Linear Logical Algorithms

Robert J. Simmons **Frank Pfenning**

May 2008
CMU-CS-08-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Bottom-up logic programming can be used to declaratively specify many algorithms in a succinct and natural way, and McAllester and Ganzinger have shown that it is possible to define a cost semantics that enables reasoning about the running time of algorithms written as inference rules. Previous work with the programming language Lollimon demonstrates the expressive power of logic programming with linear logic in describing algorithms that have imperative elements or that must repeatedly make mutually exclusive choices. In this technical report, we identify a bottom-up logic programming language based on linear logic that is amenable to efficient execution and describe a novel cost semantics that can be used for complexity analysis of algorithms expressed in linear logic. This report is an extended companion to [16].

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Keywords: Bottom-up logic programming, forward reasoning, linear logic, deductive databases, cost semantics, abstract running time

1 Introduction

Logical inference rules are a concise and powerful tool for expressing many algorithms in a declarative way. In the last decade, several lines of work have advanced the argument that it is not only possible but convenient to formally reason about the *running time* of algorithms expressed as inference rules.

Work on this topic can be broadly categorized into two groups: work that takes a language similar to the pure bottom up logic programming language presented by McAllester [11] and automates reasoning about the complexity of algorithms expressed in that language [12, 9], and work aimed at allowing analysis for logic programming languages with richer features [5, 6, 13, 7].

This paper falls into the second category; we present a bottom-up logic programming language based on intuitionistic linear logic [2] that cleanly integrates a notion of state transition with the saturating forward reasoning present in pure bottom-up logic programming. We follow the two-part approach taken by McAllester and Ganzinger in [11, 5, 6]. First, we give the language a dynamic cost semantics called the *abstract running time* that looks at a chain of logical inferences as a computation and defines the cost of that computation, and then we describe an interpreter that can be shown to execute those computations in time proportional to the abstract running time. Both of these concepts are critical – without the interpreter, there is no reason to believe that the notion of abstract running time is based in reality, and without the definition of abstract running time, reasoning about the complexity of algorithms requires understanding the intricacies of the interpreter's implementation.

1.1 Pure bottom-up logic programming

We start by briefly describing a pure logic programming language [11] in which various graph algorithms and program analyses can be expressed concisely and executed efficiently. One example is the program in Figure 1 that computes connectivity over an undirected graph.

$$\frac{\mathsf{edge}(x,y)}{\mathsf{edge}(y,x)}\ r1 \qquad \frac{\mathsf{edge}(x,y)}{\mathsf{path}(x,y)}\ r2 \qquad \frac{\mathsf{edge}(x,y) \quad \mathsf{path}(y,z)}{\mathsf{path}(x,z)}\ r3$$

Figure 1: A simple pure, bottom-up program for computing graph connectivity.

Given a graph $G = (E, V)$, this algorithm starts with a database that has a fact $\mathsf{edge}(\mathsf{a}, \mathsf{b})$ for every edge $(\mathsf{a}, \mathsf{b}) \in E$. The intended meaning of this program is that $\mathsf{path}(\mathsf{a}, \mathsf{b})$ should hold if and only if there is a path between vertex a and b in graph G. Here, and throughout the paper, we will represent constants as $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$ and variables as $x, y, z, \ldots$, and we will insist that all the terms in our database be *ground*, meaning that they contain no free variables, and that all rules be *range-restricted*, meaning that the variables in the conclusions (below the line) are a subset of the variables in the premises (above the line). This last restriction ensures that the database continues to contain only ground facts as new facts are derived.

In order to calculate the path relation, rules are applied exhaustively in the forward direction until *saturation* is reached; that is, until no possible forward inference can cause us to learn anything new. The *closure* of an initial database Γ under the rules in a program $\mathcal{P}$ (written $\mathsf{Clo}_{\mathcal{P}}(\Gamma)$ or just $\mathsf{Clo}(\Gamma)$) is the smallest set containing Γ closed under the rules in $\mathcal{P}$. As a concrete example, say we want to compute the path relation over the graph $G = (\{(\mathsf{a},\mathsf{b}),(\mathsf{b},\mathsf{c})\},\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\})$. One possible evolution of this database is as follows:

Base facts:	$\mathsf{edge}(\mathsf{a},\mathsf{b}), \mathsf{edge}(\mathsf{b},\mathsf{c})$
After applying rule $r1$ **to fact** $\mathsf{edge}(\mathsf{a},\mathsf{b})$**, add:**	$\mathsf{edge}(\mathsf{b},\mathsf{a})$
After applying rule $r1$ **to fact** $\mathsf{edge}(\mathsf{b},\mathsf{c})$**, add:**	$\mathsf{edge}(\mathsf{c},\mathsf{b})$
After applying rule $r2$ **to fact** $\mathsf{edge}(\mathsf{a},\mathsf{b})$**, add:**	$\mathsf{path}(\mathsf{a},\mathsf{b})$
After applying rule $r2$ **to fact** $\mathsf{edge}(\mathsf{b},\mathsf{c})$**, add:**	$\mathsf{path}(\mathsf{b},\mathsf{c})$
After applying rule $r2$ **to fact** $\mathsf{edge}(\mathsf{b},\mathsf{a})$**, add:**	$\mathsf{path}(\mathsf{b},\mathsf{a})$
After applying rule $r3$ **to facts** $\mathsf{edge}(\mathsf{a},\mathsf{b})$ **and** $\mathsf{path}(\mathsf{b},\mathsf{c})$**, add:**	$\mathsf{path}(\mathsf{a},\mathsf{c})$
After applying rule $r2$ **to fact** $\mathsf{edge}(\mathsf{c},\mathsf{b})$**, add:**	$\mathsf{path}(\mathsf{c},\mathsf{b})$
After applying rule $r3$ **to facts** $\mathsf{edge}(\mathsf{c},\mathsf{b})$ **and** $\mathsf{path}(\mathsf{b},\mathsf{a})$**, add:**	$\mathsf{path}(\mathsf{c},\mathsf{a})$
Saturated!	

In terms of correctness, we do not have to worry about the specific implementation of Clo that determines in what order we apply inference rules to derive new facts, because any method of exhaustive forward reasoning applied in a fair manner (meaning that every rule that can be applied will eventually be applied) produces the same result due to the monotonic nature of the reasoning.

Unlike Datalog, the language contains function symbols, so the closure may be infinite. This would be a problem; in order for our programs to behave in a reasonable manner, it is important that we be able to implement the Clo function and show that it terminates. As an example of what can go wrong, consider bottom-up ground deduction with the base fact $\mathsf{nat}(\mathsf{z})$ and a single rule:

$$\frac{\mathsf{nat}(x)}{\mathsf{nat}(\mathsf{s}(x))}\ s$$

If we tried to perform exhaustive forward reasoning on that fact and that rule, we would proceed as follows:

Base facts:	$\mathsf{nat}(\mathsf{z})$
After applying rule s **to fact** $\mathsf{nat}(\mathsf{z})$**, add:**	$\mathsf{nat}(\mathsf{s}(\mathsf{z}))$
After applying rule s **to fact** $\mathsf{nat}(\mathsf{s}(\mathsf{z}))$**, add:**	$\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$
After applying rule s **to fact** $\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$**, add:**	$\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))$
After applying rule s **to fact** $\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))))$**, add:**	$\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))$
After applying rule s **to fact** $\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z}))))))$**, add:**	$\mathsf{nat}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))))))$
...	

... and so on without end.

Here, we are only interested in programs with a finite closure.

1.2 Related work

The pure bottom-up logic programming language sketched above and described fully in [11] and elsewhere has great expressive power but also some obvious limitations. The research addressing those limitations is far too extensive to review all of it here. Some approaches, like stratified negation and its many variants, should be compatible as extensions to our approach, as we discuss in Section 6. We will focus here on efforts most closely related to our own.

Consider the way we encoded the graph G in Figure 1. The collection of edges was represented not as a matrix or an adjacency list, but merely as a collection of facts - the data structure that we were working over was implicit in the database. This idiom of *database-as-data-structure* is a strength of this declarative style of programming, as details of underlying data structures can be omitted. However, because the notion of database we use is one that incrementally "learns" all derivable facts in an unspecified manner, it is difficult to describe algorithms that have distinct states or phases. Several attempts at addressing this problem amount to the identification of reasonable forms of locally stratified negation, such as temporal [14], state [8], or XY [1] stratification. However, stratified negation cannot easily describe algorithms that must repeatedly take only one of a number of possible steps. Greco and Zaniolo identify this as a particular stumbling block to the logical specification of greedy algorithms [7].

Several disconnected lines of research have approached this problem. Greco and Zaniolo describe a variant of Datalog with an intrinsic notion of *choice* that has a semantics based on stable models and can naturally express a number of greedy algorithms [7]. Their work builds on experience with the deductive database system LDL++ [1], an implemented language that integrates choice with other desirable features such as a XY-stratified negation. While they define an execution model for their system and show a number of complexity results for individual algorithms, they do not give a cost semantics, so all complexity results are based on directly reasoning about the interpreter.

Ganzinger and McAllester [6] do not explicitly consider the applicability of their system to greedy algorithms, but they demonstrate that their system, based on *deletion* of facts and *priorities* on rules, can express many of the same algorithms that motivated Greco and Zaniolo, such as algorithms computing minimum spanning trees and shortest paths. Unfortunately, the expressiveness of their system is hard to determine because they define an unusual notion of deletion that does not have any clear logical justification.

Pfenning and López et al. [15, 10] propose linear logic as a more principled foundation of Ganzinger and McAllester's work, and show that their implementation of a linear logic programming language, Lollimon, is powerful enough to express many of the algorithms shown in Ganzinger and McAllester's previous work. However, they can only consider the correctness, not the complexity, of algorithms presented in Lollimon, and it would appear to be extremely difficult to reasoning about the complexity of algorithms in a language such as Lollimon that allows for almost arbitrary integration of forward and backward chaining.

$$\frac{\underline{\mathsf{wins}}(x, n) \quad \underline{\mathsf{wins}}(y, n)}{\underline{\mathsf{wins}}(x, \mathsf{s}(n)) \quad \mathsf{won}(x, y, n)}\ \textit{play} \qquad \frac{\mathsf{won}(x, y, n)}{\mathsf{order}(x, y)}\ \textit{rank}_1 \qquad \frac{\mathsf{order}(x, y) \quad \mathsf{order}(y, z)}{\mathsf{order}(x, z)}\ \textit{rank}_2$$

Figure 2: A simple linear logic program describing arbitrary single-elimination tournaments.

1.3 Contributions

The primary contribution of this paper is the presentation of a programming language based on bottom-up reasoning in linear logic – essentially a first-order, Horn-like fragment of Lollimon – that is both useful for the specification of algorithms and in the analysis of their running time. To our knowledge, this is the first such result for a programming language based on linear logic. Section 2 describes the use of first-order linear logic in specifying a number of simple algorithms. Section 3 defines the operational semantics and cost semantics of the language and demonstrates the use of cost semantics in reasoning about complexity. Section 4 describes an interpreter that proves that the cost semantics are reasonable by running the programs specified in our language on a RAM machine extended with a constant-time hash table operations in time proportional to the abstract running time. Section 6 concludes and describes a number of possible extensions to the basic, pure language considered here.

2 Bottom-up programming in linear logic

The pure bottom-up logic programming language introduced in the previous section is built from *atomic propositions* like $\mathsf{nat}(n)$, $\mathsf{edge}(a, b)$, and $\mathsf{path}(v, u)$. These facts represent truth in the usual, mathematical sense - the rule $r2$ in Figure 1 says that if we know that there is an edge between some vertices a and b, we can also know that there is a path between them. However, after we learn $\mathsf{path}(\mathsf{a}, \mathsf{b})$, we still know $\mathsf{edge}(\mathsf{a}, \mathsf{b})$, because we treat truth as *persistent*.

