of the now folded `even_and_odd_list`. Observe that `even_and_odd_list` does not need to include aggregation at the top, like user-defined **R**-exprs that come from Dyna programs, but rather is a conjunction of two aggregations that come from the definition of `even_list` and `odd_list`. Further rewrites can be performed against the definition of rewrite rule 96.

The system can ten perform rewrites against rewrite rule 96, to determine that the `(L=[])` and `(L=[X])` are both impossible to satisfy. This is done by creating another **R**-expr where `(L=[])` is lifted to the of the **R**-expr, and then observing that the resulting **R**-expr rewrites as `0`.[221] This results in the **R**-exprs shown in figure 14-6.

[221] E.g. `(A=sum(X,R1+R2))*(B=sum(Y,S1+S2))` $\xrightarrow{97}$ `(A=sum(X,R2))*(B=sum(Y,S1+S2))` **if** $\forall_E [\![$`(A=sum(X,R1+R2))*(B=sum(Y,S1+S2))*proj(X,proj(A,R1))`$]\!] = 0$.

```
even_list(L)*odd_list(L) $\xrightarrow{95}$ even_and_odd_list(L)
even_and_odd_list(L) $\xrightarrow{96}$
  (true=exists(true,
    proj(X,proj(Y,proj(Tail,
      (L=[X,Y|Tail])*
      even_list(Tail))))))*
  (true=exists(true,
    proj(X,proj(Y,proj(Tail,
      (L=[X,Y|Tail])*
      odd_list(Tail))))))
no_possible_list(List) → (true=exists(true,
  even_and_odd_list(List)))
```

Figure 14-6. Analysis on rewrite rule 96 allows for `(L=[])` and `(L=[X])` to both be rewritten as `0`.

Now by rearranging rewrite rule 96, we can lift `even_list` and `odd_list` out of the aggregator as there are no other disjunctions present. This allows us to recognize that `even_list` and `odd_list` are conjuncts and will create an **R**-expr like figure 14-7.

```
even_list(L)*odd_list(L) $\xrightarrow{95}$ even_and_odd_list(L)
even_and_odd_list(L) $\xrightarrow{96}$
  (true=exists(true,
    proj(X,proj(Y,proj(Tail,
      (L=[X,Y|Tail])*
      even_list(Tail)*
      odd_list(Tail))))))
no_possible_list(List) → (true=exists(true,
  even_and_odd_list(List)))
```

Figure 14-7. `even_list` and `odd_list` are conjuncts inside of `even_and_odd_list`, rewrite rule 96.

Using rewrite rule 95, we can apply the fold to rewrite rule 96's definition, which will allow us to identify that this has already been defined, and it recursively calls itself.

```
even_list(L)*odd_list(L) $\xrightarrow{95}$ even_and_odd_list(L)
even_and_odd_list(L) $\xrightarrow{96}$
  (true=exists(true,
    proj(X,proj(Y,proj(Tail,
      (L=[X,Y|Tail])*
      even_and_odd_list(Tail))))))
no_possible_list(List) → (true=exists(true,
  even_and_odd_list(List)))
```

Figure 14-8. `even_list` and `odd_list` are folded together using rewrite rule 95, so we can now recognize that `even_and_odd_list` recurses to itself.

The same mechanism that identifies recursive functions also checks that all recursive functions have a base case. The reason is that recursive Dyna functions without a base case are equivalent to null for all values. Hence, we can rewrite rewrite rule 96 as `0` as shown in figure 14-9.

```
even_list(L)*odd_list(L) $\xrightarrow{95}$ even_and_odd_list(L)
even_and_odd_list(L) $\xrightarrow{96}$ 0
no_possible_list(List) → (true=exists(true,
  even_and_odd_list(List)))
```

Figure 14-9. `even_and_odd_list` is found to be impossible due to the recursion without a base case. Hence, it is rewritten as `0`

This sequence of rewrites and folding shows how the Dyna system can determine that there is no list that is simultaneously even and odd in length at the same time.

14.4 Folding Updatable User-Defined **R**-exprs

Folded **R**-expr depends on the original user-defined **R**-expr. As such, if a user-defined **R**-expr is modified (either by the API or by a user at the REPL), then all downstream folded **R**-exprs have to be invalidated and refolded.

For user-defined **R**-exprs with memos, we completely disable folding for that term. This is done by preventing user-defined **R**-exprs with memoization policies from unfolding when the system generates an **R**-expr to check for folding opportunities (disabling rewrite rule 74). This is because memoized **R**-exprs can include guesses, which cannot be bypassed (section §10.5). Furthermore, guesses are expected to change frequently during the running of the program as the system converges toward a final answer.

Chapter 15

Properties of Simplify

This chapter builds on the rewriting system defined in chapter §8. What we can prove about our rewriting system is unfortunately less interesting than what we might *like*—such is the reality of building a non-trivial programming language. Nevertheless, I claim the following properties about our rewrite system, and provide sketches for how these properties can be proven:

1. Soundness of Rewrites
2. Completeness on Datalog subset of Dyna
3. Incompleteness of Rewrites in general
4. Turing Completeness
5. Termination of SimplifyNormalize on bounded **R**-exprs

15.1 Is Simplify Sound and Complete?

Being *sound* and *complete* are desirable properties of a logical system. A system is **sound** if it never proves a false statement. In other words, all statements that are *proven* true are actually true. A system is **complete** if there exists a proof for all true statements. In the case of term rewriting procedure (such as we have in this dissertation), completeness may instead be defined in terms of the proofs that are *found* through the systematic application of rewrite rules (e.g. SIMPLIFY given in chapter §8), rather than claiming that there *exist* of a sequence of rewrites that can prove a statement.

When it comes to **R**-exprs, their semantics, and their rewrites, let us briefly discuss what it means for the rewrite system to *prove* a statement is true. Recall, the **R**-exprs semantics is defined in terms of the multiplicity that it assigns to an environment $E(\cdot)$ (chapter §5). As such, we will equate the idea of proving a statement as true with it having a non-zero multiplicity and is contained in the bag that the **R**-expr represents. Similarly, we will define false statements as those that have zero multiplicity and are not contained in the bag. To make this more concrete, we can say that for the rewrite system to prove an **R**-expr `R` that does not contain free variables ($\text{vars}(\mathtt{R}) = \emptyset$)[222] true, there must exist a sequence of rewrites such that `R`$\rightarrow$`*1+S`, for some `S`.[223] Similarly, for the system to prove an **R**-expr `R` false, there must exist a sequence of rewrites `R`$\rightarrow$`*0` for it to be proven false.

[222]Allowing free variables in the **R**-expr requires more work to concisely define this. For example, the **R**-expr `(X=1)` has a free variable `X`, but cannot rewrite as 1 or `0`. Hence, it is odd to think of this of `(X=1)` as proving `X`. Rather, by focusing on only the case without free variables, we do not have to define if `(X=1)` is *true* or *false*. This restriction is not an issue though, as one can simply project out all free variables, in which case it can be rewritten as a multiplicity. E.g. `proj(X,(X=1))`$\rightarrow$1.

[223]Writing the **R**-expr as `1+S` allows for this to be more general, in that it can match all multiplicities ≥ 1, and matches the rewrite rule for `if`-expressions (rewrite rule 63). Alternately, we could write this as `if(R,1,0)`, which would guarantee that this **R**-expr is rewritten as 1 or `0`.

15.1.1 Soundness of Simplify

We claim that our rewrite system is sound in that all **R**-exprs that are rewritten (and can be rewritten) into the form `1+S`, for some `S`, have a multiplicity greater than 1.

Proof: Our rewrite rules given in chapter §6 are semantics preserving with respect to the semantic interpretation given in chapter §5. Hence, any **R**-expr `R` that is rewritten into the form `1+S` must be semantically equivalent and therefore have a multiplicity greater than 1 and be considered true.[224] ■

15.1.2 R-exprs Rewrites are Incomplete

Our rewrite system is not complete. Given the definitions given previously, for a rewrite system to be complete, any true statement that can be expressed must be rewritable into the form `1+S`, for some `S`. In other words, there exists a **R**-expr `R` that is semantically equivalent to `1+S` for some `S`, but there are no rewrites that can rewrite `R` into the form `1+S`.

Proof: To prove that **R**-exprs are not complete, we can simply demonstrate an **R**-expr that is equivalent to `1`, but cannot be rewritten as `1`. Figure 15-1 shows an **R**-expr that encodes Fermat's last theorem, which was proven true in [148].

[224]The rewrites in chapter §6 have been checked by hand to match the semantic definitions in chapter §5. Furthermore, the rewrites in the implementations in chapters 8 and 11 have also been checked by hand. Checking rewrites and their implementation by hand does not *guarantee* that the rewrites are *bug-free*. Any rewrite, or implementation of a rewrite, that is not semantic preserving is a bug.

```
fermat_was_right →
  if(proj(A,proj(B,proj(C proj(N,proj(Aval,proj(BVal,proj(CVal,
      power(A, N, Aval)*power(B, N, BVal)*power(C, N, CVal)*
      add(AVal, BVal, CVal)*int(A)*int(B)*int(C)*int(N)*
      lessthan(2, N)*lessthan(0, A)*lessthan(0, B)*lessthan(0, C)
     ))))))), 0, 1).
```

Figure 15-1. R-exprs are sufficiently powerful to express general mathematical expressions such as Fermat's last theorem shown here (I.e., there does not exist positive integers a,b,c,n where $n \geq 3$ such that $a^n + b^n = c^n$). While this Fermat's theorem was proven true [148], hence this **R**-expr can be rewritten as 1, it is unrealistic to include the necessary rewrites to solve this **R**-expr.

Hence, the `fermat_was_right` **R**-expr in figure 15-1 is *semantically equivalent* to the **R**-expr 1. However, our rewrite system does not include any rewrites that can rewrite the `fermat_was_right` **R**-expr. (The proof of Fermat's last theorem requires mathematical properties that we currently do not have implemented for **R**-exprs). The fact that there exists an **R**-expr that cannot be rewritten shows that the rewrite system is incomplete. ■

Admittedly, users of the Dyna system are likely not expecting it to automatically prove Fermat's last theorem, hence this is not an informative result.

More generally, we can say that any sound rewriting system for **R**-exprs cannot be complete in the sense that any two semantically equivalent **R**-exprs can be non-deterministically rewritten into the same form. (In other words, for *some* pair of **R**-exprs R, R' with $[\![\texttt{R}]\!]_E = [\![\texttt{R'}]\!]_E$, there is no S such that $\texttt{R} \to^* \texttt{S}$ and $\texttt{R'} \to^* \texttt{S}$.) The reason is that **R**-exprs are powerful enough to express any first-order statement about arithmetic. Hence, **R**-exprs are subject to Gödel's incompleteness theorem, which ensures that there exists an **R**-expr that is a true, i.e., semantically equivalent to `1`, yet cannot be rewritten as `1`.

15.1.2.1 Completeness on Datalog Subset

While **R**-expr and their rewrites are incomplete, we can prove that there are some subsets of the Dyna language that are complete.

First, I will prove that the *Datalog* subset of Dyna is complete when using the appropriate memoization policy.

Proof: First, let us briefly recall how Datalog works, which was previously discussed in section §3.1.2. Datalog memoizes all terms that have been proven true. Rules in a Datalog program combine terms that have been proven to deduce new terms that are also true. Furthermore, for a program to be a valid Datalog program, it must be stratified when aggregation or negation is used. This means that we do not have to worry about cyclic programs, which require the guessing mechanism from section §10.5. Finally, Datalog terms only contain ground values (e.g. the term `foo[1,2]`, but not `bar[X,X]`). This means that all of the terms can be easily enumerated by enumerating the ground assignments to variables in the term.

To emulate the *memoize everything* behavior, we can use the `$memo` policy that returns `"null"` for all user-defined terms.[225] This works because we are only concerned with the Datalog subset, which only proves ground terms. This means that the memo tables only consist of disjunctions of ground assignments to the variables present on a term.

For example, if we have `$memo(foo[X,Y]) = "null".` as the memoization policy for a Datalog program. The `foo/2` memo will look something like `(X=1)*(Y=2)*(Result=true)+(X=7)*(Y=3)*(Result=true) +···+(X=11)*(Y="hello")*(Result=true)`, where all of the values to `X` and `Y` are known ground values, and therefore enumerable.[226]

Because Datalog rules combine the purely ground terms together by looping over the ground bindings to variables, we will exactly replicate this behavior using our standard execution procedure. Furthermore, Datalog programs are stratified[227] when it comes to aggregation and negation, which ensures that the set of terms that are true and memoized will converge. Hence, Dyna's execution exactly matches what happens in a typical Datalog implementation.

Because our execution exactly replicates Datalog (when given an appropriate memoization policy), we can claim that our implementation is complete in the

[225]This would be equivalent to defining `$memo(X) = "null".` However, due to limitations in the $memo control mechanism, each user-defined term must be specified with its own `$memo(·)` policy. Hence, this would require specifying multiple $memo policies like:
```
$memo(foo[X,Y]) = "null".
$memo(bar[X,Y,Z]) = "null".
```
[226]In fact, memos of this form will be efficiently handled by the prefix-tries (section §11.6.1) and iterators (section §11.7).

[227]There is nothing in Dyna that checks that a program is stratified and enforces stratification. Hence, I am assuming that the program is already *in* the Datalog subset and is correctly stratified, meaning it does not contain cycles that involve negation or aggregation.

same way that Datalog inference is complete. ■

15.1.2.2 Emulating SLD Resolution

Selective Linear Definite (SLD) resolution, as defined in Foundations of Logic Programming by Lloyd [101], is a technique to implement resolution on horn clauses.[228] SLD resolution performs unification between non-recursive structural terms that can contain variables. For example, if the terms `foo[X,bar[X,Y]]` and `foo[A,bar[B,B]]` are unified together, then the variable `X`, `Y` and `Z` will all be unified together (`X=Y=Z`, hence `foo[X,bar[X,X]]`). SLD resolution also includes an occurs check which ensures that terms recursive terms are identified as impossible. For example, the unification between `f[X,X]` and `f[Y,g[Y]]` would fail as this would result in `Y=g[Y]`, and there is no value of `Y` that will make this expression true.

Horn clauses[228] that are true can be forward chained to identify all terms that are true. This is the same sort of forward chaining that we have in a Datalog program without negation or aggregation. As such, SLD resolution is known to be sound and complete for horn clauses [101].

To see an example of SLD resolution, consider the following program.

```
a(f[X]).
a(g[I,J]).
b(Y,Z) :- a(Y).
c(W) :- c(W).
```

[228] Horn clauses are sometimes referred to as Pure-Prolog. The term Pure-Prolog is not used consistently across all publications, so I will avoid the term Pure-Prolog.

First SLD resolution will deduce the terms `a[f[X]]` and `a[g[I,J]]` are true from lines 767 and 768 as there are no conditions for these horn clauses. SLD resolution will then use `a[f[X]]` to forward chain into the rule on line 769, where the variable `Y` is unified with the structure `f[X]`. This causes SLD resolution to deduce the term `b[f[X],Z]` is true. Next, SLD resolution forward chains `a[g[I,J]]`, which causes `b[g[I,J],Z]` to be deduced as true as well. Now the terms that have been proven true are `a[f[X]]`, `a[g[I,J]]`, `b[f[X],Z]`, and `b[g[I,J],Z]`. SLD resolution will not stop running as it has reached a fixed point where all possible terms have been deduced as true. Note that `c[W]` is never deduced as true, as there is no base case for `c[W]` to get it started.

Emulating SLD with R-exprs: SLD resolution can be emulated using **R**-exprs and our rewrites. However, it requires a slightly different *control* strategy from the one discussed in chapter §8.

First, observe that we already have the necessary rewrites for unification and the occurs check (section §6.1.1). Second, our memo table is capable of memoizing structural terms with variables like `foo[X,bar[X,Y]]` by memoizing an **R**-expr containing structural unifications.

However, there is a slight difference from what SLD memoizes and uses to forward chain. Observe that in the previous SLD example, the terms `a[f[X]]` and `a[g[I,J]]` were each memoized individually and forward chained individually. Essentially, the disjunction between the different ways that `X` can be satisfied in `a(X)` is handled directly by the SLD algorithm. In the case of **R**-exprs, we instead would create a single memo for all of `a(X)`, and then use the disjunction (section §5.2.2.5) to represent the different ways that `X` can be satisfied. In other words, when evaluating line 769 and expanding `a(Y)` using **R**-exprs, we are going to have an **R**-expr like `b(Y,Z)→(true=exist(true, proj(X,(Y=f[X]))+ proj(I,proj(J,(Y=g[I,J])))))`. In other words, because of our desire to make *factored* **R**-exprs to work around aggregation (section §8.2.3.1), we do not end up expanding out the terms entirely as SLD resolution does.

To emulate SLD resolution using **R**-exprs. We could instead consider a system that uses the distributive rewrite to expand **R**-exprs rather than factor them (e.g. $\texttt{Q*(R+S)} \xrightarrow{22} \texttt{Q*R + Q*S}$). In this case, we will have that each **R**-expr would only represent a single conjunctive structural term with variables rather than the factored form that we currently have.

As a secondary issue, we also have aggregation in the program that needs to be removed before the distributive rewrite can be used to fully expand the **R**-expr. In principle, nested `exists` aggregators could be removed.[229] E.g. $\texttt{(true=exists(true, Q*R*(true=exists(true, S1+S2))))} \xrightarrow{98} \texttt{(true=exists(true, Q*R*(S1+S2)))}$.[230]

In conclusion, if aggregators were removed from the **R**-expr, and the distributive

[229]The `(true=exists(true,R))` aggregator essentially forces the multiplicity of `R` to 1. Hence, as long as there is an `exists` aggregator at the top of the **R**-expr, the *entire* **R**-expr will have the same semantics.
[230]Currently rewrite rule 98 is not included in the implementation.

rewrite was used to expand **R**-exprs, rather than factor, then our memoization mechanism is sufficiently powerful to memoize representations of structured terms with variables, and our unification rewrites are the same as those used by SLD resolution. The system does not perform rewrites in this way because of our desire to work around aggregation (a frequent operation in Dyna programs).

15.2 Dyna is Turing Complete

Dyna is Turing complete. This should not be surprising as Prolog is known to be Turing complete, and Dyna is a superset of Prolog. Proving Dyna is Turing complete is useful as it shows that all computable functions *can* be computed using Dyna using *some* program. However, being Turing complete does not mean that *all* ways in which a function can be represented will result in Dyna being able to compute using the result of the function. Hence, showing that Dyna is Turing complete does not contradict the previous proof that **R**-exprs and rewrites on **R**-exprs are incomplete.

Proof: Proving Turing completeness can be done by reducing[231] from a known universal Turing machine into Dyna. I have chosen to use Rule-110[232], as it has been proven to be Turing complete [40][233] and is considered one of the simplest known Turing complete systems. The implementation of Rule-110 in Dyna is shown

[231]This kind of reduction is the same kind of reductions that one sees with NP-complete proofs using a reduction from SAT.
[232]`https://en.wikipedia.org/wiki/Rule_110`
[233]The proof that Rule-110 is Turing complete requires several very dense math papers.

on lines 771 to 786:

```
771 turing_machine_step([1,1,1|X]) = [0|turing_machine_step([1,1|X])].
772 turing_machine_step([1,1,0|X]) = [1|turing_machine_step([1,0|X])].
773 turing_machine_step([1,0,1|X]) = [1|turing_machine_step([0,1|X])].
774 turing_machine_step([1,0,0|X]) = [0|turing_machine_step([0,0|X])].
775 turing_machine_step([0,1,1|X]) = [1|turing_machine_step([1,1|X])].
776 turing_machine_step([0,1,0|X]) = [1|turing_machine_step([1,0|X])].
777 turing_machine_step([0,0,1|X]) = [1|turing_machine_step([0,1|X])].
778 turing_machine_step([0,0,0|X]) = [0|turing_machine_step([0,0|X])].
779 turing_machine_step([A,B]) = [A,B].
780 turing_machine_step([A]) = [A].
781 turing_machine_step([]) = [].
782
783 turing_machine(X) := turing_machine(turing_machine_step(X)).
784 turing_machine(X) := X for X = turing_machine_step(X).  % Turing
       machine has reached a fixed point
785
786 final_output = turing_machine(start_state).
```

■

Rule-110 is a cellular automaton that uses a one-dimensional array consisting of 0s and 1s. The array is progressively rewritten using a pattern of 3 bits and then shifting over in the array by 1 position. Rule 110 is applied repeatedly using line 783 until it reaches a final state where the Turing machine stops changing, which is identified by line 784.

Observe that this program depends on recursion (line 783). Recursion is the only way in which a programmer can represent an unbounded computation (such as a Turing machine), which I will prove next.

15.3 Termination of SimplifyNormalize on a Bounded Size **R**-expr

A potential complication when designing a rewrite rule system is that the rewrite rules could oscillate unproductively, causing a terminating program to become non-terminating. We claim that all terminating programs will terminate when rewritten by SimplifyNormalize, and that the resulting **R**-expr will be "*simple*", according to some definition of "simple". This ensures that 1) we are doing *something* useful when rewriting (hence the result will be simple), and 2) that we are not accidentally causing non-termination.[234]

First, the only way one can write a Dyna program that does not terminate is either through the use of a bad memoization policy (chapter §10 and section §2.5), or through the use of recursion to write an unbounded loop. The function SimplifyNormalize is not directly involved with memoization; hence, memoization need not be considered in this proof. Therefore, for this proof to apply, we only need to ensure that there is no unbounded recursion, which means that the **R**-expr being rewritten by SimplifyNormalize is bounded in size.

[234]This proof does not prove that this will terminate with the correct runtime, or that it will run "fast".