Linear logic has a notion of persistent truth, but also has a notion of truth that describes the current (and possibly changing) state of the world. We refer to this notion of "truth in the current state of the system" as *ephemeral truth*, and in addition to the persistent atomic propositions that we have previously seen, we introduce ephemeral (or *linear*) atomic propositions that we distinguish from persistent propositions by using an underline: $\underline{\mathsf{linear}}(x)$.

2.1 Single-elimination tournaments

Rules with ephemeral propositions as premises introduce the possibility of changing the state of the world. The rule given in Figure 2 describes a single-elimination tournament in which any team can play another team that has the same number of wins. If we have two teams a and c that have both won zero games, we can represent this as the two linear atomic propositions $\underline{\mathsf{wins}}(\mathsf{a}, \mathsf{z})$ and $\underline{\mathsf{wins}}(\mathsf{c}, \mathsf{z})$. These atomic propositions satisfy the two premises of the rule in Figure 2. If we arbitrarily let $x = \mathsf{c}$ and $y = \mathsf{a}$, treating c as the "winning team," the rule represents the possibility

of transitioning from a state where both teams a and c have won zero games and are still in the running to a state where team c has won one game and where team a is out of the running. There is no wins(y, n) in the conclusion because the tournament is single-elimination – after losing, a team cannot play any other teams. Applying this rule requires *consuming* the two linear propositions we had before and replacing them with a single new linear atomic proposition wins$(\mathsf{c}, \mathsf{s}(\mathsf{z}))$. Applying the rule also adds the persistent atomic proposition won$(\mathsf{c}, \mathsf{a}, \mathsf{z})$ to the database, which represents a persistent record of the fact that c defeated a in round z.

One possibility of the evolution of a system with four teams is shown below. Notice that we have two sets of ground atomic propositions – persistent atomic propositions, as before, and a new set of ephemeral atomic propositions. The set of persistent propositions still grows monotonically, but ephemeral propositions are consumed when they are used in the application of a rule.

Initial state: wins(a, z), wins(b, z), wins(c, z), wins(d, z)

After applying rule *play* **to** wins(c, z) **and** wins(a, z)**, add:**	wins$(\mathsf{c}, \mathsf{s}(\mathsf{z}))$, won$(\mathsf{c}, \mathsf{a}, \mathsf{z})$
After applying rule $rank_1$ **to** won$(\mathsf{c}, \mathsf{a}, \mathsf{z})$**, add:**	order(c, a)
After applying rule *play* **to** wins(d, z) **and** wins(b, z)**, add:**	wins$(\mathsf{d}, \mathsf{s}(\mathsf{z}))$, won$(\mathsf{d}, \mathsf{b}, \mathsf{z})$
After applying rule $rank_1$ **to** won$(\mathsf{d}, \mathsf{b}, \mathsf{z})$**, add:**	order(d, b)
After applying rule *play* **to** wins(c, s) **and** wins(d, s)**, add:**	wins$(\mathsf{c}, \mathsf{s}(\mathsf{s}(\mathsf{z})))$, won$(\mathsf{c}, \mathsf{d}, \mathsf{s}(\mathsf{z}))$
After applying rule $rank_1$ **to** won$(\mathsf{c}, \mathsf{d}, \mathsf{z})$**, add:**	order(c, d)
After applying rule $rank_2$ **to** order(c, d) **and** order(d, b)**, add:**	order(c, b)

Final state: wins$(\mathsf{c}, \mathsf{s}(\mathsf{s}(\mathsf{z})))$, won$(\mathsf{c}, \mathsf{a}, \mathsf{z})$, won$(\mathsf{d}, \mathsf{b}, \mathsf{z})$, won$(\mathsf{c}, \mathsf{d}, \mathsf{s}(\mathsf{z}))$, order$(\mathsf{c}, \mathsf{a})$, order$(\mathsf{d}, \mathsf{b})$, order$(\mathsf{c}, \mathsf{d})$, order$(\mathsf{c}, \mathsf{b})$

Changes to the state of a system are not necessarily reversible. While we could imagine a backtracking semantics that would eventually consider team a beating team c, or consider them playing other teams in the first round, we instead read rules with linear premises as describing a *committed choice* – once we apply a rule that consumes an ephemeral proposition, we will never consider any other way that proposition could have been consumed. Put another way, while our rules may describe a number of different ways a system could evolve from the initial state, when reading our rules as an algorithm, the algorithm will follow *one* particular evolution of that system in a *don't-care* non-deterministic manner, as is the case in many concurrent programming languages.

2.2 Rooted spanning trees

We can use these ephemeral atomic propositions to support algorithms that require certain actions to happen a fixed number of times, as well as algorithms that require some actions to be mutually exclusive. The example in Figure 3 is a linear algorithm to compute a spanning tree of a connected, undirected graph $G = (E, V)$ that has some distinguished vertex root $\in V$. The input to the algorithm is a persistent atomic proposition edge(a, b) for every edge $(\mathsf{a}, \mathsf{b}) \in E$ and a single ephemeral atomic proposition vert(a) for every vertex $\mathsf{a} \in V$. We view the relation tree as a directed subgraph of G where tree(a, b) is true iff there is an edge from a to b in the tree.

$$\frac{\mathsf{edge}(x,y)}{\mathsf{edge}(y,x)}\ r1 \qquad \frac{\underline{\mathsf{vert}}(\mathsf{root})}{\mathsf{intree}(\mathsf{root})}\ r2 \qquad \frac{\begin{array}{c}\mathsf{edge}(x,y)\\ \mathsf{intree}(x)\\ \underline{\mathsf{vert}}(y)\end{array}}{\begin{array}{c}\mathsf{tree}(x,y)\\ \mathsf{intree}(y)\end{array}}\ r3$$

Figure 3: Finding a rooted spanning tree of an undirected graph.

$$\frac{\begin{array}{c}\underline{\mathsf{item}}(x)\\ \underline{\mathsf{list}}(l)\end{array}}{\underline{\mathsf{list}}(x :: l)} \qquad\qquad \frac{\begin{array}{c}\underline{\mathsf{item}}(x)\\ \underline{\mathsf{last}}(y)\end{array}}{\begin{array}{c}\underline{\mathsf{last}}(x)\\ \mathsf{succ}(y,x)\end{array}}$$

Figure 4: Two different programs, each expressed in a single rule, for arbitrarily collecting items in a list. The list created by the left-hand rule is represented as a structured term; the list created by the right-hand rule is represented by the succ relation.

Correctness of this spanning tree algorithm follows from invariants maintained by the rules. Take V' to be the set of all x such that $\mathsf{intree}(x)$ holds, and take E' to be the the set of all ordered pairs (x, y) such that $\mathsf{tree}(x, y)$ holds. We have two state invariants, maintained by rule application:

1. E' is a subgraph of E and a spanning tree over V'.
2. The set V/V' is always the set of variables x' such that $\underline{\mathsf{vert}}(x')$ holds.

2.3 List and heap collection

The examples in Figures 4 and 5 are more imperative in nature; all involve collecting a multiset of items – represented by linear atomic propositions of the form $\underline{\mathsf{item}}(x)$ – arbitrarily into some data structure. The two separate programs in Figure 4 collect items into a list. The program on the left requires an additional input of the form $\underline{\mathsf{list}}(\mathsf{nil})$, and collects items into a list represented by a structured term: we use $x :: l$ as a shorthand for $\mathsf{cons}(x, l)$. The program on the right requires an additional input of the form $\underline{\mathsf{last}}(\mathsf{root})$, where root is some distinguished first element, then collects items into a list represented in the $\mathsf{succ}(x, y)$ relation in the database.

The program in Figure 5 requires no additional inputs, and collects items into a forest of binary-heap-like trees. Each member of the forest is represented as linear atomic propositions of the form $\underline{\mathsf{tree}}(n, t)$, where n is a natural number expressing the depth of the tree and t is a structured term representing the actual tree, a term consisting of an item and a list of subtrees.

These examples bring up another important property of linear/ephemeral propositions. For the purposes of bottom-up logic programming, deriving a persistent proposition twice is not any different than deriving it once; however, with linear propositions we are concerned with the *multiplicity* of those propositions: having two copies of $\underline{\mathsf{item}}(\mathsf{a})$ is different than having one. We will

$$\frac{\mathsf{item}(x)}{\mathsf{tree}(\mathsf{z}, \mathsf{node}(x, \mathsf{nil}))} \qquad \frac{\mathsf{tree}(n, \mathsf{node}(x, ts)) \quad \mathsf{tree}(n, t)}{\mathsf{tree}(\mathsf{s}(n), \mathsf{node}(x, t :: ts))}$$

Figure 5: Arbitrarily collecting items in a forest of binary-heap-like trees (right).

ensure that we can unambiguously refer to linear propositions by labeling them uniquely. For instance, the list-collection example on the left side of Figure 4 could, from the multiset of atomic propositions $\{l_0 : \mathsf{list}(\mathsf{nil}), l_1 : \mathsf{item}(\mathsf{a}), l_2 : \mathsf{item}(\mathsf{a}), l_3 : \mathsf{item}(\mathsf{b})\}$, derive $\mathsf{list}(\mathsf{a} :: \mathsf{b} :: \mathsf{a} :: \mathsf{nil})$ and $\mathsf{list}(\mathsf{a} :: \mathsf{a} :: \mathsf{b} :: \mathsf{nil})$, but not $\mathsf{list}(\mathsf{a} :: \mathsf{a} :: \mathsf{a} :: \mathsf{nil})$, because there are only two linear resources $\mathsf{item}(\mathsf{a})$ and the derivation of that proposition requires three such resources. Committed choice comes into play, and requires that we only be interested in one of the three possible result lists here; in general, committed choice means that execution will only provide one of the $n!$ possible resulting lists when starting from n distinct elements.

These examples demonstrate the power of linear logic to express algorithms that have proven difficult or inelegant to code in other systems resources (with the notable exception of [7]). In the next section, we will make the description here more formal by defining an operational semantics based on linear logic and a cost semantics that allows us to reason about running time and complexity without knowing the details of an interpreter for the language.

3 Semantics of linear logical algorithms

In this section, we will develop the tools for reasoning about the the algorithms we began to specify in the previous section. Two of the fundamental properties of an algorithm are its run time behavior, specified by an operational semantics, and its running time, specified by a cost semantics. We will describe both.

We have already presented a number of programs, but we will formally define a program $\mathcal{P}$ as a series of rules. Each rule has one or more atomic propositions $A_1, \ldots, A_n$ as premises and zero or more atomic propositions $C_1, \ldots, C_m$ as conclusions; for example, in clause $r2$ of Figure 3, $A_1 = \mathsf{vert}(\mathsf{root})$ and $C_1 = \mathsf{intree}(\mathsf{root})$. There are two additional restrictions on the form of rules:

- **Range restriction**. The free variables in the conclusion must be a subset of the free variables in the premises; this ensures that a ground database will remain ground when inference rules are applied.

- **Separation**. The program must consistently identify some propositions as linear and some as persistent; this was indicated before by writing linear propositions as prop and persistent propositions as prop. Separation also requires that in any rule with linear atomic propositions among the conclusions $C_1, \ldots, C_m$, at least one of the premises $A_1, \ldots, A_n$ must be a linear atomic proposition. This requirement helps ensure that we will not "flood" the database with unlimited copies of ephemeral propositions, and also allows us to implement the saturation function Clo effectively.

3.1 Operational semantics

In this section, we describe an operational semantics for the language we have defined. This operational semantics does not make sense as an implementation, as it makes transparently bad choices like running saturating forward chaining redundantly. The input is a finite *initial state* $\langle \Gamma_0, \Delta_0 \rangle$ where Γ_0 is a set of persistent propositions and Δ_0 is a set of labeled linear propositions; a *program trace* is a finite list of states $\langle \Gamma_0, \Delta_0 \rangle \ldots \langle \Gamma_m, \Delta_m \rangle$.