15.3.1 Making an **R**-expr Bounded in Size

In general, **R**-exprs are not bounded in size. Hence, to make the proof in the next section go through, we first need to ensure that the **R**-expr that is being rewritten is bounded in size so that the proof will apply. To do this, we “modify” the **R**-expr to ensure that the **R**-expr is bounded in size.

The way this is done is by tracking the *stack depth*[235] of user-defined rewrites. This is done by modifying the presentation from section §6.7 so that there is a stack depth counter. Here, I denote the stack depth as a superscript on the user-defined **R**-expr types and modify rewrite rule 74 so that the counter is incremented for *all* internal user-defined **R**-exprs.

$$\mathtt{f}^k(\mathtt{X}_1,\ldots,\mathtt{X}_n) \xrightarrow{74} \cdots{*}\mathtt{f}^{k+1}(\cdots){*}\cdots{*}\mathtt{g}^{k+1}(\cdots){*}\cdots \qquad \textbf{if } k < K$$

Now the rewrite rule 74 rule *only* runs if the stack depth is less than some max-stack-depth, denoted here as K.

The approach of bounding the number of expansions of recursive user-defined **R**-exprs is not without consequences. For example, if we set the stack limit too low, then we can end up with **R**-exprs that are not useful to the user. Figure 15-2 shows one such instance, where the factorial program stops expanding before reaching its base case. The **R**-expr that would be returned to the user’s query would be partially expanded but does not return the desired numerical result of factorial.

[235]We do not really have a call stack in the traditional sense, as we are expanding the **R**-expr out using rewrite rule 74, but we can think of the stack depth as corresponding to the depth at which a given call would appear if the program was executed under a traditional programming language.

```
787 factorial(X) :=
788   factorial(X-1)*X.
789 factorial(0) := 1.
```

(a) Dyna

```
proj(Tmp1,
  times(100,Tmp1,Result)*
  proj(Tmp2,
    times(99,Tmp2,Tmp1)*
    proj(Tmp3,
      times(98,Tmp3,Tmp2)*
      factorial⁴(97, Tmp3))))
```

(b) The `factorial` **R**-expr expanded up to $K = 4$ after some rewrites are applied.

Figure 15-2. Recursive Dyna program which requires many steps of recursion to get a "useful" result. If the program's expansion is cut short, with $K = 4$, then the resulting **R**-expr is still *correct*, though not useful as in (b).

```
790 f(X) := f(X-1)*f(X-1)*f(X-1).
791 f(0) := 2.
```

Figure 15-3. Assuming that no rewrites are done to combine `f(X-1)` on line 790,[236]then this program will expand to an exponential 3^K-sized **R**-expr. Hence, the *stack depth* does not mean *small* program, or *limited memory* etc.

Similarly, limiting the stack depth does not necessarily result in a small **R**-expr or a limited amount of memory consumed (as would commonly happen in a procedural stack-based programming language). For example, figure 15-3 shows a Dyna program that would create an exponential 3^K-sized **R**-expr when expanded to depth K.

[236]For example, `f(X-1)*f(X-1)*f(X-1)` could rewritten into `Y=f(X-1), Y*Y*Y`. In which case `f(X-1)` would only be called once, avoiding the exponential-sized expansion.

15.3.2 Outline: How to Prove Termination

To prove that SimplifyNormalize terminates and Simplify reaches a fixed-point, we prove that there cannot be an infinite sequence of directional rewrites on a bounded size **R**-expr. To do this, we define a set of *energies* $\mathbb{E}$ such that there is no infinite descending sequence, and we define an energy function $|\cdot| : \mathbf{R} \to \mathbb{E}$ such that

$$|\textsc{Simplify}(\mathsf{R})| \leq |\mathsf{R}|$$

and

$$(\textsc{Simplify}(\mathsf{R}) \neq \mathsf{R}) \implies (|\textsc{Simplify}(\mathsf{R})| < |\mathsf{R}|)\,.$$

Together, these properties mean that Simplify will either return the **R**-expr *unmodified*, meaning no rewrites were applied, or if it returns a different **R**-expr, where there were one or more rewrites applied to, it will have decreased the energy. Because there is no infinite descending sequence in $\mathbb{E}$, this means that repeated applications of Simplify must reach a fixed point in a finite number of steps.

What is left to complete this proof is to define $|\cdot| : \mathbf{R} \to \mathbb{E}$ and show that the rewrites defined in chapter §6 will individually decrease the energy of the **R**-expr.

This proof is structured as follows. First, I will define $|\cdot|_{\text{core}} : \mathbf{R} \to \mathbb{N}_{\geq 1}$ that is the energy for all **R**-expr types *excluding* built-in **R**-exprs (section §5.2.2.3). Second, in section §15.3.4, I will define a notion of energy specifically for built-ins. Finally in section §15.3.5, I will define $|\cdot| : \mathbf{R} \to \mathbb{E}$ which applies to all **R**-exprs.

15.3.3 Construction of "Core" Energy

First, we define $|\cdot|_{\text{core}} : \mathbf{R} \to \mathbb{N}_{\geq 1}$ as the energy for all **R**-exprs except for built-in **R**-expr types (section §5.2.2.3). The reason built-ins are excluded for now is that the rewrites on built-ins can be extended. This can be done by defining new rewrite rules or by incorporating existing software libraries to solve conjunctions of built-in constraints (such as a linear programming solver or an SMT solver, section §16.8).

Non-recursive: Non-recursive **R**-exprs have energies defined as follows:

1. $|\mathtt{M}|_{\text{core}} = 1$ — Multiplicities are a base case of the **R**-expr language as they cannot be rewritten further. They take on the smallest possible energy regardless of the value of $\mathtt{M} \in \mathcal{M}$.

2. $|\mathtt{(X=G)}|_{\text{core}} = 2$ where $\mathtt{G} \in \mathcal{G}$ — Unification of a variable with a *constant* has energy 2 regardless of the value of $\mathtt{G}$. In the case where the unification with a constant is redundant, it is rewritten as a multiplicity of 1, with an energy of 1. Hence, this decreases the energy.

3. $|\mathtt{(X=Y)}|_{\text{core}} = 3 + \begin{cases} 0 & \mathtt{X} \prec \mathtt{Y} \\ 1 & \mathtt{X} \succ \mathtt{Y} \end{cases}$ where $\mathtt{X}, \mathtt{Y} \in \tilde{\mathcal{V}}$ — With two variables in a unification **R**-expr. One of the variables could is bound with a constant value, the energy will decrease. For example, $|\mathtt{(X=Y)}|_{\text{core}} > |\mathtt{(X=7)}|_{\text{core}}$, where this rewrite could presumably happen as a result of some other **R**-expr determining the value of $\mathtt{Y}$. E.g. $\mathtt{(Y=7)*(X=Y)} \xrightarrow{5} \mathtt{(Y=7)*(X=7)}$. To define the energy in the case

that variables are reordered, there is an arbitrarily chosen order $\prec$ which defines what variable should appear first in the unification **R**-expr.

4. $|\texttt{(X=f[U}_1\texttt{,}\cdots\texttt{,U}_n\texttt{])}|_{\text{core}} = 3m+3$ — Unification with a structured term that contains variables is assigned $3m+3$ where m is the number of free variables in the **R**-expr. The reason we need to track the number of variables is that propagating a value into these **R**-exprs must decrease the energy of the overall **R**-expr (rewrite rule 5). For example: `(X=9)*(Y=f[X,Z])` $\rightarrow$ `(X=9)*(Y=f[9,Z])`.

Recursive: Recursive **R**-exprs have energies defined as follows:

5. $|\texttt{R*S}|_{\text{core}}=|\texttt{R}|_{\text{core}} + |\texttt{S}|_{\text{core}}$ — Conjunctive **R**-exprs energies is defined as the sum of their sub-**R**-exprs. This means that any reduction in the energy of `R` or `S` will reduce the energy of the overall **R**-expr.

   Furthermore, given that all **R**-exprs have a strictly positive energy, this means that removing a conjunct and unneeded constraints will reduce the energy. For example, rewrite rule 12 will multiply multiplicities, e.g. $\texttt{2*3} \xrightarrow{12} \texttt{6}$, which takes this **R**-expr from an energy of 2 to 1.

6. $|\texttt{R}_1\texttt{+R}_2\texttt{+}\cdots\texttt{+R}_n|_{\text{core}}= 2^{|\texttt{R}_1\ +\ \texttt{R}_2\ +\ \cdots\ +\ \texttt{R}_n|_{\text{core}}}$ — Disjunctive **R**-exprs are raised to a power of 2. This ensures that *nested* **R**-exprs are more expensive than **R**-exprs which are not nested. Hence, the energy of an **R**-expr is reduced when common sub-**R**-exprs are *factored* out.

Note that the presentation of disjunction presented here assumes a n-arity disjunction and not a recursively nested binary disjunction. However, the definition can be binarized by matching against the nested sub-**R**-exprs.[237]

7. $|\texttt{proj(X, R)}|_{\text{core}} = 2^{|\texttt{R}|_{\text{core}}}$ — Projected **R**-exprs are also nested. During simplification, we attempt to factor out expressions as much as possible; this means that more deeply nested sub-**R**-exprs will have *higher* energy. For example: $|\texttt{proj(X, R*S)}|_{\text{core}} > |\texttt{R*proj(X, S)}|_{\text{core}}$.

   Additionally, note that unlike structural unification (item 4), we do not have to check if `X` is a variable or a ground value. The reason is that if $\texttt{X} \in \mathcal{G}$, then we can eliminate the projection.

8. $|\texttt{if(Q, R, S)}|_{\text{core}} = |\texttt{Q}|_{\text{core}} + |\texttt{R}|_{\text{core}} + |\texttt{S}|_{\text{core}}$ — `if`-expressions have an energy which is the sum of the nested expressions. The `if`-expressions are typically handled by first rewriting `Q` until it can be determined if it is rewritten as `0` or as a non-zero multiplicity. Once the `if`-expression is determined to be true or false, it is rewritten as either `R` or `S` (rewrite rules 63 and 64). Thus, the energy of this expression needs to be higher than `R` or `S`. Additionally, rewrites can be performed directly on `R` or `S` for the purposes of making the **R**-expr more

[237]The binarised definition for disjunctions can be written as:

$$|\texttt{R} + \texttt{S}|_{\text{core}} = 2^{\wedge}\left(\begin{cases} \log_2 |\texttt{R}|_{\text{core}} & \textbf{if } \texttt{R} \textbf{ matches } \texttt{A+B} \\ |\texttt{R}|_{\text{core}} & \textbf{otherwise} \end{cases} + \begin{cases} \log_2 |\texttt{S}|_{\text{core}} & \textbf{if } \texttt{S} \textbf{ matches } \texttt{A+B} \\ |\texttt{S}|_{\text{core}} & \textbf{otherwise} \end{cases}\right)$$

This definition works by matching if the nested expression is *also* a disjunction and undoing the $2^{|\texttt{R}|_{\text{core}}}$ operation via $\log_2$.

efficient (e.g., as in the case of memoization chapter §10). When this happens, the energy of the `if`-expression needs to still decrease, regardless if we rewrite R or S.[238]

<u>Bidirectional Rewrites:</u> Observe that conjunctions, disjunctions, and projections have bidirectional **R**-exprs that are *unproductive*. By *unproductive*, I mean that a rewrite is only used to rearrange the **R**-expr, and does not result in any meaningful change or decrease in energy. For example, the commutativity rewrites on conjunctions and disjunctions swap the order in which two sub-**R**-exprs are represented: e.g. $\texttt{R*S} \overset{19}{\longleftrightarrow} \texttt{S*R}$. For these rewrite rules 18 to 21 and 40, we *do not* actually use them and instead depend on the context $\mathcal{C}$ to identify the necessary conjunctions between **R**-exprs when rewriting; therefore, we do not rearrange the **R**-expr explicitly.

Some bidirectional rewrite rules are useful, though. For example, the distributivity rewrite can create factored **R**-exprs from a disjunction ($\texttt{R*S+R*Q} \xrightarrow{22} \texttt{R*(S+Q)}$). These kinds of bidirectional rewrites are applied *explicitly* in a uni-directional manner to create more factored **R**-exprs and used *implicitly* via the context $\mathcal{C}$ to handle the "expanding out" direction of the rewrite (rewrite rules 22, 38 and 39, also recall section §8.2.3.1).

<u>Aggregation:</u> Aggregation requires a bit more care in how we define en-

[238]For the `if`-expression, an energy of $|\texttt{Q}|_{\text{core}} + \max(|\texttt{R}|_{\text{core}}, |\texttt{S}|_{\text{core}})$ would not work, as if we have $|\texttt{R}|_{\text{core}} > |\texttt{S}|_{\text{core}}$, and rewrites are performed on S, then the energy of the `if`-expression would not decrease.

ergy. Recall that we want to split the aggregation into smaller **R**-exprs when it is an aggregation over a disjunction (e.g. `(A=sum(X,R+S))`), so that each disjunct can be aggregated independently (rewrite rule 50). To accomplish this, we had to introduce aggregators that have a disjunction with `(X=agg_null)` to suppress the behavior of rewrite rule 48 which would cause it to be rewritten as `0` (e.g. `(A=sum(X,(X=agg_null)+R))`, see section §6.5.1 for the complete discussion). The reason we chose this representation for the aggregator **R**-expr is that it makes it clear what the required behavior is from an aggregator when handing a disjunction. However, this `(X=agg_null)` does *not* behave like a disjunction but rather a different kind of behavior of the aggregator. Hence, for the purposes of defining the energy of an aggregator, we are going to include `(X=agg_null)` as part of the aggregator instead of as a nested disjunction.[239]

9. $\lvert$`(A=sum(X,R))`$\rvert_{\text{core}} = \lvert$`(A=sum(X,(X=agg_null)+R))*not_equal(A,agg_null)`$\rvert_{\text{core}}$ $+1$ **if** (`R` **not match** `(X=agg_null)+S`) — The "normal" aggregator is defined in terms of `(A=sum(X,(X=agg_null)+R))`, which contains the `(X=agg_null)` disjunction, which suppresses rewrite rule 48. The reason for this energy definition is that we can think of this as handling the first part of rewrite rule 50, where a constraint `not_equal(A,agg_null)` is added to the **R**-expr to ensure that the case where `R` is `0` is handled correctly. This looks something like rewrite rule 99,

[239]In the implementation, `(X=agg_null)` is tracked as a boolean flag on the aggregator class instead of as a disjunction in the aggregator's body. See footnote 94 in section §6.5.1.

and we can observe that this rewrite decreases the energy of the **R**-expr.

$\texttt{(A=sum(X,R+S))} \xrightarrow{99} \texttt{(A=sum(X,(X=agg\_null)+R+S))*not\_equal(X,agg\_null)}$

10. $\big|\texttt{(A=sum(X,(X=agg\_null)+R))}\big|_{\text{core}} = {}^{(2\cdot|\texttt{R}|_{\text{core}})}2$, where the left superscript denotes tetration[240] — Aggregation with `(X=agg_null)` follows the same basic idea of projection and disjunction in that more deeply nested sub-**R**-exprs will have a higher energy. As such, "solving" an aggregator, either by determining the value returned by an aggregator (rewrite rules 49, 51, 55 and 56) or splitting an aggregator into aggregations over small **R**-exprs (rewrite rules 50 and 53) will decrease the energy of the **R**-expr. In this case, we only have to concern ourselves with rewrite rule 53, as the other rewrites for aggregators either factor an **R**-expr out, which will obviously decrease the energy, or solve the aggregator without having to deal with any intermediate states.

$$\begin{aligned}\texttt{A=sum(X, (X=agg\_null)+R+S)} \xrightarrow{53} \;& \texttt{proj(B,proj(C,} \\ &\quad \texttt{(B=sum(X,(X=agg\_null)+R))} \\ &\quad \texttt{* (C=sum(X,(X=agg\_null)+S))} \\ &\quad \texttt{* plus(B,C,A)))} \\ &\textbf{if } \texttt{R}\neq\texttt{(X=agg\_null)} \textbf{ and } \texttt{S}\neq\texttt{(X=agg\_null)}\end{aligned}$$

Looking at rewrite rule 53, we can observe that there are two nested projections and that there are two nested aggregators. In other words, we need:

$$\left|\begin{array}{l}\texttt{(A=sum(X,}\\ \texttt{(X=agg\_null)}\\ \texttt{+R+S))}\end{array}\right|_{\text{core}} > 2^\wedge\left(2^\wedge\left(\begin{array}{l}\big|\texttt{(B=sum(X,(X=agg\_null)+R))}\big|_{\text{core}}+\\ \big|\texttt{(C=sum(X,(X=agg\_null)+S))}\big|_{\text{core}}+\\ \big|\texttt{plus(B,C,A)}\big|_{\text{core}}\end{array}\right)\right).$$

[240]For example, ${}^3a = a^{a^a}$. The notation and the term **tetration** are due to Rucker [115]:
$${}^n a \overset{\text{def}}{=} \begin{cases} 1 & \textbf{if } n=0 \\ a^{({}^{n-1}a)} & \textbf{otherwise}\end{cases}$$

This is what the energy will be once we expand rewrite rule 53 with two additional projections.

To make this work, first observe for a n-way splittable disjunction that

$$|\mathtt{R}_1 \ + \ \mathtt{R}_2 \ + \ \cdots \ + \ \mathtt{R}_n|_{\text{core}} \geq 2^n ,$$

by the definition of the disjunction's energy. Therefore, the energy of a disjunction is an upper bound on the number of disjunctive branches that will need to be handled by the aggregator. In other words, the system could potentially introduce up to n projections and intermediate variables to handle aggregation over the disjuncts $\mathtt{R}_1$ to $\mathtt{R}_n$. Therefore, the energy of the aggregator's body is an upper bound[241] on the number of times that rewrite rule 53 can be applied, and the number of projections that will be introduced. To handle this, we use *tetration*[240] which takes a number to a power multiple times. In performing rewrite rule 53, other **R**-exprs are added in (such as `plus(B,C,A)` above), those are covered by multiplying $|\mathtt{R}|$ by 2, which gives the whole energy for the aggregator as ${}^{(2\cdot|\mathtt{R}|_{\text{core}})}2$.

User-defined R-exprs

The user-defined **R**-expr that guarantees termination has a stack depth limit K (introduced in section §15.3.1) that limits the number of times that a user-defined

[241]This is *very loose* upper bound.

R-expr will be expanded. This limited K allows us to define the energy of user-defined **R**-exprs as a finite number.

11. $\left|\mathsf{f}^k(\mathsf{X}_1,\ldots,\mathsf{X}_n)\right|_{\text{core}} = m+1+\begin{cases} \left|\mathsf{R}_f^{k+1}\right|_{\text{core}} & \textbf{if } k < K \\ 0 & \textbf{otherwise} \end{cases}$ — User-defined **R**-exprs' energy is defined both in terms of the number of free variables m and in terms of the energy of the expanded **R**-expr R_f^{k+1}. The reason for tracking the number of free variables is the same as with structured terms, if we propagate an assignment to a variable, we need to be able to record in the energy that some useful work was done in rewriting the **R**-expr.

    The recursive part of the energy checks the stack depth k of the user's **R**-expr. When $k = K$, this means that the user's **R**-expr is expanded as far as it will be expanded. Therefore, rewrite rule 74 will no longer run. We can now break the recursive definition of energy and replace that part with the value 0.

15.3.4 Energy for Built-in R-exprs

To define the energy for built-ins, we construct a "pluggable" definition of energy that is dependent on the rewrites provided on the built-ins.

To do this, we introduce the function $\mathcal{E}(\cdot) : \wr \mathsf{R} \int \to \mathbb{N}$ which is defined by the library of rewrites included in our system, implying that the rewrites on built-ins will also terminate.[242] The function $\mathcal{E}(\cdot)$ maps from a *bag*—not an **R**-expr—of

[242]As an example if we add an external linear programming solver as a library, there would a $\mathcal{E}(\cdot)$ function that is associated with the linear programming library.

conjunctive, *non-recursive* **R**-exprs to a *finite* natural number that is the upper bound on the number of rewriting steps which can be performed against that bag of constraints.

The way that we use $\mathcal{E}(\cdot)$ is that we will gather built-ins from the **R**-expr and use $\mathcal{E}(\cdot)$ to determine what *might* happen. For example, if we start with the **R**-expr `(X=1)*plus(X,2,Z)+lessthan(Z,W)`, then it is converted into the bag

$$\wr \texttt{(X=1)}@1, \texttt{plus(X,2,Z)}@1, \texttt{lessthan(Z,W)}@1 \int, \tag{15.1}$$

and passed to the function $\mathcal{E}(\cdot)$. In this case, the function $\mathcal{E}(\cdot)$ returns that the energy is 3. This follows from the longest sequence of rewrites being:

$$\texttt{(X=1)*plus(X,2,Z)*lessthan(Z,W)} \xrightarrow{5} \tag{15.2}$$

$$\texttt{(X=1)*plus(1,2,Z)*lessthan(Z,W)} \xrightarrow{28} \tag{15.3}$$

$$\texttt{(X=1)*(Z=3)*lessthan(Z,W)} \xrightarrow{5} \tag{15.4}$$

$$\texttt{(X=1)*(Z=3)*lessthan(3,W).} \tag{15.5}$$

Note that $\mathcal{E}(\cdot)$ operates on the *bag* of constraints, not an **R**-expr (which contains disjunction, conjunctions, projections, and aggregations). As such, if we have an **R**-expr with a disjunction like `(X=1)+plus(X,2,Z)+lessthan(Z,W)`, then the bag of constraints contained in the **R**-expr is *exactly the same* as equation (15.1) above.

Additionally, we define that all built-ins will have an energy of 1 under the *core* energy definition. E.g. $|\texttt{lessthan(X,Y)}|_{\text{core}} = 1$.