For each state $\langle \Gamma_i, \Delta_i \rangle$, the operational semantics calculates the saturated database $\mathsf{Clo}(\Gamma_i)$ of all the persistent atomic facts that are implied by Γ_i in $\mathcal{P}$ by exhaustive forward reasoning, disregarding any rules with linear propositions among the premises. Assuming the process of saturated forward inference terminates, the operational semantics picks an arbitrary rule $r \in \mathcal{P}$ that has at least one ephemeral premise and a grounding substitution σ (that is, a substitution that maps every free variable x in the rule to a variable-free term t) such that, for each premise A_i of the rule r, $A_i\sigma$ is in $\mathsf{Clo}(\Gamma_i)$ (if it is persistent) or Δ_i (if it is linear). Applying that rule removes one or more linear resources from Δ_i and adds each conclusion $C_i\sigma$ to either Γ_i or Δ_i depending on whether C_i is linear or persistent. This results in a new state $\langle \Gamma_{i+1}, \Delta_{i+1} \rangle$, and the trace is extended. If, at any point, there is no rule r and substitution σ satisfying the conditions described above, then the trace cannot be extended and is called a *complete* trace.

3.1.1 Termination

The operational semantics separates treatment of monotonic deduction that involves only persistent propositions and the committed choice reasoning that involves consuming ephemeral propositions. This distinction will be reflected in the definition of abstract running time, but we can already see that it is reflected in the arguments about the *termination* of algorithms. It was mentioned in Section 1.1 that we have to give an argument that the closure will be finite, as this is not true in general; we also have to give a termination argument bounding the length of the program trace by bounding the number of possible applications of rules with linear premises. Separation ensures that we only need to reason about the rules with only persistent premises in order to argue that Clo is total, and in most cases we only need to reason about the rules with some ephemeral premises in order to argue that the length of the trace is bounded.

In all the programs in Section 2, the argument that all traces are bounded is simple; in most examples every rule strictly decreases the number of ephemeral propositions, so that the number of states in a complete trace is bounded by the size of the initial set of labeled linear propositions Δ_0. The one exception to this is the first rule in Figure 5; this rule consumes one linear proposition $\mathsf{item}(x)$ and creates one linear proposition $\mathsf{tree}(\mathsf{z}, \mathsf{node}(x, \mathsf{nil}))$. In this case, we may assign a positive "weight" to each proposition and observe that if the weight of item is greater than the weight of tree, then the sum of the weights of all linear propositions in Δ decreases with the application of each linear rule.

3.2 Linear logic

We will sketch the description of the language and operational semantics in terms of intuitionistic linear logic; our system is a fragment of the judgmental reconstruction of first order intuitionistic linear logic described in [2, 3]. The necessary fragment of linear logic is roughly analogous to the Horn fragment of standard intuitionistic logic.

Atomic propositions	A, B, C		
Basic propositions	Q	$::=$	$A \mid !A$
State propositions	S	$::=$	$Q \mid \mathbf{1} \mid S \otimes S$
State transitions	R	$::=$	$S \mid S \multimap R \mid \forall x.R$
Persistent hypotheses	Γ	$::=$	$\cdot \mid \Gamma, R\ pers \mid \Gamma, A\ pers$
Ephemeral hypotheses	Δ	$::=$	$\cdot \mid \Delta, R\ eph \mid \Delta, A\ eph$

We highlight both $R\ pers$ and $A\ pers$ (and likewise for eph) because, when we describe sequents later on, we will often tacitly omit the rules from the program that are assumptions of the form $R\ pers$ and only refer to the atomic facts in the database, either $A\ pers$ or $A\ eph$.

The translation of a rule r with premises $A_1, \ldots, A_n$ and conclusions $C_1, \ldots, C_m$ is the persistent proposition

$$r : \forall \mathbf{x_1}.Q_1 \multimap \ldots \multimap \forall \mathbf{x_n}.Q_n \multimap (Q'_1 \otimes \ldots \otimes Q'_m)$$

with $Q_i = A_i$ if A_i is an ephemeral atomic proposition and $Q_i = !A_i$ if A_i is a persistent atomic proposition; similarly for $Q'_i = C_i$ if C_i is an ephemeral atomic proposition and $Q'_i = !C_i$ if C_i is a persistent atomic proposition. The separation between ephemeral and persistent in our language is not enforced by linear logic, but separation is required in order to allow us to reason about the Clo operation only in terms of the rules without linear premises. The $\forall \mathbf{x_i}$ are shorthand for zero or more $\forall x_{i0}.\forall x_{i1} \ldots$ where the x_{ij} are the variables that appear in the premise A_i but not in any of the premises $A_1, \ldots, A_{i-1}$.

Judgments in the sequent calculus presentation of intuitionistic linear logic have the form $\Gamma; \Delta \vdash R\ eph$; we write $R\ pers$ to write the judgment that R is persistent, and we write $R\ eph$ to write the judgment that R is ephemeral (or linear).

3.2.1 Focused derivations

The concepts of *polarity* and *focusing* as described in [4] are useful in describing logic programming from a proof theoretic perspective. In particular, focusing allows us to define *derived rules* in linear logic for any formula in the fragment described above. If we have a rule with premises $\underline{\mathsf{a}}$ and b, and with conclusions $\underline{\mathsf{c}}$ and d, we express that rule in linear logic as $\underline{\mathsf{a}} \multimap !\mathsf{b} \multimap (\underline{\mathsf{c}} \otimes !\mathsf{d})$. If we treat every atomic proposition as having positive polarity, focusing on this persistent proposition give us this derived inference rule:

$$\frac{\Gamma; \cdot \vdash \mathsf{b}\ eph \qquad \Gamma, \mathsf{d}\ pers; \Delta, \underline{\mathsf{c}}\ eph \vdash \gamma}{\Gamma; \Delta, \underline{\mathsf{a}}\ eph \vdash \gamma}$$

where γ is an arbitrary conclusion.

Theorem 1 below proves that $\Gamma; \cdot \vdash A\ eph$ if and only if $A \in \mathsf{Clo}(\Gamma)$, which gives us license to rewrite this rule as follows:

$$\frac{\mathsf{b} \in \mathsf{Clo}(\Gamma) \qquad \Gamma, \mathsf{d}\ pers; \Delta, \underline{\mathsf{c}}\ eph \vdash \gamma}{\Gamma; \Delta, \underline{\mathsf{a}}\ eph \vdash \gamma}$$

What we see from these rules is that our "next state" actually appears in the *premise* of the derived rule; this may seem a bit unnatural, but it is consistent with the formalization of algorithms in CLF, Lollimon [10, 17], and other linear logic programming languages.

To demonstrate, we can express the derivation that represents finding a spanning tree over the complete graph with vertices root, a, and b using the algorithm in Figure 3. We abbreviate $A\ eph$ and $A\ pers$ as simply A, and additionally abbreviate $\Gamma_0 = \mathsf{edge}(\mathsf{a}, \mathsf{root}), \mathsf{edge}(\mathsf{b}, \mathsf{root}), \mathsf{edge}(\mathsf{a}, \mathsf{b})$, $\Gamma_1 = \Gamma_0, \mathsf{intree}(\mathsf{root})$, and $\Gamma_2 = \Gamma_1, \mathsf{intree}(\mathsf{a}), \mathsf{tree}(\mathsf{root}, \mathsf{a})$. Our derivation using derived rules looks like this:

$$\dfrac{\dfrac{\mathsf{edge}(\mathsf{root}, \mathsf{a}) \in \mathsf{Clo}(\Gamma_1) \quad \mathsf{intree}(\mathsf{root}) \in \mathsf{Clo}(\Gamma_1) \qquad \dfrac{\mathsf{edge}(\mathsf{a}, \mathsf{b}) \in \mathsf{Clo}(\Gamma_2) \quad \mathsf{intree}(\mathsf{a}) \in \mathsf{Clo}(\Gamma_2) \qquad \dfrac{\vdots}{\Gamma_2,\ \mathsf{intree}(\mathsf{b}),\ \mathsf{tree}(\mathsf{a}, \mathsf{b}); \cdot \vdash \gamma}}{\Gamma_1,\ \mathsf{intree}(\mathsf{a}),\ \mathsf{tree}(\mathsf{root}, \mathsf{a});\ \underline{\mathsf{vert}}(\mathsf{b}) \vdash \gamma}}{\Gamma_0,\ \mathsf{intree}(\mathsf{root});\ \underline{\mathsf{vert}}(\mathsf{a}), \underline{\mathsf{vert}}(\mathsf{b}) \vdash \gamma}}{\Gamma_0;\ \underline{\mathsf{vert}}(\mathsf{a}),\ \underline{\mathsf{vert}}(\mathsf{b}),\ \underline{\mathsf{vert}}(\mathsf{root}) \vdash \gamma}$$

3.2.2 Operational semantics and focused derivations

While we omitted details of linear logic and focusing that are described elsewhere, we can still state the soundness and (non-deterministic) completeness of our language with respect to linear logic.

First, we must be precise about what we mean by $\mathsf{Clo}_{\mathcal{P}}(\Gamma)$ when $\mathcal{P}$ contains persistent and ephemeral propositions. We translate only the rules in $\mathcal{P}$ where all premises $A_1, \ldots, A_n$ and all conclusions $C_1, \ldots, C_m$ are persistent into formulas in intuitionistic logic as follows:

$$\forall \mathbf{x_1}.A_1 \supset \ldots \supset \forall \mathbf{x_n}.A_n \supset (C_1 \wedge \ldots \wedge C_m)$$

$\mathsf{Clo}_{\mathcal{P}}(\Gamma)$ then has the property that, if it is finite (and therefore its calculation terminates), $A \in \mathsf{Clo}_{\mathcal{P}}(\Gamma)$ if and only if $\mathcal{P}, \Gamma \vdash A\ true$, where $\mathcal{P}$ is the appropriate translation of the rules of $\mathcal{P}$ that have only persistent premises.

As in the linear logic translation, we interpret these translated rules by way of focusing, but we give every atom *negative* polarity instead of positive polarity. This is mostly for the sake of convenience in some of the proofs in Section 4 dealing with the correctness of the program translation described there; the meaning of Clo is the same whether atoms are given positive or negative polarity.

The derived rules are correspondingly different. The derived rules for the clause $\mathsf{a} \supset \mathsf{b} \supset (\mathsf{c} \wedge \mathsf{d})$ are:

$$\frac{\Gamma \vdash \mathsf{a}\ true \quad \Gamma \vdash \mathsf{b}\ true}{\Gamma \vdash \mathsf{c}\ true} \qquad \frac{\Gamma \vdash \mathsf{a}\ true \quad \Gamma \vdash \mathsf{b}\ true}{\Gamma \vdash \mathsf{d}\ true}$$

Separation allows us to prove the following essential lemma:

Theorem 1 (Separation and closure)**.** *For any (separated and range-restricted) program $\mathcal{P}$, let $\mathcal{P}^*$ be the subset of rules that contain only persistent propositions. $A \in \mathsf{Clo}_{\mathcal{P}}(\Gamma)$ if and only if $\mathcal{P}, \Gamma; \cdot \vdash A\ eph$; that is, assuming Γ contains only persistent atomic propositions, $\mathcal{P}^*, \Gamma \vdash A\ true$ if and only if $\mathcal{P}, \Gamma; \cdot \vdash A\ eph$.*

Proof. The "only if" direction is a straightforward derivation on focused derivations of $\Gamma \vdash A\ true$; the "if" direction is the interesting one. Proof proceeds by induction over focused derivations of $\Gamma; \Delta \vdash A\ eph$. We have to consider the case where $\Gamma, A; \cdot \vdash A\ eph$ is an initial sequent; in this case, $\Gamma, A \vdash A\ true$ is also an initial sequent. Otherwise, we have a case for each derived rule in $\mathcal{P}$.

All rules with linear premises result in a derived rule that requires a linear atomic proposition to be immediately present. These rules can then all be excluded: the propositions cannot be present in Δ because it is empty, and cannot be in Γ because it contains only *persistent* atomic propositions. All derived rules with only persistent premises have (by separation) only persistent conclusions, and so their derived rule takes the following form, for any σ that substitutes a variable-free term for every free variable in $A_1, \ldots, A_n$:

$$\frac{\Gamma; \cdot \vdash A_1\sigma\ eph \quad \ldots \quad \Gamma; \cdot \vdash A_n\sigma\ eph \qquad \Gamma, C_1\sigma, \ldots, C_m\sigma; \Delta \vdash B\ eph}{\Gamma; \Delta \vdash B\ eph}$$

We need to show $\Gamma \vdash B\ true$. By the induction hypothesis, we have $\Gamma \vdash A_i\ true$ for all A_i, and because the derived rule that we applied had, by separation, only persistent conclusions, $\Gamma, C_1\sigma, \ldots, C_n\sigma$ still contains only persistent atomic propositions, and therefore we can apply the induction hypothesis to get $\Gamma, C_1\sigma, \ldots, C_n\sigma \vdash B\ true$. The derived rules of the same rule into intuitionistic logic takes the following form for each C_i:

$$\frac{\Gamma \vdash A_1\sigma\ true \quad \ldots \quad \Gamma \vdash A_n\sigma\ true}{\Gamma \vdash C_i\sigma\ true}$$

We therefore have $\Gamma \vdash C_i\sigma\ true$ by these derived rules for each C_i, and so we have $\Gamma \vdash B\ true$ by the induction hypothesis and repeated appeals to cut elimination. □

Separation plays a crucial rule at two points in the proof. First, we know we cannot apply a rule with linear premises because the translation requires that these linear premises be immediately available, but they cannot be in Γ by an explicit condition and they cannot be in Δ because it is empty. Second, because rules with only persistent premises have only persistent conclusions, we know that nothing is added to the (empty) linear context Δ and that only persistent atomic propositions are added to Γ in the smaller derivation, allowing the appeal to the induction hypothesis.

Theorem 2 (Soundness of operational semantics)**.** *For any (separated and range-restricted) program $\mathcal{P}$ and for any program trace $\langle \Gamma_0, \Delta_0 \rangle \ldots \langle \Gamma_m, \Delta_m \rangle$, given a sequent of the form $\Gamma_m; \Delta_m \vdash \gamma$ for an arbitrary γ, there exists a derivation of $\Gamma_0; \Delta_0 \vdash \gamma$.*

Proof. By induction on the length of the abstract trace and Theorem 1. □

Aside from a the somewhat counter-intuitive change in direction brought on by "working on the left," the statement of soundness is relatively straightforward. The statement of non-deterministic completeness is a bit more subtle, in part because the abstract algorithm is not non-deterministically complete in the way that a top-down logic programming language like Prolog is. In pure Prolog, an attempt to prove A will either succeed, fail finitely, or not terminate; if search fails finitely, we know A was not provable under the rules. In our system, non-deterministic completeness means only that if a derivation representing a trace is provable then there is some execution of the abstract operational semantics that follows this trace; it may not be the only possible trace, and a given implementation might always follow some other trace.

Theorem 3 (Nondeterministic completeness of operational semantics)**.** *For any (separated and range-restricted) program $\mathcal{P}$, if the sequent $\Gamma_0; \Delta_0 \vdash \gamma$ is derivable using the sequent $\Gamma_m; \Delta_m \vdash \gamma$, where Γ_0, Δ_0, Γ_m, and Δ_m contain only ground, atomic propositions and γ is an arbitrary conclusion, then there exists some program trace $\langle \Gamma_0, \Delta_0 \rangle \ldots \langle \Gamma'_m, \Delta_m \rangle$ where $\mathsf{Clo}(\Gamma_m) = \mathsf{Clo}(\Gamma'_m)$.*

Proof. By induction on focused derivations and Theorem 1. □

Theorem 3 says that if we can "work on the left" in linear logic from a sequent $\Gamma_0; \Delta_0 \vdash \gamma$ to a sequent $\Gamma_m; \Delta_m \vdash \gamma$, then some trace obeying the operational semantics follows an equivalent path; however, because the operational semantics allows an arbitrary choice of which applicable rule with linear premises to apply, a correct implementation of the operational semantics might never take such a path. We could additionally consider a notion of a *fair* implementation that, at any state $\langle \Gamma_i, \Delta_i \rangle$ where multiple rules with linear inferences can be applied, has a non-zero probability of applying each of them, but that is beyond the scope of this paper.

3.3 Cost semantics

We define, following Ganzinger and McAllester [5, 6], a cost semantics called the *abstract running time*. This cost semantics will allow us to reason about algorithms written in this language, such as the ones in in Section 2, without considering the details of the implementation described in Section 4. The abstract running time of a trace $\langle \Gamma_0, \Delta_0 \rangle \ldots \langle \Gamma_m, \Delta_m \rangle$ is the sum of four components: $|\Gamma_0| + |\Delta_0| + m + \Phi$. The first two components, $|\Gamma_0|$ and $|\Delta_0|$, are just the number of persistent and linear resources (respectively) given as input. The other two components are m, the number of transitions involving rules with linear premises, and Φ, the number of *unique prefix firings* – a quantity we will now define.

Definition 1 (Prefix firing)**.** *Let $\langle \Gamma_0, \Delta_0 \rangle \ldots \langle \Gamma_m, \Delta_m \rangle$ be a program trace of a program $\mathcal{P}$. A* prefix firing *is a triple $\langle r, \sigma, [l_1, \ldots, l_k] \rangle$ such that*

- *There is a rule r in $\mathcal{P}$ with premises $A_1, \ldots, A_n$.*
- *The substitution σ assigns a ground term for every free variable in the premises $A_1, \ldots, A_k$.*
- *There is some state $\langle \Gamma_i, \Delta_i \rangle$ where for all $0 \leq j < k$, either $A_j\sigma \in \mathsf{Clo}(\Gamma_i)$ or $l_j : A_j\sigma \in \Delta_i$, and either*

- *All of $A_1, \ldots, A_k$ are persistent atomic propositions, or else*
- *$k < n$ and there is* **no** *substitution σ' that assigns the same terms as σ to the free variables in $A_1, \ldots, A_k$ and additionally assigns ground terms to all the free variables in A_k such that $A_k\sigma' \in \mathsf{Clo}(\Gamma_i)$ or such that $l : A_k\sigma' \in \Delta_i'$, where Δ_i' is Δ_i with all the linear propositions labeled $l_1, \ldots, l_k$ removed.*

As mentioned previously, if multiple instances of the ground linear proposition appear in Δ_i, they have distinct labels and can be used to form distinct prefix firings. Because we don't care about labels of persistent atomic propositions, and the definition doesn't use them, we write them as an underscore "_".

The majority of the definition just expresses the fact that the order of premises matters; the last bullet point is the complicated one. It describes the conditions where we can ignore would-be prefix firings that include linear propositions; we can do so if we know that, in every state such that all the linear elements of the prefix are present and all persistent elements of the prefix are derivable, we will always be able to expand the would-be prefix firing to a larger one. In the next section, we will give some examples of using the abstract running time to analyze algorithms.

3.4 Using the abstract running time

Because the operational semantics is quite non-deterministic, and because our cost semantics depends on the number of applications of rules with linear premises, we can expect reasoning in general about the running time of programs to be undecidable. However, for well-designed programs it is usually still possible to effectively reason about both the number of prefix firings and the length of the program trace in order to get an informative abstract running time. We give some examples in this section.

3.4.1 Spanning trees

We will show that the spanning tree algorithm in Figure 3 has an abstract running time in $O(|E| + |V|)$, that is, proportional to the number of edges plus the number of vertices. The abstract running time is $|\Gamma_0|+|\Delta_0|+m+\Phi$. It is obvious that $|\Gamma_0| = |E|$ and $|\Delta_0| = |V|$ based on how the problem is set up; also, because every linear transition consumes a linear resource corresponding to some $v \in V$, an abstract trace can have at most $|V|$ transitions, which is to say that m is bounded by $|V|$.

We consider the prefix firings for each of the three rules in turn. Rule $r1$ can have at most $2|E|$ prefix firings, as every edge $(\mathsf{a}, \mathsf{b}) \in E$ leads to two facts: $\mathsf{edge}(\mathsf{a}, \mathsf{b})$ and $\mathsf{edge}(\mathsf{b}, \mathsf{a})$. Rule $r2$ has no prefix firings; it has as a premise one linear proposition that is either present in the set of linear resources or is not. Rule $r3$ can have at most $4|E|$ prefix firings. The first premise $\mathsf{edge}(x, y)$ effectively "grounds" the rest of the premises and leads to $2|E|$ prefix firings of the form $\langle r3, \sigma, [\_]\rangle$. The final state will include $\mathsf{intree}(\mathsf{a})$ for every vertex a, resulting in at most $2|E|$ prefix firings of the form $\langle r3, \sigma, [\_, \_]\rangle$. However, there are no prefix firings of the form $\langle r3, \sigma, [\_, \_, l]\rangle$, because a prefix firing that covers all the premises does not meet the condition that $k < n$. This bounds the number of prefix firings by $6|E|$, which in turn bounds the abstract running time bounded by $2|V| + 7|E|$, so the abstract running time is in $O(|E| + |V|)$.

This example also gives us a good example of why the ordering of premises matters. If we had reversed the order of the three premises of rule $r3$ in Figure 3, then there would have been $|V|$ premises of the form $\langle r3, \sigma, [l]\rangle$, because in the first state there are $|V|$ propositions vert(x) and no propositions of the form intree(x). Furthermore, if the graph is sparse, there may be on the order of $|V|^2$ prefix firings of the form $\langle r3, \sigma, [l, \_]\rangle$, because for any two points in the graph, there must be a state in the abstract trace when one of them has been added to the graph and one has not, and because of that, for any two points there is a unique prefix firing of the form $\langle r3, \sigma, [l, \_]\rangle$ if there is no direct edge between those two points.

3.4.2 List collection

We can also show that the list collection examples in Figure 4 always has an abstract running time in $O(n)$, where n is the number of items given as input. The algorithms specify that an ephemeral proposition of the form list(nil) (alternatively last(root)) is present in Δ_0, and every time we remove a proposition of the form list(x) (alternatively last(x)), we add another one back. Therefore, in every state, we will either have no ephemeral propositions of the form item(x), in which case we obviously cannot form any prefix firings, or else we will be prevented from forming a prefix firing of the form $\langle r, \sigma, [l]\rangle$ by the presence of a ephemeral atomic proposition of the form list(x) (alternatively last(x)) that allows the prefix to be extended. Because each step consumes some resource item(x), the number of transitions m is equal to n, the number of items being collected. Overall, we have $|\Gamma_0| = 0$, $|\Delta_0| = n + 1$, $m = n$, and $\Phi = 0$; the abstract running time is $2n + 1$, which is in $O(n)$.

This example may seem simplistic, but it is worth mentioning because an equivalent program written in the language of [6] has an abstract running time that is in $O(n^2)$ and not in $O(n)$. One explanation of this is that the concept of deletion used in that work is not tuned towards representing stateful change as well as linear logic. Another important point is that our analysis relies on the strict separation of persistent and ephemeral propositions, which is not a distinction in the language described in that paper.

3.4.3 Heap collection

Finally, we present an analysis of the heap-collection example in Figure 5. There are two rules; the first will have no prefix firings, and the second will have some number of prefix firings of the form $\langle r, \sigma, [l]\rangle$; at most, there will be one of these for every tree created over the course of execution, as every tree could potentially exist in some state where no other trees of the same size are present.