Why we need $\mathcal{E}(\cdot)$

As stated above, we are constructing a proof of termination *independent* of the built-in rewrites included. The reason for this is two-fold. First, we do not know the "complete set" of all built-in rewrites, and second, proving termination of built-in requires theories *outside* of what we are proving here. Hence, we represent the number of potential steps needed to rewrite a bag of built-ins using the $\mathcal{E}(\cdot)$ function.

To see this, consider rewrite rule 34 which combines two `lessthan` constraints together to infer new `lessthan` constraints:

$$\texttt{lessthan(A,B)*lessthan(B,C)*lessthan(C,D)} \xrightarrow{34} \tag{15.6}$$
$$\begin{pmatrix} \texttt{lessthan(A,B)*lessthan(B,C)*lessthan(C,D)*} \\ \texttt{lessthan(A,C)} \end{pmatrix} \xrightarrow{34} \tag{15.7}$$
$$\begin{pmatrix} \texttt{lessthan(A,B)*lessthan(B,C)*lessthan(C,D)*} \\ \texttt{lessthan(A,C)*lessthan(B,D)} \end{pmatrix} \xrightarrow{34} \tag{15.8}$$
$$\begin{pmatrix} \texttt{lessthan(A,B)*lessthan(B,C)*lessthan(C,D)*} \\ \texttt{lessthan(A,C)*lessthan(B,D)*lessthan(A,D)} \end{pmatrix} \tag{15.9}$$

Proving termination of rewrite rule 34 requires a theory about `lessthan` and how many `lessthan` constraints can be inferred. Furthermore, we need to be able to "understand" the **R**-expr in that the energy of equation (15.6) must be *greater* than the energy of equations (15.7), (15.8) and (15.9) despite the fact that equation (15.6) has the *smallest* **R**-expr representation. Hence, energy is not a purely syntactic property of the **R**-expr. This complexity for built-ins is punted into $\mathcal{E}(\cdot)$.

15.3.5 Energy for all **R**-exprs

Given $|\cdot|_{\text{core}}$ as the energy for the core **R**-exprs and $\mathcal{E}(\cdot)$ as the energy for built-in **R**-exprs, we need to construct a single definition of the energy $|\cdot|$ that can be used for *all* **R**-exprs at the same time.

To do this, we first need to convert from $\mathcal{E}(\cdot)$, which operates on *bags of non-recursive* **R***-exprs*, into a representation that operates over **R**-exprs that includes disjunctions and aggregation.

To accomplish this, observe that in constructing the energy for an **R**-expr, we only need *some* upper bound—regardless of how loose the upper bound is. Furthermore, identifying which sub-**R**-exprs interact with each other is just as difficult as running Simplify itself. Hence, it would be somewhat complex to determine the interaction between built-ins. Instead, we will construct an *upper bound* by considering *all* possible interactions which *might* happen between built-in. This can be done by summing the $\mathcal{E}(\cdot)$ function over all *power sets* of the sub-**R**-exprs that are in a given **R**-expr.

A **power set** $\mathcal{P}(x)$ is defined as a set of all subsets, or in this case sub-bags of x. For example if we have the bag $\wr a@2, b@1 \int$, the power set is:

$$\mathcal{P}\left(\wr a@2, b@1 \int\right) = \left\{\wr\int, \wr a@1\int, \wr a@2\int, \wr b@1\int, \wr a@1, b@1\int, \wr a@2, b@1\int\right\}$$

Observe that the power set will contain the bags that correspond to the *true* interaction between the built-in **R**-exprs. Hence, we can be sure that $\sum_{p \in \mathcal{P}(\mathsf{R})} \mathcal{E}(p)$

is an upper bound on the energy of the built-in **R**-exprs.

All together, we can now define the energy of an **R**-expr R_k as the ordered pair $\mathbb{E} = \mathbb{N}^2_{\geq 1}$:

$$|\mathsf{R}_k| \overset{\text{def}}{=} \left\langle \min_{s=0,1,2,\ldots,k} \left(\left(\sum_{p \in \mathcal{P}(\mathsf{R}_s)} \mathcal{E}(p) \right) - (k-s) \right), |\mathsf{R}_k|_{\text{core}} \right\rangle$$

where $\mathbb{E}$ lexicographical ordered (equation 15.10 below).

The subscript k represents the number of times that built-ins have been rewritten. In other words, R_0 is the **R**-expr before *any* rewrites have been applied, and R_1 is the **R**-expr after one rewrite on the built-ins, and so on up to the "current" **R**-expr R_k. The reason for this "*dependency on the previous* ***R****-exprs*" energy is that new **R**-exprs can be inferred, and the size of the power set will increase—as there are more **R**-exprs contained in the power set. By using *min* on all power sets up to R_k, we ensure that the first value in the pair does not increase when the size of the power set increases.

The second number in this pair is $|\mathsf{R}|_{\text{core}}$ as previously defined (section §15.3.3). This value will decrease for all rewrites that do not modify any of the built-in **R**-exprs, such as rearranging the **R**-expr or creating a more factored **R**-expr. The value of $|\mathsf{R}|_{\text{core}}$ may increase when there is a rewrite performed on a built-in, as previously shown with equations (15.6) to (15.9). However, every time $|\mathsf{R}|_{\text{core}}$ increases, it will correspond with a decrease in the first element of this pair.

We can therefore define an ordering on these order pairs as equation (15.10):

$$\langle a,b\rangle < \langle c,d\rangle \overset{\text{def}}{=} \begin{cases} a < c & \textbf{if } a \neq c \\ b < d & \textbf{otherwise} \end{cases} \tag{15.10}$$

It can also be seen from the construction of this ordered pair that there does not exist an infinite descending sequence, as both the first and second values in the tuple are positive natural numbers $\mathbb{N}_{\geq 1}$.[243]

15.3.6 Checking Energy of Rewrites

All of the *directional* rewrites in chapter §6 have been checked to ensure that they decrease the energy. With the few notable exceptions that were previously mentioned in section §15.3.3, the reason a rewrite decreases the energy tends to be *obvious*. For example, consider rewrite rule 63 $\texttt{if(1+M,R,S)} \xrightarrow{63} \texttt{R}$. The energy of $|\texttt{if(1+M,R,S)}|_{\text{core}} = |\texttt{1+M}|_{\text{core}} + |\texttt{R}|_{\text{core}} + |\texttt{S}|_{\text{core}}$, which is clearly larger than $|\texttt{R}|_{\text{core}}$. Furthermore, the definition of energy has that **R**-exprs that are *higher up* in the **R**-expr (more factored), have a lower energy. This means that rewrites that make the **R**-expr *more factored* will reduce the energy. This matches with the principle from section §8.2.3 that we will create factored **R**-exprs to get around the issues of aggregation.

This concludes the proof that Simplify will decrease the energy of the **R**-expr during every step under the assumptions that the **R**-expr is bounded in size and

[243]This could also be represented as an ordinal number $\langle a,b\rangle = \omega^a + b$. Given that we do not need the "full power" of ordinal numbers, we have chosen to represent this as an ordered pair of two natural numbers instead of an ordinal number.

the built-ins and rewrites on the built-ins terminate. Therefore, SimplifyNormalize, which repeatedly invokes Simplify will eventually reach a stopping point when Simplify has reached a fixed point since the energy cannot decrease *forever*. ■

Admittedly, this proof is not useful for proving that SimplifyNormalize terminates in *useful* amount of time or achieves an optimal asymptotic runtime.

Chapter 16

Future work

This dissertation represents my work on the Dyna programming language and the development of the **R**-expr-based rewrite system. Over the years, our research group has discussed a number of potential directions for which additional work is required, and I have additionally thought of a number of potential project directions on my own. I think one could easily fill ten to twenty person-years adding to the Dyna system and turning it into a more practical tool for researchers. Here are some ideas that I think are worthy of consideration as well as brief comments about how one might go about achieving these ideas.

16.1 Additional Disjunctive **R**-exprs

As noted in section §11.6.1, there is currently one efficient disjunctive **R**-expr kind. Realistically, there should be many different of efficient disjunctive kinds for different scenarios (e.g. [120]).

16.1.1 Improvements to the Trie

The current trie implementation has a few limitations which should be addressed. First, there are no secondary indexes on the trie. This means that if we order the variables as `X` and then `Y`, but want to enumerate the domain of `Y` using an iterator, we first need to enumerate all possible values of `X`.

Second, the order in which variables are stored in the trie should somehow be controllable. Currently, the system just picks the variable order arbitrarily—which is clearly suboptimal. The first step here would likely be to figure out some annotation that can be specified by the user about the order and indexes that a trie should maintain. Then any information collected from the user's annotation would have to be tracked and pushed all the way into the efficient disjunct. This is a little bit harder than it might sound at first. A disjunct does not have a one-to-one correspondence with a user-defined rule. Hence it can be tricky to track which disjunct should have what policy.

16.1.2 Dense Numerical Types

Given that our target audience are researchers who write numerical algorithms, we should really have some way to represent matrices and dense numerical arrays. Ideally, this would be something that would not require any changes to the user's program to use dense array types, though that might be difficult in practice.

I think the first step towards this would be to define a dense array, matrix, and tensor types as **R**-exprs. From there, one could look into either custom syntax or just defining built-in functions that expose operations on the dense arrays. This would be akin to building a NumPy [90] like matrix library for Dyna.

```
(def-base-rexpr dense-array-type [:var array-index
                                  :var output-value
                                  :other pointer-to-dense-array])
```

Figure 16-1. Theoretical way in which a dense array could be exposed into an **R**-expr.

16.1.3 Backed by Something other than Memory

Currently, all disjuncts are over **R**-exprs and held as structures in memory. There could realistically be a disjunctive type that represents other sources of data. For example, there could be an **R**-expr that reads data from a CSV file or a SQL database. There are no complications here to overcome in implementing this might be a good project to give to an undergraduate who wants to get started on the Dyna project.

16.2 User Studies

The work in this dissertation focused on the *implementation* of the Dyna programming language, with the design completed before the start of my Ph.D. back in 2011 [59]. Now that we finally have a working version of Dyna, I think prioritizing user studies would be beneficial. I think this would be an opportunity to determine which future work should be prioritized.

As an example, I note that the landscape of ML and AI research and tooling has shifted significantly from when Dyna was initially conceived—with Dyna's design predating modern neural networks by at least five years.

16.2.1 Libraries Written in Dyna

Related to the changing landscape of ML/AI research, most researchers are leveraging a large collection of existing software when developing ML models. In particular, neural networks are virtually never written from scratch and instead take advantage of a large collection of pre-existing modules from libraries such as PyTorch or Tensorflow [4, 23, 111]. If a user were to attempt the same in Dyna today, they only have core Dyna operations, which in this case are akin to low-level tensor manipulations.

I think that developing a useful library of neural techniques or automatic differentiation—which could either be done internally on **R**-exprs, or externally on source level Dyna. This would serve as a useful exercise in writing Dyna programs while simultaneously resulting in something that is useful to other users.

16.2.2 User-Friendliness

The current system is at the level of a *research prototype* in that it works, but I think the system's user-friendliness can still be greatly improved. These projects are probably at the right level to get an undergraduate started on the Dyna project.