If we look at the number of transitions in terms of the number of times each individual element (of the n elements, represented by linear propositions of the form item(x) in the initial state) can make a transition, we can see that it is bounded by $\log(n) + 1$ - one transition involving the first rule, and $\lfloor \log(n) \rfloor$ transitions involving the second rule, because each transition with the second rule doubles the size of the tree, and the tree can get no larger than n. This means there can be at most $n \log(n) + n$ transitions. Since each transition creates a single resource of the form tree(n, t), this also means that the number of unique trees, and therefore the number of unique prefix firings,

is bounded by $n\log(n) + n$. We conclude that the abstract running time for the algorithm is in $O(n\log(n))$.

4 Implementing linear logical algorithms

Having presented abstract running time, which gives a cost semantics for our algorithms, we must finally show that the notion of abstract running time that we have defined is based in reality by describing an interpreter for running programs written in our language that runs in time proportional to the abstract running time. The important theorem is the following one:

Theorem 4 (Running time)**.** *For any terminating program $\mathcal{P}$, there exists an interpreter running on a RAM machine extended with constant time hash table operations such that for any initial state $\langle\Gamma_0, \Delta_0\rangle$ the interpreter executes a complete trace $\langle\Gamma_0, \Delta_0\rangle, \ldots, \langle\Gamma_m, \Delta_m\rangle$ and returns $\mathsf{Clo}(\Gamma_m)$ and Δ_m in time proportional to the abstract running time of the trace.*

Theorem 4 establishes that reasoning about the behavior of algorithms described in the language we have presented is a three-part process. First, we must demonstrate that, for a given program, $\mathsf{Clo}(\Gamma)$ is always finite and that no trace of the operational semantics can have unbounded length. Second, we must give a bound to the abstract running time of *all* possible complete traces in terms of the initial state. Having shown that $\mathcal{P}$ will always terminate, Theorem 4 ties the knot by ensuring that the implementation will execute one of the possible complete traces and will do so in time proportional to the abstract running time of that trace. Because we have bounded the abstract running time of all traces, we know that the arbitrary trace selected by the interpreter has an abstract running time within that bound and is therefore executed in time proportional to that bound.

4.1 Overview

We will outline the interpreter that establishes Theorem 4 here, and then describe it in detail in the rest of this section. Given a program, we create a derived program where for each rule r with n premises in the original program, the derived program has $2n$ rules and introduces $2n$ new atomic propositions (referred to as derived propositions), one for each premise A_i and one for each prefix $A_1, \ldots, A_n$. The interpreter uses three important data structures. The first is an index INDEX that allows the discovery of all atomic propositions that share a certain ground term, and the second two are worklists (queues) that together contain all the immediate consequences of the propositions in INDEX. The first, QUEUE, handles reasoning with with persistent propositions and the second, PQUEUE, handles reasoning with ephemeral propositions.

The portion of the interpreter dealing with purely persistent propositions is similar to the interpreter in [11]. When a fact is removed from the persistent work queue and added to the INDEX, the index is used to find all immediate consequences of that proposition and all those propositions already in INDEX; these consequences are then added to the queue. The treatment of linear atomic propositions is novel. The index and linear work queues are allowed to temporarily contain multiple derived propositions that are all consequences of Δ_i, the current multiset of non-derived atomic

propositions, even if some cannot simultaneously be consequences of Δ_i because they require consuming the same ephemeral propositions. These ephemeral propositions are only consumed when a rule from the original program is applied, in the process removing all the derived propositions that depended on the consumed propositions.

In order to avoid declaring and then deleting atomic propositions, upon removing an item from the linear work queue, the index is repeatedly used to find the *first* atomic proposition implied by the program, and then find the first atomic proposition implied by that proposition, and so on. Either this will succeed until we have shown that every premise of the a rule from the original program is currently available, in which case we apply that rule, or it will fail, in which case backtracking, depth-first search looks for a way to fully apply the rule. Each failure corresponds to a prefix that cannot be extended, so the time required for a successful search is represented by the presence of the term m, the number of linear transitions, in the abstract running time, and the time required for a failed search is represented by a unique prefix firing resulting from non-extendable prefixes that include linear propositions.

4.2 Compiling programs

We define a program transformation on the rule r with premises $A_1, \ldots, A_n$ and conclusions $C_1, \ldots, C_m$ that is run on every rule in the program $\mathcal{P}$ to produce the compiled program $\mathcal{R}$. For each premise A_i in a rule r, the transformation creates two unique predicate symbols $\mathsf{spec_{ri}}$ and $\mathsf{rule_{ri}}$ (which may alternatively be predicate symbols for linear atomic propositions, in which case we say $\underline{\mathsf{spec_{ri}}}$ and $\underline{\mathsf{rule_{ri}}}$), two function symbols $\mathsf{f_{ri}}$ and $\mathsf{g_{ri}}$, and two rules, a *specialization* rule and an *execution* rule – similarly, we call the introduced atomic propositions specialization and execution propositions.

- The specialization rule for the premise A_i of the rule r is a rule s_{ri} with premise A_i and conclusion S_{ri}
  - If A_i is persistent, S_{ri} is persistent and had the form $\mathsf{spec_{ri}}(\mathsf{g_{ri}}(\mathbf{x_i}), \mathbf{z_i})$.
  - If A_i is linear, S_{ri} is linear and has the form $\underline{\mathsf{spec_{ri}}}(\mathsf{g_{ri}}(\mathbf{x_i}), \mathbf{z_i})$.
  - The variables $\mathbf{x_i}$ are all variables that are in A_i and also in $A_1, \ldots, A_{i-1}$.
  - The variables $\mathbf{z_i}$ are all variables that are in A_i that are *not* in $A_1, \ldots, A_{i-1}$.
- The execution rule for the premise A_i of the rule r is a rule e_{ri} with premises S_{ri} and E_{ri} and conclusion $E_{r(i+1)}$ (if $i < n$) or $C_1, \ldots, C_n$ (if i = n).
  - If $i = 1$, or if $E_{r(i-1)}$ and $S_{r(i-1)}$ are both persistent, then E_{ri} is persistent and has the form $\mathsf{exec_{ri}}(\mathsf{f_{ri}}(\mathbf{x_i}), \mathbf{y_i})$; otherwise, E_{ri} is linear and has the form $\underline{\mathsf{exec_{ri}}}(\mathsf{f_{ri}}(\mathbf{x_i}), \mathbf{y_i})$.
  - S_{ri} is defined as above and has the form $\mathsf{spec_{ri}}(\mathsf{g_{ri}}(\mathbf{x_i}), \mathbf{z_i})$ or $\underline{\mathsf{spec_{ri}}}(\mathsf{g_{ri}}(\mathbf{x_i}), \mathbf{z_i})$.
  - The variables $\mathbf{x_i}$ and $\mathbf{z_i}$ are defined as above.
  - The variables $\mathbf{y_i}$ are all the variables that are in $A_1, \ldots, A_{i-1}$; this is a superset of $\mathbf{x_i}$,.

We call the first term $\mathsf{f}_{\mathsf{ri}}(\mathbf{x})$ or $\mathsf{g}_{\mathsf{ri}}(\mathbf{x})$ in in the newly introduced rules the *index term*. For each rule r in $\mathcal{P}$ we an add to the program an additional rule $\mathsf{exec}_{\mathsf{r0}}(\mathsf{f}_{\mathsf{r0}})$ to the compiled program $\mathcal{R}$ that has no premises.

The intuition behind the execution propositions E_{ri} and specialization propositions S_{ri} is that each way that we we can to simultaneously derive both $E_{ri}\sigma$ and $S_{ri}\sigma$ for some substitution σ that substitutes ground variables for all the free variables in E_{ri} and S_{ri}, corresponds to a prefix firing $\langle r, \sigma, [l_1, \ldots, l_i]\rangle$.

4.2.1 Examples

To conserve space, we will write out rules in a horizontal style as "rule : premises / conclusion" in this section. The rule *play* from the single-elimination-tournament program in Figure 2 turns into five clauses in the derived program – we refer to the rule as p instead of *play*. The index term $\mathsf{f}_{\mathsf{p1}}(n)$ represents the necessity that only two teams with the same number of wins play each other.

e_{p} : / $\mathsf{exec}_{\mathsf{p1}}(\mathsf{f}_{\mathsf{p1}})$
s_{p1} : $\underline{\mathsf{wins}}(x, n)$ / $\underline{\mathsf{spec}_{\mathsf{p1}}}(\mathsf{f}_{\mathsf{p1}}, x, n)$
s_{p2} : $\underline{\mathsf{wins}}(y, n)$ / $\underline{\mathsf{spec}_{\mathsf{p2}}}(\mathsf{g}_{\mathsf{p2}}(n), y)$
e_{p2} : $\mathsf{exec}_{\mathsf{p1}}(\mathsf{f}_{\mathsf{p1}})$, $\underline{\mathsf{spec}_{\mathsf{p1}}}(\mathsf{g}_{\mathsf{p1}}, x, n)$ / $\underline{\mathsf{exec}_{\mathsf{p2}}}(\mathsf{f}_{\mathsf{p2}}(n), x, n)$
e_{p1} : $\underline{\mathsf{exec}_{\mathsf{p1}}}(\mathsf{f}_{\mathsf{p2}}(n), x, n)$, $\underline{\mathsf{spec}_{\mathsf{p1}}}(\mathsf{g}_{\mathsf{p2}}(n), y)$ / $\underline{\mathsf{wins}}(x, \mathsf{s}(y))$, $\mathsf{won}(x, y, n)$

The rule $r3$ in Figure 3 turns into seven clauses in the derived program – we refer to the rule as r instead of $r3$. The index terms reflect that the variables are all introduced in the first premise; also, because only the last premise is linear, we do not get any linear execution propositions.

e_{r} : / $\mathsf{exec}_{\mathsf{r1}}(\mathsf{f}_{\mathsf{r1}})$
s_{r1} : $\mathsf{edge}(x, y)$ / $\mathsf{spec}_{\mathsf{r1}}(\mathsf{f}_{\mathsf{r1}}, x, y)$
s_{r2} : $\mathsf{intree}(x)$ / $\mathsf{spec}_{\mathsf{r2}}(\mathsf{f}_{\mathsf{r2}}(x))$
s_{r3} : $\underline{\mathsf{vert}}(y)$ / $\underline{\mathsf{spec}_{\mathsf{r3}}}(\mathsf{f}_{\mathsf{r3}}(y))$
e_{r1} : $\mathsf{exec}_{\mathsf{r1}}(\mathsf{f}_{\mathsf{r1}})$, $\mathsf{spec}_{\mathsf{r1}}(\mathsf{f}_{\mathsf{r1}}, x, y)$ / $\mathsf{exec}_{\mathsf{r2}}(\mathsf{f}_{\mathsf{r2}}(x), x, y)$
e_{r2} : $\mathsf{exec}_{\mathsf{r2}}(\mathsf{f}_{\mathsf{r2}}(x), x, y)$, $\mathsf{spec}_{\mathsf{r2}}(\mathsf{f}_{\mathsf{r2}}(x))$ / $\mathsf{exec}_{\mathsf{r3}}(\mathsf{f}_{\mathsf{r3}}(y), x, y)$
e_{r3} : $\mathsf{exec}_{\mathsf{r3}}(\mathsf{f}_{\mathsf{r3}}(y), x, y)$, $\underline{\mathsf{spec}_{\mathsf{r3}}}(\mathsf{f}_{\mathsf{r3}}(y))$ / $\mathsf{tree}(x, y)$, $\mathsf{intree}(y)$

4.3 Correctness of the compiled program

The semantics of the compiled program are not precisely the same as the committed choice semantics of the original language; we will explain the relationship between the compiled and original programs by the proofs in this section. First, we relate the behavior of the closure Clo of the two programs, and then give Theorems 7 and 9 that describe the correspondence of the operational semantics of the original and compiled programs. Throughout, we will refer to a program $\mathcal{P}$ and its compiled variant $\mathcal{R}$.