1. Better Syntax Errors—Syntax errors currently returned are from the Antlr4 [110] parser and could be greatly improved.

2. Better Semantic Errors—The only warning currently reported to the user is attempting to use a user-defined term without any rules that define it (which is represented as multiplicity `0`). Other issues, such as type incompatibility (e.g., `int(X)*string(X)`$\to$`0`), should probably also be reported, but it is currently difficult to determine if type incompatibility was an error or was intentional.

3. Better Display of **R**-exprs—There is some code for printing **R**-exprs to the terminal. I have attempted to make the printed representation as close to the representation presented throughout this dissertation, but it is not 100% perfect. Furthermore, I do not believe that **R**-exprs are the best presentation for people who are trying to *use* the language (rather than understand its internals). As such, I think it would be worth reconsidering how **R**-exprs are presented to the user, possibly designing some **R**-expr to Dyna translation so that **R**-exprs can be presented as Dyna code.

4. The front-end parser currently reports an error if there is more than one dynabase defined per rule. This restriction could removed by adding additional transformations in the translation of dynabases to **R**-exprs, such as those described in section §13.2.1.

5. Debuggability and Visualization—There is currently no support for debugging the user's program. When I encounter issues, I have to drop into the Java debugger and step through **R**-expr rewrites—which is not a pleasant experience.

   One possible approach to debugging could be to go through the steps of rewriting that are performed. There is already support in the implementation for tracking how an **R**-expr is constructed (when the right command line flags are passed). So, this might be building a user interface to expose those details.

   A second approach could be to build a tool for visualizing the values of different user-defined rules in the program. This was previously done with Dyna 1.0 with the derivative *Dynasty* project [62].

6. Visualization and Interactions in Jupyter notebooks [99]—I have defined a %dyna Jupyter notebook cell/line magic[244], which allows for running simple Dyna programs from Jupyter notebooks. This user experience could also be improved with better visualizations.

16.3 More Rewrite Rules

Our implementation currently contains a number of rewrite rules. However, this is by no means expected to be all possible rewrites. It is reasonable that one could invent new rewrites which can efficiently handle different scenarios. There could

[244]https://ipython.readthedocs.io/en/stable/config/custommagics.html

also be more rewrite rules that correspond to different mathematical operations. For example, there could be rewrite rules involving identities of trigonometric functions.

16.4 Automatic Configuration

Currently, there are a number of different "*decisions*" that need to be made to configure how the program runs and can have a significant impact on the program's runtime. For example, we already have seen memoization, which is controllable by `$memo` and `$priority`. However, other things, such as variable order and indexing on disjunctions and variable loop order, should be considered.

Some of these proposed automatic configurations are akin to database query optimizers. However, controls like `$memo` are much more complicated, as a bad policy can cause the system to become inoperable and have consequences outside of the scope of a single query (section §10.6.2).

I should also note that the automatic algorithm configuration was the central topic of Tim Vieria's dissertation [141], in that he explored different ways a program could be folded to make it more efficient (chapter §14). Many of the ideas in Tim's dissertation should be adaptable to the system presented here.

16.4.1 Automatic Guessing on Cycles

In the current implementation, all memoization decisions are manual. However, there are a few cases that could be automatically detected and would likely improve the user experience. One such example is the case of a cyclic program. The system will detect that a cycle happens and eventually stop expanding the **R**-expr (section §15.3.1). However, it does not currently enable memoization automatically.

In the case that a cycle is detected that does not contain memoization, then memoization could be automatically enabled, or a helpful error message could be reported to the user about how they can fix their program.

16.4.2 Automatic Prioritization of Updates

Prioritization of updates is currently entirely dependent on the user's declarations. I believe there is an "easy to exploit" opportunity here to make this process automatic. The reason is that a bad update prioritization should not cause the system to accidentally not-terminate in most cases (like in the case of a bad `$memo`). Hence, it is "mostly safe" to experiment with different prioritization when the system is running.

I believe that a possible approach would be to run tasks and track when "priority inversions" happens. Essentially, the system could detect priority inversions by tracking when a downstream value was previously computed has to be recomputed.

A machine learning regression model could be trained to predict numerical priorities such that no priority inversions will happen. In the case that the program does not contain a cycle, then this would be akin to using features on the nodes in a graph to learn a topological ordering. Additional handling might be required in the case of cycles, though I believe that it might be possible to set up the machine learning algorithm such that cycles do not change the learned weights of the machine learning algorithm by having the notifications of priority inversion cancel each other out.

16.4.3 Automatic Variable Ordering

The variable order used by disjunctive data structures, such as the trie, can have an impact on the efficiency of the system. A bad variable ordering should not cause any issues around non-termination, so it should be safe to explore different representations.

I am not sure if there is an "efficient" way in which runtime costs can be associated with a variable ordering. It is possible that the variable orderings up being *zero-sum*, in that improving the runtime of one operation could negatively impact the runtime of something else. I believe that doing this correctly will require a reinforcement learning approach, where different orders are experimented with until something satisfactory is found [142].

16.5 Concurrency

Currently, the system is single-threaded. I have tried to design the internal data structures with the intention of *eventually* being used in a concurrent, multi-threaded environment, so hopefully, adding in concurrency is not too much of a chore. I think that the best approach would be to leave the rewriting done by Simplify and SimplifyNormalize as single-threaded and instead focus on doing parallel processing of the pending work on the UpdateQueue (section §10.7). The system could reasonably process updates in parallel, tracking if a conflict occurs much like a database. The update operations are not externally observable, so simple retry logic should be sufficient in most cases.

16.6 The Memoization Update Queue

The UpdateQueue used in the implementation of memoization (section §10.7) is a priority queue that is controlled by `$priority(·)` (section §10.8.5). Hence, this requires $O(\log N)$ time to push and pop update operations from this queue, where N is the number of update messages on the queue. Comparably, a dynamic program with a predetermined execution order does not require a queue and will avoid this $O(\log N)$ operations. When problems get sufficiently large, this extra $O(\log N)$ overhead might become significant, so figuring out how to eliminate this overhead may be a good idea.

16.7 GPU Coprocessor

GPUs are very important to modern ML algorithms and are 100% essential to the implementation of large Neural Networks. I think that there are a few potential ways that GPU support could be added:

1. Expose existing GPU Kernels—There are a number of different GPU kernels that currently exist. Using existing kernels usually results in the best performance.

   This could either be done by defining rules that match with the existing GPU kernels and calling those operations when the program matches an existing pattern. Alternately, one could simply define built-in operations for every kernel and require that the user manually reference the GPU kernel operations themselves.

2. Compiling Dyna to custom GPU Kernels—It should be possible to compile Dyna programs to a GPU. This might want to build on the work done for the JIT compiler (chapter §12), as it is already attempting to generate nested loops over variables' domains. Some challenges with this approach would be that the system still allows for unpredictable **R**-exprs to be returned at various points while executing. This would mean that the system would either have to require some limits on what can be represented or somehow support **R**-exprs on the GPU.

3. GPU DSL—Dyna has support for implementing DSL in the language (section §2.10), one could create a GPU DSL which compiles into a custom **R**-expr type and defines its own rewrites so that it is integrated with the rest of the system.

4. Develop an explicit matrix type and library of matrix operations that use a GPU. This would be an extension of the dense numerical types proposed previously (section §16.1.2).

16.8 External Solvers

Although we can continue to add many features to the **R**-expr system to expand the ability of the Dyna programming language, we should also consider that there exist a number of powerful frameworks and software tools that might be worth leveraging in the Dyna project. For example, SMT solvers are able to efficiently solve SAT-like problems that require searching through possible assignments [47, 48, 113]. Being specially designed for a "more limited" problem formalism than Dyna, their implementation is much more specialized than we would be able to realistically emulate with **R**-exprs. Additionally, there are algebra systems designed to solve formulas, such as Mathematica and SymPy, both of which have a very large collection of algebra rules [103, 150].[245]

Integrating external solvers into Dyna could be integrated as an external library (as suggested in section §2.10.1 with an example of a linear programming module)

[245] https://www.wolfram.com/engine/

or as custom rewrite rules against **R**-exprs.

16.9 Advanced Update Propagation

The updates of memoized values are currently handled by recomputation of the original **R**-expr. While this is sufficient for the small programs with which we are experimenting with currently, in the future, it might be beneficial if updates could be aware of the values and kinds of updates that are being propagated through the system.

For example, suppose that we have that `a(I,J)` is representing a matrix, and there is a rank-1 update to the matrix. A rank-1 update could potentially modify all entries in the matrix `a(I,J)`. However, knowing that `a(I,J)` received a rank-1 update could allow for more asymptotically efficient computation to be used downstream.

16.10 Automatic Runtime Analysis and Folding of Programs

This was part of Tim Vieira's dissertation [141]. However, that work was not done using the **R**-expr formalism, and the work to integrate these two bodies of work has not been started. Chapter §14 discussed how folding can be done on **R**-exprs, which is critical for adapting Tim's dissertation work.

16.11 Improved Context $\mathcal{C}$

The context $\mathcal{C}$ currently uses a hash-map to track the current value assigned to variables. This means that every time that matcher preconditions such as `:ground` or `:free` are evaluated, or when the `get-value` function is called, the system consult the hash-map (section §11.5.3). Hence, the context and the hash-maps that are used in its implementation have the potential to become a major bottleneck. Currently, the hash-map used is the default Clojure hash-map. It is possible that simply replacing the hash-map with another hash-map implementation that is more efficient could improve the runtime.

As a more long term solution, redesigning the get-value-of-variable mechanism such that there are as few memory access operations as possible and no hard-to-predict branches would probably be most beneficial. One possible solution to this might be to make a new class that implements the value type interface, which supports faster operations than the standard named variable. The context could then contain an array in addition to the hash-map that would enable retrieving the relevant value in a single memory access.

Chapter 17

Conclusion

This dissertation documents my work in implementing the Dyna language using **R**-expr based term rewriting. For the first time in the Dyna 2.0 project, we have an operational semantics for the language, an approach to implement the language, and an implementation at the level of the research prototype. A foundation has been laid for future work on the Dyna project. Although the material in this dissertation does *work*, I think there is still a lot of work to do to make it usable. I believe this will easily occupy the next several years of the Dyna project, and I hope to see this work continue, as I believe there is great promise.

Bibliography

[1] Dyna.org website — logic programming for machine learning. http://dyna.org.

[2] Torchscript – pytorch. https://pytorch.org/docs/stable/jit.html.

[3] 2008. Journal of functional and logic programming (jflp). http://danae.uni-muenster.de/lehre/kuchen/JFLP/.

[4] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[5] Aït-Kaci, H. 1999. Warren's abstract machine: A tutorial reconstruction.

[6] Antoy, S. 2010. Programming with narrowing: A tutorial. *Journal of Symbolic Computation 45,* 5, 501 – 522. Symbolic Computation in Software Science.

[7] Antoy, S., Echahed, R., and Hanus, M. 2000. A needed narrowing strategy. *J. ACM 47,* 4 (jul), 776–822.

[8] Antoy, S. and Hanus, M. 2010. Functional logic programming. *Commun. ACM 53,* 4 (apr), 74–85.

[9] Apt, K. R. and Wallace, M. 2007. *Constraint Logic Programming using ECLiPSe.* Cambridge University Press.

[10] Arias, E. J. G., Lipton, J., and Mariño, J. 2015. Declarative compilation for constraint logic programming. In *Logic-Based Program Synthesis and Transformation*, M. Proietti and H. Seki, Eds. Springer International Publishing, Cham, 299–316.

[11] Augustsson, L., Breitner, J., Claessen, K., Jhala, R., Peyton Jones, S., Shivers, O., Steele Jr., G. L., and Sweeney, T. 2023. The Verse calculus: A core calculus for deterministic functional logic programming. *Proc. ACM Program. Lang. 7,* ICFP (aug).

[12] Baader, F. and Nipkow, T. 1998. *Term rewriting and all that / Franz Baader and Tobias Nipkow.* Cambridge University Press, Cambridge, U.K. ;.

[13] Baeten, J. C. M., Bergstra, J. A., Klop, J. W., and Weijland, W. P. 1989. Term-rewriting systems with rule priorities. In *Theoretical Computer Science*. Vol. 67.

[14] Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., and Meuter, W. d. 2013. A survey on reactive programming. *ACM Comput. Surv. 45,* 4 (aug).

[15] Barrett, C., Fontaine, P., and Tinelli, C. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[16] Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. 2009. Satisfiability modulo theories. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 825–885.

[17] Bellia, M. and Occhiuto, M. E. 1993. C-expressions: a variable-free calculus for equational logic programming. *Theoretical Computer Science.*

[18] Bellman, R. 1954. The theory of dynamic programming. *Bull. Amer. Math. Soc. 60,* 6 (11), 503–515.

[19] Bellman, R. 1958. On a routing problem. *Quarterly of Applied Mathematics 16*, 87–90.

[20] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. 2009. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems.* ICOOOLPS '09. Association for Computing Machinery, New York, NY, USA, 18–25.

[21] Bolz, C. F., Leuschel, M., and Rigo, A. 2009. Towards Just-In-Time partial evaluation of Prolog. In *Logic-Based Program Synthesis and Transformation, 19th International Symposium, LOPSTR 2009, Coimbra, Portugal, September 2009, Revised Selected Papers.*

[22] Boulanger, D. and Bruynooghe, M. 1993. Deriving fold/unfold transformations of logic programs using extended oldt-based abstract interpretation. *Journal of Symbolic Computation 15,* 5, 495 – 521.

[23] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. 2018. JAX: composable transformations of Python+NumPy programs.

[24] Brassel, B., Hanus, M., Peemöller, B., and Reck, F. 2011. Kics2: A new compiler from curry to haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011).* Springer LNCS 6816, 1–18.

[25] Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *J. ACM 24,* 1 (Jan.), 44–67.

[26] Byrd, W. 2015. Differences between minikanren and prolog. http://minikanren.org/minikanren-and-prolog.html.

[27] Byrd, W. E. 2009. Relational programming in minikanren: Techniques, applications, and implementations. Ph.D. thesis, USA. AAI3380156.

[28] Byrd, W. E. 2012. minikanren.org. http://miniKanren.org.

[29] Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software, Articles 76,* 1.

[30] Ceri, S., Gottlob, G., and Tanca, L. 1989. What you always wanted to know about Datalog (and never dared to ask). In *IEEE Transactions on Knowledge and Data Engineering*. Number 1. 146–166.

[31] Chang, M., Bebenita, M., Yermolovich, A., Gal, A., and Franz, M. 2007. Efficient just-intime execution of dynamically typed languages via code specialization using precise runtime type inference.

[32] Cirstea, H., Lenglet, S., and Moreau, P.-E. 2017. Faithful (meta-)encodings of programmable strategies into term rewriting systems. *Logical Methods in Computer Science 13,* 4:16 (November).

[33] Clavel, M., Durán, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., and Quesada, J. F. 2002. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science 285,* 2, 187–243.

[34] Clavel, M., Eker, S., Lincoln, P., and Meseguer, J. 1996. Principles of maude. *Electronic Notes in Theoretical Computer Science 4*, 65–89. RWLW96, First International Workshop on Rewriting Logic and its Applications.

[35] Clocksin, W. F. and Mellish, C. S. 1984. *Programming in Prolog*. Springer-Verlag, New York, NY, USA.

[36] Codd, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM 13,* 6.

[37] Cohen, W. W., Yang, F., and Mazaitis, K. 2017. Tensorlog: Deep learning meets probabilistic dbs. *CoRR abs/1707.05390.*

[38] Colmerauer, A. and Roussel, P. 1996. *The Birth of Prolog.* Association for Computing Machinery, New York, NY, USA, Chapter 7, 331–367.

[39] Conway, T. C., Henderson, F., and Somogyi, Z. 1995. Code generation for mercury. In *ILPS*.

[40] Cook, M. 2004. Universality in elementary cellular automata. *Complex Systems 15*.

[41] Cook, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Association for Computing Machinery, New York, NY, USA, 151–158.

[42] Cook, W. R. and Lämmel, R. 2011. Tutorial on online partial evaluation. In *IFIP Working Conference on Domain-Specific Languages 2011 (DSL 2011)*, O. Danvy and C.-C. Shan, Eds.

[43] da Silva, A. F. and Costa, V. S. 2006. The design and implementation of the yap compiler: An optimizing compiler for logic programming languages. In *Logic Programming*, S. Etalle and M. Truszczyński, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 461–462.

[44] Date, C. J. 1989. *A Guide to the SQL Standard (2nd Ed.).* Addison-Wesley Longman Publishing Co., Inc., USA.

[45] Davis, M., Logemann, G., and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM 5,* 7.

[46] Dayal, U., Goodman, N., and Katz, R. H. 1982. An extended relational algebra with control over duplicate elimination. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. PODS '82. Association for Computing Machinery, New York, NY, USA, 117–123.

[47] de Moura, L. and Bjørner, N. 2008. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[48] De Moura, L. and Bjørner, N. 2011. Satisfiability modulo theories: Introduction and applications. *Commun. ACM 54,* 9 (sep), 69–77.

[49] de Moura, P. J. L. 2003. Logtalk design of an object-oriented logic programming language. Ph.D. thesis.

[50] Denecker, M. and De Cat, B. 2010. Dpll(agg): An efficient smt module for aggregates. https://lirias.kuleuven.be/retrieve/112271.

[51] Dershowitz, N. 1987. Termination of rewriting. *Journal of Symbolic Computation 3*.

[52] Diaz, D., Abreu, S., and Codognet, P. 2010. On the implementation of GNU prolog. *CoRR abs/1012.2496*.

[53] Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 269–271.

[54] Dreyer, M. and Eisner, J. 2006. Better informed training of latent syntactic features. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Sydney, 317–326.

[55] Dreyer, M., Smith, D. A., and Smith, N. A. 2006. Vine parsing and minimum risk reranking for speed and precision. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*. Association for Computational Linguistics, New York City, 201–205.

[56] Duboscq, G., Stadler, L., Würthinger, T., Simon, D., Wimmer, C., and Mössenböck, H. Graal ir: An extensible declarative intermediate representation.

[57] Duboscq, G., Würthinger, T., Stadler, L., Wimmer, C., Simon, D., and Mössenböck, H. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. VMIL '13. ACM, New York, NY, USA.

[58] Eisner, J. and Blatz, J. 2007. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of FG 2006: The 11th Conference on Formal Grammar*, S. Wintner, Ed. CSLI Publications.

[59] Eisner, J. and Filardo, N. W. 2011. Dyna: Extending Datalog for modern AI. In *Datalog Reloaded*, O. de Moor, G. Gottlob, T. Furche, and A. Sellers, Eds. Lecture Notes in Computer Science, vol. 6702. Springer.

[60] Eisner, J., Goldlust, E., and Smith, N. A. 2004. Dyna (1): A declarative language for implementing dynamic programs. In *Proceedings of Association for Computational Linguistics (ACL), Companion Volume*. Barcelona.

[61] Eisner, J., Goldlust, E., and Smith, N. A. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP)*. Vancouver, 281–290.

[62] Eisner, J., Kornbluh, M., Woodhull, G., Buse, R., Huang, S., Michael, C., and Shafer, G. 2006. Visual navigation through large directed graphs and hypergraphs. In *Proceedings*

of the IEEE Symposium on Information Visualization (InfoVis'06), Poster/Demo Session. Baltimore, 116–117.

[63] Eisner, J. and Smith, N. A. 2005. Parsing with soft and hard constraints on dependency length. In *Proceedings of the International Workshop on Parsing Technologies (IWPT).* Vancouver, 30–41.

[64] FANDINNO, J. and SCHULZ, C. 2019. Answering the "why" in answer set programming – a survey of explanation approaches. *Theory and Practice of Logic Programming 19,* 2, 114–203.

[65] Fierens, D., Van Den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming 15,* 3.

[66] Filardo, N. W. 2017. Dyna 2: Towards a general weighted logic language. Ph.D. thesis.

[67] Filardo, N. W. and Eisner, J. 2012. A flexible solver for finite arithmetic circuits. In *Technical Communications of the International Conference on Logic Programming*, A. Dovier and V. S. Costa, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 17. Budapest.

[68] Filardo, N. W. and Eisner, J. 2016. Rigid tree automata with isolation. In *Proceedings of the Fourth International Workshop on Trends in Tree Automata and Tree Transducers (TTATT).* Seoul.

[69] Ford, L. 1956. Network flow theory.

[70] Francis-Landau, M., Vieira, T., and Eisner, J. 2020. Evaluation of logic programs with built-ins and aggregation: A calculus for bag relations. In *13th International Workshop on Rewriting Logic and Its Applications.* 49–63.

[71] Friedman, D. P., Byrd, W. E., and Kiselyov, O. 2005. *The Reasoned Schemer.* The MIT Press.

[72] Frühwirth, T. 1998. Theory and practice of constraint handling rules. *The Journal of Logic Programming 37,* 1, 95 – 138.

[73] Futamura, Y. 1983. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering.* Springer.

[74] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., and Franz, M. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on*

Programming Language Design and Implementation. PLDI '09. ACM, New York, NY, USA, 465–478.

[75] Gal, A., Probst, C. W., and Franz, M. 2006. Hotpathvm: An effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*. VEE '06. Association for Computing Machinery, New York, NY, USA, 144–153.

[76] Gallaire, H., Minker, J., and Nicolas, J.-M. 1984. Logic and databases: A deductive approach. *ACM Comput. Surv. 16,* 2 (June), 153–185.

[77] Gallego Arias, E. J., Lipton, J., and Mariño, J. 2017a. Constraint logic programming with a relational machine. *Formal Aspects of Computing 29,* 1 (Jan), 97–124.

[78] Gallego Arias, E. J., Lipton, J., and Mariño, J. 2017b. Constraint logic programming with a relational machine. *Formal Aspects of Computing 29,* 1 (Jan), 97–124.