4.3.1 Correctness of the persistent fragment

It is necessary to prove that if A is and ground atomic proposition from the original program (neither a specialization or execution proposition) and if Γ contains only ground atomic propositions from the original program, then $A \in \mathsf{Clo}_{\mathcal{P}}(\Gamma)$ if and only if $A \in \mathsf{Clo}_{\mathcal{R}}(\Gamma)$; this is an obvious corollary to the following theorem:

Theorem 5 (Correctness of the persistent compiled program)**.** *Given a program $\mathcal{P}$ and a compiled version of that program $\mathcal{R}$, where*

1. *B is a variable-free proposition from the original program,*
2. *Γ contains only variable-free atomic propositions from an original program,*

then $\mathcal{P}, \Gamma \vdash B\ true$ is derivable if and only if $\mathcal{R}, \Gamma \vdash B\ true$.

Proof. Both proofs proceed by induction over focused derivations. For the "only if" direction, we have a derivation of $\mathcal{P}, \Gamma \vdash B\ true$ and we need to show $\mathcal{R}, \Gamma \vdash B\ true$. If the derivation is an initial sequent, then the result is immediate. Otherwise, the last step of the derivation is a derived rule from some purely persistent rule $r \in \mathcal{P}$ with premises $A_1, \ldots, A_n$ and conclusions $C_1, \ldots, C_m$ under the substitution σ, and $B = C_i\sigma$ for some C_i, so that the derivation has this form:

$$\frac{\mathcal{P}, \Gamma \vdash A_1\sigma\ true \quad \ldots \quad \mathcal{P}, \Gamma \vdash A_n\sigma\ true}{\mathcal{P}, \Gamma \vdash C_i\sigma\ true}$$

By the derived rules of the program $\mathcal{R}$, we can then derive:

$$\dfrac{\dfrac{\dfrac{}{\mathcal{R}, \Gamma \vdash E_{r1}\ true} \quad \dfrac{\mathcal{R}, \Gamma \vdash A_1\sigma\ true}{\mathcal{R}, \Gamma \vdash S_{r1}\sigma\ true}}{\mathcal{R}, \Gamma \vdash E_{r2}\sigma\ true} \quad \cdots \quad \dfrac{\vdots}{\mathcal{R}, \Gamma \vdash E_{rn}\sigma\ true} \qquad \dfrac{\mathcal{R}, \Gamma \vdash A_n\sigma\ true}{\mathcal{R}, \Gamma \vdash S_{rn}\sigma\ true}}{\mathcal{R}, \Gamma \vdash C_i\sigma\ true}$$

This concludes the proof in the "only if" direction. The "if" direction is similar; the structure of any derivation ending with $\mathcal{R}, \Gamma \vdash C_i\sigma\ true$ is either initial or has the form:

$$\frac{\mathcal{R}, \Gamma \vdash E_{rn}\sigma\ true \quad \mathcal{R}, \Gamma \vdash S_{rn}\sigma\ true}{\mathcal{R}, \Gamma \vdash C_i\sigma\ true}$$

In this case, the shape of the derivation is uniquely determined up to smaller derivations $\mathcal{R}, \Gamma \vdash A_i\sigma\ true$, and the induction hypothesis can be applied to get $\mathcal{P}, \Gamma \vdash A_i\sigma\ true$ so that the rule r can be applied. This completes the case and the proof. □

We will require the following theorem in the next section: it relates the provability of a specialization or execution proposition to the provability of atomic propositions from the original program.

Theorem 6 (Specalization and execution propositions)**.** *If Γ contains atomic propositions (no execution or specialization propositions), $\mathcal{R}$ is the compiled version of program $\mathcal{P}$, $\mathcal{P}$ contains a rule r with premises $A_1, \ldots, A_n$, and conclusions $C_1, \ldots, C_m$, and σ is substitutes a variable-free term for every variable in $A_1, \ldots, A_n$, then:*

1. $\mathcal{R}, \Gamma \vdash S_{ri}\sigma\ true$ *if and only if* $\mathcal{R}, \Gamma \vdash A_i\sigma\ true$.
2. $\mathcal{R}, \Gamma \vdash E_{ri}\sigma\ true$ *if and only if* $\mathcal{R}, \Gamma \vdash A_j\sigma\ true$ *for every* $j < i$.

Proof. By case analysis of focused derivations; the first point about specialization predicates is immediate, and the second point about execution predicates is true by a straightforward construction in the "if" direction, but the "only if" direction requires induction on i. The base case, $i = 1$, is immediately true because E_{r1} is declared to be true in $\mathcal{R}$, and in the inductive case we have a derivation $\mathcal{R}, \Gamma \vdash E_{r(i+1)}\sigma\ true$ and have to show $\mathcal{R}, \Gamma \vdash A_i\sigma\ true$ for every A_i in $A_1\sigma, \ldots, A_i\sigma$.

That derivation could only have been derived if $\mathcal{R}, \Gamma \vdash E_{ri}\sigma\ true$, which means that which means $A_1\sigma, \ldots, A_{i-1}\sigma$ are true by the induction hypothesis, and also if $\mathcal{R}, \Gamma \vdash S_{ri}\sigma\ true$, which means $A_i\sigma$ is true by the first point, which completes the case. □

4.3.2 Correctness of the full language

Now, we describe the relationship between $\mathcal{P}$ and $\mathcal{R}$ in terms of the operational semantics. As was the case when we related the operational semantics to the semantics of linear logic, soundness is fairly straightforward, but completeness is more subtle.

Theorem 7 (Soundness of compiled program)**.** *If $\mathcal{T}_1 : \langle \Gamma, \Delta \rangle \ldots \langle \Gamma^*, \Delta^* \rangle$ is a trace under the program $\mathcal{P}$, then there exists a trace $\mathcal{T}_2 : \langle \Gamma, \Delta \rangle \ldots \langle \Gamma^*, \Delta^* \rangle$ under the compiled program $\mathcal{R}$.*

Proof. Structural induction on $\mathcal{T}_1$. If the trace only has one state, the result is immediate, otherwise, the last transition is by some rule r in $\mathcal{P}$ that has premises $A_1, \ldots, A_n$ and conclusions $C_1, \ldots, C_m$ under substitution σ.

This means that the last state $\langle \Gamma^*, \Delta^* \rangle = \langle (\Gamma', C_i\sigma, \ldots), (\Delta, C_j\sigma, \ldots) \rangle$, where $C_i\sigma, \ldots$ are the persistent conclusions of r and $C_j\sigma, \ldots$ are the ephemeral conclusions. The next to last state is then $\langle \Gamma', (\Delta', A_i\sigma, \ldots) \rangle$, where $A_i\sigma, \ldots$ are all the ephemeral premises in $A_1, \ldots, A_n$, and for every persistent A_j among the premises, $A_j\sigma \in \mathsf{Clo}_{\mathcal{P}}(\Gamma')$.

By the induction hypothesis we have a trace $\mathcal{T}$ under $\mathcal{R}$ starting with $\langle \Gamma, \Delta \rangle$ and ending in that last state. We can extend this trace to $\langle \Gamma^*, \Delta^* \rangle$ using rules from the derived program $\mathcal{R}$.

We can then construct a trace using rules from the derived program $\mathcal{R}$.

$\langle \Gamma, \Delta \rangle, \ldots$	$\langle \Gamma', (\Delta', A_i\sigma, \ldots) \rangle$,	
$\ldots$	$\langle \Gamma', (\Delta'', S_{ri}\sigma, \ldots) \rangle$,	Apply s_{rj} for every linear A_j
	$\langle \Gamma', (\Delta', E_{r(i+1)}\sigma, \ldots) \rangle$,	By rule e_{ri} ($E_{ri} \in \mathsf{Clo}_{\mathcal{R}}(\Gamma)$ by Theorem 6)
$\ldots$	either $\langle \Gamma', (\Delta', E_{rn}\sigma, S_{rn}\sigma) \rangle$	Apply e_{ri} for every $i < j < n$
	or else $\langle \Gamma', (\Delta', E_{rn}\sigma) \rangle$	
	$\langle \Gamma^*, \Delta^* \rangle$	By rule e_{rn}

Each step in the first omitted part of the trace is immediate, as we know from the fact that r was the last rule applied that $A_j\sigma \in \Delta$ for each A_j. In the second part of the trace, at every step either $E_{ri}\sigma$ and $S_{ri}\sigma$ are both present in the linear context, or else $E_{ri}\sigma$ is present and $S_{ri}\sigma \in \mathsf{Clo}_{\mathcal{R}}(\Gamma)$ by Theorem 6. □

The completeness theorem will require a permutation lemma; its proof is a straightforward induction on the length of a traces.

Theorem 8 (Permutation)**.** *If a trace under a compiled program $\mathcal{R}$ begins in a state $\langle \Gamma, \Delta \rangle$ containing no specialization or execution propositions, then*

1. *If the final state is $\langle \Gamma', (\Delta', S_{ri}\sigma) \rangle$, then there exists another trace of the same length that has the next-to-last state $\langle \Gamma', (\Delta', A_i\sigma) \rangle$.*

2. *If a trace starts in a state with no specialization or execution predicates and ends in a state $\langle \Gamma', (\Delta, E_{ri}\sigma) \rangle$, then there exists another trace of the same length that has next-to-last state with one of the following forms, depending on whether $E_{r(i-1)}$ or $S_{r(i-1)}$ are linear or persistent propositions:*

   - $\langle \Gamma', (\Delta, E_{r(i-1)}\sigma, S_{r(i-1)}\sigma) \rangle$
   - $\langle \Gamma', (\Delta, S_{r(i-1)}\sigma) \rangle$*, and additionally* $E_{r(i-1)}\sigma \in \mathsf{Clo}_{\mathcal{R}}(\Gamma')$
   - $\langle \Gamma', (\Delta, E_{r(i-1)}\sigma) \rangle$*, and additionally* $S_{r(i-1)}\sigma \in \mathsf{Clo}_{\mathcal{R}}(\Gamma')$

Theorem 9 (Completeness of compiled program)**.** *If Γ, Δ, Γ', and Δ' contain no execution or specialization propositions and $\mathcal{T}_1 : \langle \Gamma, \Delta \rangle \ldots \langle \Gamma', \Delta' \rangle$ is a trace under the compiled program $\mathcal{R}$, then there exists another trace $\mathcal{T}_2 : \langle \Gamma, \Delta \rangle \ldots \langle \Gamma', \Delta' \rangle$ under the compiled program $\mathcal{R}$ that simulates a trace $\mathcal{T}_3 : \langle \Gamma, \Delta \rangle \ldots \langle \Gamma', \Delta' \rangle$ in the original program $\mathcal{P}$.*

Proof. By induction on the length of a $\mathcal{T}_1$. If $\mathcal{T}_1$ contains more than 1 state, then the next-to-last state must take the form $\langle \Gamma'', (\Delta'', S_{ri}\sigma) \rangle$, $\langle \Gamma'', (\Delta'', E_{ri}\sigma) \rangle$, or $\langle \Gamma'', (\Delta'', E_{ri}\sigma, S_{ri}\sigma) \rangle$, and repeated appeals to the permutation theorem (Theorem 8) can work backwards to obtain a smaller trace ending with $\langle \Gamma'', (\Delta''', A_1\sigma, \ldots, A_n\sigma) \rangle$ that contains no execution or specialization propositions, allowing the induction hypothesis to be applied to give a smaller trace in the program $\mathcal{P}$ ending with $\langle \Gamma'', (\Delta''', A_1\sigma, \ldots, A_n\sigma) \rangle$; rule r in the program $\mathcal{P}$ can then be applied to extend the trace to end with $\langle \Gamma', \Delta' \rangle$. □

4.4 Indexing and the complied program

As a segue into the description of the algorithm, we introduce the indexing structure that is the reason for the way the compiled program is structured.