[79] Gebser, M., Kaufmann, B., and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell. 187–188*, 52–89.

[80] Gergatsoulis, M. and Katzouraki, M. 1994. Unfold/fold transformations for definite clause programs. In *Programming Language Implementation and Logic Programming*, M. Hermenegildo and J. Penjam, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 340–354.

[81] Goodman, N. D., Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. B. 2008. Church: A language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*. UAI'08. AUAI Press, Arlington, Virginia, USA, 220–229.

[82] Greco, S. 1999. Dynamic programming in datalog with aggregates. *IEEE Trans. on Knowl. and Data Eng. 11,* 2 (Mar.).

[83] Green, T. J. 2009. Bag semantics. In *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Springer, Boston, MA, 201–206.

[84] Hanus, M. 1997. A unified computation model for functional and logic programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. ACM, New York, NY, USA.

[85] Hanus, M. 2007. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*. Springer LNCS 4670, 45–75.

[86] Hanus, M., Antoy, S., Brassel, B., Kuchen, H., López-Fraguas, F. J., Lux, W., Navarro, J. J. M., Peemöller, B., and Steiner, F. 2016. Curry an integrated functional logic language.

[87] Hanus, M., Kuchen, H., and Moreno-Navarro, J. J. 1995. Curry: A truly functional logic language.

[88] Hanus, M. and Prehofer, C. 1996. Higher-order narrowing with definitional trees. In *Rewriting Techniques and Applications*, H. Ganzinger, Ed. Springer Berlin Heidelberg, Berlin, Heidelberg, 138–152.

[89] Hanus, M. and Sadre, R. 1999. An abstract machine for curry and its concurrent implementation in java. *J. Funct. Log. Program. 1999*.

[90] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. 2020. Array programming with NumPy. *Nature 585,* 7825 (Sept.), 357–362.

[91] Hemann, J., Friedman, D. P., Byrd, W. E., and Might, M. 2018. A Simple Complete Search for Logic Programming. In *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*, R. Rocha, T. C. Son, C. Mears, and N. Saeedloei, Eds. OpenAccess Series in Informatics (OASIcs), vol. 58. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[92] Hickey, R. 2008. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*. DLS '08. Association for Computing Machinery, New York, NY, USA.

[93] Hoffmann, B. 1992. Term rewriting with sharing and memoïzation. In *Algebraic and Logic Programming: Proc. of the Third International Conference*. Springer, 128–142.

[94] Kamperman, J. 1996. Compilation of term rewriting systems. Ph.D. thesis, University of Amsterdam.

[95] Karakos, D., Eisner, J., Khudanpur, S., and Dreyer, M. 2008. Machine translation system combination using ITG-based alignments. In *Proceedings of ACL-08: HLT, Short Papers*. Columbus, Ohio, 81–84.

[96] Karp, R. M. 1972. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 85–103.

[97] Kawamura, T. and Kanamori, T. 1990. Preservation of stronger equivalence in unfold/-fold logic program transformation. *Theoretical Computer Science 75,* 1, 139–156.

[98] Klug, A. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM 29,* 3 (jul), 699–717.

[99] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., and Willing, C. 2016. Jupyter notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 87 – 90.

[100] Kowalski, R. 1979. Algorithm = logic + control. *Commun. ACM 22,* 7 (jul), 424–436.

[101] Lloyd, J. W. J. W. 1987. *Foundations of logic programming / J.W. Lloyd.*, 2nd, extended ed. ed. Symbolic computation. Artificial intelligence. Springer-Verlag, Berlin ;.

[102] Martelli, A. and Montanari, U. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4,* 2, 258–282.

[103] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. 2017. Sympy: symbolic computing in python. *PeerJ Computer Science 3*, e103.

[104] Michie, D. 1968. “memo” functions and machine learning. *Nature 218,* 5136 (Apr), 19–22.

[105] Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic.* Cambridge University Press.

[106] Mishra, P. and Eich, M. H. 1992. Join processing in relational databases. *ACM Comput. Surv. 24,* 1 (mar), 63–113.

[107] Monniaux, D. 2016. A Survey of Satisfiability Modulo Theory. In *Computer Algebra in Scientific Computing.* Bucharest, Romania.

[108] Overton, D. 2003. Precise and expressive mode systems for typed logic programming languages. Ph.D. thesis, University of Melbourne.

[109] Pall, M. The LuaJIT Project.

[110] Parr, T. 2013. *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf.

[111] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. 2017. Automatic differentiation in pytorch. In *NIPS 2017 Workshop on Autodiff.*

[112] Pfeffer, A. 2016. *Practical probabilistic programming / Avi Pfeffer.*, 1st edition ed. Manning Publications, Shelter Island, New York.

[113] Rossi, F., van Beek, P., and Walsh, T., Eds. 2006. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence, vol. 2. Elsevier.

[114] Roșu, G. and Șerbănută, T. F. 2010. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming 79,* 6, 397–434. Membrane computing and programming.

[115] Rucker, R. 1982. *Infinity and the mind: the science and philosophy of the infinite*. Birkhäuser, Boston.

[116] Ruf, E. 1993. Topics in online partial evaluation. Ph.D. thesis, Stanford University. Published as Technical Report CSL-TR-93-563.

[117] Schafer, C. 2006. Novel probabilistic finite-state transducers for cognate and transliteration modeling. In *Proceedings of the 7th Conference of the Association for Machine Translation in the Americas: Technical Papers*. Association for Machine Translation in the Americas, Cambridge, Massachusetts, USA, 203–212.

[118] SCHIMPF, J. and SHEN, K. 2012. ECLiPSe – From LP to CLP. *Theory and Practice of Logic Programming 12,* 1-2, 127–156.

[119] Scholz, B., Jordan, H., Subotić, P., and Westmann, T. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. ACM, New York, NY, USA.

[120] Sekar, R., Ramakrishnan, I. V., and Voronkov, A. 2001. *Term Indexing*. Elsevier Science Publishers B. V., NLD, 1853–1964.

[121] Smith, D. A. and Eisner, J. 2006a. Minimum-risk annealing for training log-linear models. In *Proceedings of the International Conference on Computational Linguistics and the Association for Computational Linguistics (COLING-ACL), Companion Volume*. Sydney, 787–794.

[122] Smith, D. A. and Eisner, J. 2006b. Quasi-synchronous grammars: Alignment by soft projection of syntactic dependencies. In *Proceedings of the HLT-NAACL Workshop on Statistical Machine Translation*. New York, 23–30.

[123] Smith, D. A. and Eisner, J. 2008. Dependency parsing by belief propagation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Honolulu, 145–156.

[124] Smith, D. A. and Smith, N. A. 2004. Bilingual parsing with factored estimation: Using English to parse Korean. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Barcelona, Spain, 49–56.

[125] Smith, N. A. and Eisner, J. 2004. Annealing techniques for unsupervised statistical language learning. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. ACL '04. Association for Computational Linguistics, USA, 486–es.

[126] Smith, N. A. and Eisner, J. 2005a. Contrastive estimation: Training log-linear models on unlabeled data. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. Association for Computational Linguistics, Ann Arbor, Michigan, 354–362.

[127] Smith, N. A. and Eisner, J. 2005b. Guiding unsupervised grammar induction using contrastive estimation. In *International Joint Conference on Artificial Intelligence (IJCAI) Workshop on Grammatical Inference Applications*. Edinburgh, 73–82.

[128] Smith, N. A. and Eisner, J. 2006c. Annealing structural bias in multilingual weighted grammar induction. In *Proceedings of the International Conference on Computational Linguistics and the Association for Computational Linguistics (COLING-ACL)*. Sydney, 569–576.

[129] Smith, N. A., Smith, D. A., and Tromble, R. W. 2005. Context-based morphological disambiguation with random fields. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Vancouver, British Columbia, Canada, 475–482.

[130] Smith, N. A., Vail, D. L., and Lafferty, J. D. 2007. Computationally efficient M-estimation of log-linear structure models. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. Association for Computational Linguistics, Prague, Czech Republic, 752–759.

[131] Socher-Ambrosius, R. and Johann, P. 1997. *Deduction systems.* Graduate texts in computer science. Springer, New York.

[132] Somogyi, Z., Henderson, F. J., and Conway, T. C. 1995. The implementation of mercury, an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*.

[133] Stein, L. A., Lieberman, H., and Ungar, D. 1989. *A Shared View of Sharing: The Treaty of Orlando.* Association for Computing Machinery, New York, NY, USA, 31–48.

[134] Suzuki, H. 2012. The internals of postgreSQL. https://www.interdb.jp/pg/.

[135] Swift, T. and Warren, D. S. 2010. XSB: Extending Prolog with tabled logic programming. *CoRR abs/1012.5123.* Under consideration for publication in Theory and Practice of Logic Programming.

[136] Taivalsaari, A. 1996. On the notion of inheritance. *ACM Comput. Surv. 28,* 3 (sep), 438–479.

[137] Thompson, J. 2006. Yield prolog - embed prolog in your code code. https://yieldprolog.sourceforge.net/.

[138] Tolpin, D., van de Meent, J.-W., and Wood, F. 2015. Probabilistic programming in anglican. In *Proceedings of the 2015th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III.* ECMLPKDD'15. Springer, Gewerbestrasse 11 CH-6330, Cham (ZG), CHE, 308–311.

[139] Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., and Blei, D. M. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*.

[140] Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems.* Vol. 1. Computer Science Press.

[141] Vieira, T. 2023. Automating the analysis and improvement of dynamic programming algorithms with applications to natural language processing. Ph.D. thesis.

[142] Vieira, T., Francis-Landau, M., Filardo, N. W., Khorasani, F., and Eisner, J. 2017. Dyna: Toward a self-optimizing declarative language for machine learning applications. In *Proc. of the ACM SIGPLAN Workshop on Machine Learning and Programming Languages.* ACM, Barcelona, 8–17.

[143] Vittek, M. 1996. A compiler for nondeterministic term rewriting systems. In *International Conference on Rewriting Techniques and Applications (RTA).*

[144] Warren, D. H. D. 1983. An abstract prolog instruction set. Tech. rep., SRI International Artificial Intelligence Center. 10.

[145] Warren, D. S. 1992. Memoing for logic programs. *Commun. ACM 35,* 3 (mar), 93–111.

[146] Wielemaker, J. and Anjewierden, A. 2002. An architecture for making object-oriented systems available from Prolog. In *WLPE.*

[147] Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

[148] Wiles, A. 1995. Modular elliptic curves and Fermat's last theorem. *Annals of Mathematics 141,* 3, 443–551.

[149] Wimmer, C. and Würthinger, T. 2012. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity.* SPLASH '12. ACM, New York, NY, USA.

[150] Wolfram, S. 2003. *The Mathematica Book*, 5th Edition ed. Wolfram-Media.

[151] XSB 2010. XSB. http://xsb.sourceforge.net/.

[152] Zhou, N.-F. and Sato, T. 2003. Toward a high-performance system for symbolic and statistical modeling. In *Proc. of the IJCAI Workshop on Learning Statistical Models from Relational Data.*