The primary purpose of the compiled program is to reduce a program that has rules with many premises to one that has clauses with only two premises. This is because the implementation we will describe relies on being able to take a database of propositions and a proposition not in the database and rapidly determine all the immediate consequences of adding that proposition to the

database. This is difficult to do in an efficient manner if there are arbitrarily many premises, but can be done straightforwardly when there are only two.

This rapid discovery of all immediate consequences is achieved using the data structure INDEX. In the pseudocode below we will have commands like "Find all E_{ri} (alternatively S_{ri}) in INDEX matching proposition S_{ri} (alternatively E_{ri})," and "Insert E_{ri} (alternatively S_{ri}) into INDEX."[1] The data structure can be implemented as a hash table where the keys are ground index terms $\mathsf{f}_{\mathsf{ri}}(\mathbf{t})$ or $\mathsf{g}_{\mathsf{ri}}(\mathbf{t})$ and the values are lists of all asserted propositions that contain that term. If we have a ground term E_{ri}, we can look up all matching S_{ri} by taking $\mathsf{f}_{\mathsf{ri}}(\mathbf{t})$, the index term of E_{ri}, and looking up the corresponding index term $\mathsf{g}_{\mathsf{ri}}(\mathbf{t})$ in INDEX. We add E_{ri} to INDEX by first looking up its index term $\mathsf{f}_{\mathsf{ri}}(\mathbf{t})$ in INDEX, and then adding E_{ri} to the list that is returned. We use hash-consing to represent variable-free terms in order to allow structured terms to be used as hash table keys as needed; therefore, both lookup from and insertion into INDEX can be implemented as constant-time operations. We also use the hash-consing representation to allow for constant time equality check on ground terms.

4.5 Saturating forward reasoning

We will first focus on the portion of the interpreter that implements the Clo function over the fragment of the logic that has no linear propositions. This is exactly the language that McAllester deals with in [11], but our algorithm is a bit more detailed and is tuned towards the integration of linear forward reasoning as discussed in Section 4.6.

In addition to INDEX, we have a hash set of persistent ground propositions FACTS; intuitively, if a ground proposition A is in FACTS, that means we know it to be true. The set FACTS only contains ground atomic propositions from the original program; it does not contain specialization or execution propositions, which go exclusively into INDEX.

The saturation loop in Figure 6 is a simple seven lines; saturation is performed by removing a ground (specialization, execution) proposition from QUEUE, adding it to INDEX, finding all matching (execution, specialization) propositions in INDEX, and asserting the consequences of every pair of propositions. The `conclude` function is described in detail in Section 6, but its function is simple – `conclude`(E_{ri}, S_{ri}) returns the unique consequence of a matching specialization and execution predicate in the compiled program $\mathcal{R}$.

The `assert` function in Figure 7 is only a bit more complicated. The function behaves in two different ways depending on whether it is called with a ground execution or specialization proposition, or with some number of ground conclusions from the original program.

4.5.1 The persistent queue

The persistent queue QUEUE is the basic data structure that drives the exhaustive forward reasoning. The following points describe what behavior we require, and do not require, out of QUEUE, as well as its invariants with respect to the rest of the algorithm.

[1] Throughout, we will use green highlighting as an extra indicator to point out when terms or propositions are ground (that is, variable-free); however, if you are reading a black and white copy, we do not depend on this notation.

`function saturate()`
1. While QUEUE is nonempty, dequeue an element A from QUEUE
2. If $A = E_{ri}$, then:
3. Add E_{ri} to INDEX, find all matching S_{ri} in INDEX
4. For each such S_{ri}, `assert`(`conclude`(E_{ri}, S_{ri}))
5. If $A = S_{ri}$, then:
6. Add S_{ri} to INDEX, find all matching E_{ri} in INDEX
7. For each such E_{ri}, `assert`(`conclude`(E_{ri}, S_{ri}))

Figure 6: The function `saturate` for computing the closure of the set FACTS.

`function assert`(A)
1. If A is a persistent execution proposition E_{ri} or specialization proposition S_{ri}
2. Add P to QUEUE
3. If A is a conclusion $C_0, \ldots, C_{m-1}$ from the original program
4. For each ground atomic proposition C_i
5. If C_i is persistent and not in FACTS, insert C_i into FACTS
6. For each rule $A \multimap S_{ri}$ where $A\sigma = C_{ri}$ for some σ, `assert`($S_{ri}\sigma$)

Figure 7: The function `assert` for executing the persistent fragment

- QUEUE contains ground execution propositions E_{ri} and specialization propositions S_{ri}.
- The order in which atomic propositions are removed from QUEUE is completely unspecified. The name is suggestive of FIFO behavior because, if Clo might not terminate, the algorithm will eventually find any true fact if QUEUE's behavior is FIFO, but not if its behavior is LIFO. We are only concerned with terminating programs in this work, so we never need to rely on this property.

4.5.2 Correctness

Assuming that all the ground execution propositions E_{r1} that were introduced by the compilation process have already been introduced into INDEX before `saturate`() is called, the `saturate` function implements the Clo operation, and its main invariants can be described independently of the linear fragment of the language. There are two crucial invariants. The first invariant primarily guarantees soundness – the `assert` function is only ever called on ground atomic propositions that are immediate consequences of the set of persistent propositions in either FACTS or INDEX. The second invariant ensures completeness – if a persistent atomic proposition S_{ri} or E_{ri} is an immediate consequence (the result of a single focusing step in the compiled program) of the persistent propositions in INDEX and FACTS, and is not itself in INDEX or FACTS, it is in QUEUE. Because QUEUE is empty when `saturate`() terminates, termination means that FACTS and INDEX represent a set of persistent propositions closed under the rules of $\mathcal{R}$. Theorem 7 gives us as a corollary that the set of persistent propositions in FACTS is closed under the rules of the original program $\mathcal{P}$.

`function conclude`(E_{ri}, S_{ri})

1. $E_{ri} = \mathsf{exec}_{\mathsf{ri}}(\mathsf{f}_{\mathsf{ri}}(\mathbf{r}), \mathbf{s})$ or $\underline{\mathsf{exec}_{\mathsf{ri}}}(\mathsf{f}_{\mathsf{ri}}(\mathbf{r}), \mathbf{s})$
2. $S_{ri} = \mathsf{spec}_{\mathsf{ri}}(\mathsf{g}_{\mathsf{ri}}(\mathbf{r}), \mathbf{t})$ or $\underline{\mathsf{spec}_{\mathsf{ri}}}(\mathsf{g}_{\mathsf{ri}}(\mathbf{r}), \mathbf{t})$
3. There is a unique clause in $\mathcal{R}$ with one of the following forms:
4. e_{ri}, premises $\mathsf{exec}_{\mathsf{ri}}(\mathsf{f}_{\mathsf{ri}}(\mathbf{x}), \mathbf{y}), \mathsf{spec}_{\mathsf{ri}}(\mathsf{g}_{\mathsf{ri}}(\mathbf{x}), \mathbf{z})$ and conclusion(s) C_{ri}
5. e_{ri}, premises $\underline{\mathsf{exec}_{\mathsf{ri}}}(\mathsf{f}_{\mathsf{ri}}(\mathbf{x}), \mathbf{y}), \mathsf{spec}_{\mathsf{ri}}(\mathsf{g}_{\mathsf{ri}}(\mathbf{x}), \mathbf{z})$ and conclusion(s) C_{ri}
6. e_{ri}, premises $\mathsf{exec}_{\mathsf{ri}}(\mathsf{f}_{\mathsf{ri}}(\mathbf{x}), \mathbf{y}), \underline{\mathsf{spec}_{\mathsf{ri}}}(\mathsf{g}_{\mathsf{ri}}(\mathbf{x}), \mathbf{z})$ and conclusion(s) C_{ri}
7. e_{ri}, premises $\underline{\mathsf{exec}_{\mathsf{ri}}}(\mathsf{f}_{\mathsf{ri}}(\mathbf{x}), \mathbf{y}), \underline{\mathsf{spec}_{\mathsf{ri}}}(\mathsf{g}_{\mathsf{ri}}(\mathbf{x}), \mathbf{z})$ and conclusion(s) C_{ri}
8. Let ground C' be $C_{ri}\sigma$, where σ is $(\mathbf{s}/\mathbf{y}, \mathbf{t}/\mathbf{z})$
9. If C' is an execution proposition, its support is the combined support of E_{ri} and S_{ri}
10. Return C'

Figure 8: The function `conclude`, which computes the unique consequence of a ground execution proposition E_{ri} and S_{ri}.

4.6 Linear forward reasoning

To execute linear programs, we need to add a new data structure LINEAR that is the companion of FACTS. Instead of a hash set, it is a hash table mapping ground propositions to a set of of unique instances of that proposition; distinct instances of the same linear proposition can be thought of, as before, as having distinguishing labels. Linear specialization and execution propositions all keep track of their *support*, the linear resources in LINEAR that the linear proposition relies on – this quantity is bounded by the size of the program. Each resource in LINEAR will similarly need to maintain forward references to all the execution and specialization propositions that rely on it as a support.

4.6.1 Prefix firings and the `conclude` function

Taking supports into account, a pair consisting of a ground execution proposition E_{ri} and specialization proposition S_{ri} that share the same index term corresponds to a potential prefix firing $\langle r, \sigma, [l_0, \ldots, l_i]\rangle$ as described by Definition 1. The substitution σ is defined by the ground terms in E_{ri} and S_{ri}, the support of E_{ri} is $[l_0, \ldots, l_{i-1}]$, and the support of S_{ri} is l_i. Every such pair of a specialization and an execution predicate has some consequence in the derived program, either a new execution proposition $E_{r(i+1)}$ or the conclusion of a rule in the original program. The function `conclude`, described in Figure 8, takes a ground execution proposition and a matching ground specialization proposition and returns the unique ground proposition provable in the compiled program from those two propositions. This function can be implemented in time proportional to the number of variables in the premises and the size of the conclusion C_{ri}; both are bounded by the size of the program, so we can treat them as constant.

In order to show that our interpreter proves Theorem 4, it is important to show that each call to `conclude` can be associated with a unique prefix firing and that function is called at most once per unique prefix firing.

```
function execute(Γ0, Δ0)
1.  For every rule r, assert(exec_r0(f_r0))
2.  For every A ∈ Γ ∪ Δ0, assert(A)
3.  saturate()
4.  While LQUEUE is nonempty, dequeue an element A
5.      If A = E_ri, then search(E_ri)
6.          If the function returns failure, add E_ri to INDEX
7.          If the function returns success, consume(E_ri)
8.      If A = S_ri, find all matching E_ri in INDEX
9.          For each E_ri, do search(conclude(E_ri, S_ri))
10.             If the function returns success,
11.                 consume(E_ri), consume(S_ri), and break out of the loop
12.         If no call to search() returned success, add S_ri to INDEX
13.         If S_ri is unrestricted, add it to Index in either case.
14.     saturate()
```

Figure 9: The top-level function `execute` for linear execution

4.6.2 The linear queue

Figure 9 shows the outermost function, `execute()`. It uses the last data structure that must be introduced, the linear queue LQUEUE; its properties are much like the properties of QUEUE, with one important extra requirement.

- LQUEUE contains ground execution propositions E_{ri} and specialization propositions S_{ri}.
- The order in which atomic propositions are removed from LQUEUE has one important constraint. If two ground propositions related to the same rule are both in LQUEUE, then a proposition S_{ri} must be removed before a proposition E_{ri}, and if $i > j$ then either S_{ri} or E_{ri} must be removed before S_{rj} or E_{rj}.

If we look at the main loop (lines 4-13 of `execute`) as taking a prefix and trying to extend it, then the second property of LQUEUE requires that we try to extend the longer prefixes first. This requirement does not interfere with constant-time removal from LQUEUE because the i in E_{ri} or S_{ri} is bounded by the maximum number of premises for any rule in the program and can be treated as a constant.

The loop in lines 4-13 is the interesting portion of `execute`; each iteration of this loop corresponds to a (possibly unsuccessful) attempt to fully apply a rule with linear premises. Line 4 represents one way an element can be removed from PQUEUE; the other way a proposition can be removed from PQUEUE is if propositions in its support are consumed; this is shown in lines 6 and 10 of `consume` in Figure 12.

`function search(`A`)`
1. If A is an execution proposition E_{ri}
2. Find all matching S_{ri} in INDEX
3. For each such S_{ri} where the support of E_{ri} and S_{ri} do not overlap
4. `search(conclude(`E_{ri}, S_{ri}`))`
5. If the function returns success, `consume(`S_{ri}`)` and return success
6. If no call to `search()` returned success, add E_{ri} to INDEX and return failure
7. Otherwise, A is a conclusion for C_r from the original program
8. `assert(`C_r`)` and return success

Figure 10: The function `search` for linear execution

`function assert(`A`)`
1. If A is an persistent execution proposition E_{ri} or specialization proposition S_{ri}
2. Add A to QUEUE or LQUEUE appropriately *(* See Section 4.6 *)*
3. If A is a conclusion $C_0, \ldots, C_{m-1}$ from the original program
4. For each ground atomic proposition C_i
5. If C_i is persistent and not in FACTS, insert C_i into FACTS
6. For each rule $A \multimap S_{ri}$ where $A\sigma = C_i$ for some σ, `assert(`$S_{ri}\sigma$`)`
7. If C_i is linear, insert C_i into LINEAR
8. For each rule $A \multimap S_{ri}$ where $A\sigma = C_i$ for some σ, `assert(`$S_{ri}\sigma$`)`

Figure 11: Modifications to the function `assert` for linear execution

4.6.3 Linear search

The most interesting new function needed for linear execution is the `search` function shown in Figure 10. The recursion depth of `search` is bounded by the number of premises in a given rule, and thus is bounded by the size of the program; again, we treat this as a constant. The function is called only on conclusions in the compiled program, that is, either on execution propositions E_{ri} or conclusions C_r from the original program. Instead of using a queue to incrementally search out possible conclusions, it eagerly tries to extend prefixes until either there is no way to do so or until the argument to `search` is a conclusion C_r, meaning that a rule r from the original program can be fully applied. The invariant on LQUEUE ensures that when, on line 2, we search for all matching S_{ri} in INDEX, there are no S_{ri} in LQUEUE, because those must necessarily already have been removed from LQUEUE and added to INDEX before `search` was called.

In addition to adding new functions, linear execution requires changes to the `assert` function. This function behaves the same as the `assert` function described in Figure 7 when describing the purely persistent fragment, but behaves differently when given a S_{ri} or E_{rj} that either is a linear proposition itself or else has a matching E_{ri} or S_{rj} that is linear. The vague comment on line 2 of `assert`, where an persistent proposition E_{ri} or S_{ri} must be added "appropriately," just refers to this difference in behavior; if E_{ri} is either linear or has a matching S_{ri} that is linear, it must be added to LQUEUE; otherwise, it must be added to QUEUE, and the same in the case of S_{ri}.

Finally, `consume` handles the revocation of linear resources once they have been used in a linear

```
function consume(A)
1. If A is a persistent execution proposition E_ri
2.       Insert E_ri into LQUEUE
3. If A is a persistent specialization proposition S_ri, do nothing
4. If A is a linear execution proposition E_ri or S_ri
5.       For every B in the support of A
6.             Remove every C in INDEX or LQUEUE that has B in its support
```

Figure 12: The function `consume` for linear execution

transition.

4.7 Correctness

Correctness again follows from a number of invariants. Completeness follows from the familiar property that, at all times, if any atomic proposition is an immediate consequence of the propositions in INDEX, FACTS, and LINEAR, then if that proposition is not in one of those places, it is in either QUEUE or LQUEUE; if LQUEUE becomes empty, that means that no rule with linear premises could be fully applied, and so the state of the system is the end of a complete trace in $\mathcal{P}$.

A number of interacting invariants ensure soundness:

- At all points where a specialization or execution proposition is passed as the argument to the `assert` on line 4 or 7 of `saturate`, it is the immediate consequence of persistent propositions in FACTS under the compiled program $\mathcal{R}$.
- Whenever a proposition is passed as an argument to the `search` function it is a consequence (not always immediate) of propositions in FACTS, INDEX, and LINEAR under the compiled program $\mathcal{R}$
- Whenever a conclusion from the original program is asserted in line 8 of `search`, it is an non-immediate consequence of the propositions in FACTS, INDEX, and LINEAR under the compiled program $\mathcal{R}$ and an immediate consequence of propositions in FACTS and LINEAR under the original program $\mathcal{P}$.
- Whenever a specialization proposition S_{ri} is added to INDEX in line 12 of `execute` and whenever a execution proposition E_{ri} is added to INDEX in line 6 of `search`, *all* of its possible consequences under the compiled program $\mathcal{R}$ have been added to INDEX
- Whenever a proposition is consumed on line 11 of `execute` or line 5 of `search`, the linear resources in the support of those propositions have already been consumed by the application of a rule from the original program $\mathcal{P}$.
- When a proposition is removed from LQUEUE, it is either added to INDEX or passed as the argument to `consume` before a new item is removed from LQUEUE.

4.8 Complexity

The bulk of the proof of Theorem 4 is that distinct points in the program can be associated with unique prefix firings, and that there is only a constant amount of work related to each prefix firing; these distinct points are points where the function `conclude` is called.

When we execute `conclude`(E_{ri}, S_{ri}) on line 4 or 7 of `saturate`, it is associated with the unique prefix firing that corresponds to the pair of E_{ri} and S_{ri}. Each of these points is followed by a constant amount of work calling `assert` and then potentially dequeueing a new proposition. We still must show that `conclude` is only called once for a given pair of S_{ri} and E_{ri}. We can prove that a given specialization predicate will only be added to QUEUE immediately, and that each execution predicate E_{ri} will only be added to QUEUE once by induction on i. Because elements will only be added to QUEUE once, they will only be removed once. Because each of the calls to `consume` involve a proposition freshly removed from QUEUE, this means that each pair can only can only be given as an argument to `conclude` once.

In dealing with linear propositions, one difficulty is that it is not clear how long it will take to execute a call to `consume`. We can deal with this cost by amortized analysis, "charging" the eventual deletion of a linear proposition at the time of its creation in the `conclude` function, because the time required to execute a call to `consume` is proportional to the number of propositions removed from INDEX and QUEUE in the course of execution; from this, we can treat every call to `consume` as taking constant (amortized) time.

When we execute `conclude`(E_{ri}, S_{ri}), on line 4 of `search`, we are in one of three cases:

- The conclusion is a conclusion C_r from the original program.
- The conclusion is a $E_{r(i+1)}$, and there is no corresponding $S_{r(i+1)}$ in INDEX (there cannot be a corresponding $S_{r(i+1)}$ in PQUEUE by the second criteria described in Section 4.6.2.
- The conclusion is a $E_{r(i+1)}$, and there is corresponding $S_{r(i+1)}$ in INDEX.

In the first case, the abstract operational semantics can take a step by applying a rule from the original program. In the second case, the search has found a prefix that cannot be extended, and in the third case, the search has found a prefix that can be extended. The call depth for `search` is bounded by the size of the program, and forms a tree where every leaf either corresponds to a prefix firing that cannot be extended as described in Definition 1 or a way for the program to take a step. If we amortize the cost of `consume`, then each recursive call does a constant amount of work, and by "paying" for the leaves we pay for the tree (because the depth of the tree is bounded by the size of the program, or specifically the maximum number of premises of any rule in the program).

With one exception – unrestricted execution propositions – when items are removed from LQUEUE, they are either consumed or added to INDEX, which results in the property that `search` will never be called twice with the same execution predicate (with the same support) twice *unless* it is an unrestricted execution proposition, which can only be the case for the top-level call to `search` from `execute`, and every one of the (possibly many) times an unrestricted execution proposition is removed from LQUEUE it either participates in a complete prefix firing or is added to INDEX, never to be removed again – in other words, all but the last removal can be "charged" to one of the m linear transitions.

$$\frac{\underline{\mathsf{nat}}(n)}{\underline{\mathsf{nat}}(\mathsf{s}(n))\quad \underline{\mathsf{item}}(\mathsf{s}(n))} \qquad \frac{\underline{\mathsf{item}}(n)\quad \underline{\mathsf{list}}(l)}{\underline{\mathsf{list}}(n :: l)}$$

Figure 13: A possibly nonterminating program.

The abstract running time has four elements, $|\Gamma_0| + |\Delta_0| + m + \Phi$, and line 2 of `execute` runs in constant time (time proportional to the size of the program), whereas line 2 runs in time proportional to $|\Gamma_0|+|\Delta_0|$. Each call to `saturate` takes constant time, plus a time charged to some unique prefix firings Φ. The loop in `execute` may remove a S_{ri} with no matching E_{ri} in INDEX and then add S_{ri} to INDEX; in this case the iteration takes constant time, which is "charged" to the creation of S_{ri}. Otherwise, this loop will either discover some number of the prefix firings Φ and/or discover a way for an original rule to be fully applied, resulting in one of m linear transitions; in either case, all the work done by each command in `execute` is paid for by elements of the abstract running time.

4.9 Coda: Alternate running time theorem

A point that may prove important in practice goes back to the structure of Theorem 4. The structure of that theorem requires proving that *all* possible initial states will terminate in the abstract algorithm. It may be that some initial states do not or may not always terminate. Take, for example, the slightly contrived Figure 13, which is a hybrid of Figure 4 and the nonterminating program example in Section 1.1. As long as we are only concerned with programs where the initial state contains no resources of the form $\underline{\mathsf{nat}}(n)$, then programs will always terminate; however, application of Theorem 4 requires that *all* initial states terminate.

Theorem 10 is a stronger version of Theorem 4 that handles these situations by restricting the analysis only to initial states satisfying some initial condition Φ; in practice, this will likely be a invariant on states, but this is not required.

Theorem 10 (Running time – revised)**.** *For any program $\mathcal{P}$ that terminates whenever the initial state satisfies some condition $\Phi(\langle \Gamma_0, \Delta_0 \rangle)$, there exists an interpreter running on a RAM machine extended with constant time hash table operations such that for any initial state $\langle \Gamma_0, \Delta_0 \rangle$ such that $\Phi(\langle \Gamma_0, \Delta_0 \rangle)$ is true, the interpreter executes a complete trace $\langle \Gamma_0, \Delta_0 \rangle, \ldots, \langle \Gamma_m, \Delta_m \rangle$ and returns $\mathsf{Clo}(\Gamma_m)$ and Δ_m in time proportional to the abstract running time of the trace.*

5 More linear logical algorithms

In this section, we give several larger examples of linear logical algorithms.

$$\frac{\mathsf{transition}(c, s, c', s', m) \quad \underline{\mathsf{state}}(l, c, r, s)}{\underline{\mathsf{move}}(l, c', r, s', m)} \qquad \frac{\underline{\mathsf{move}}(l, c, (c' :: r), s, \mathsf{right})}{\underline{\mathsf{state}}((c :: l), c', r, s)} \qquad \frac{\underline{\mathsf{move}}(l, c, \mathsf{nil}, s, \mathsf{right})}{\underline{\mathsf{state}}((c :: l), 0, \mathsf{nil}, s)}$$

$$\frac{\underline{\mathsf{move}}((c' :: l), c, s, \mathsf{left})}{\underline{\mathsf{state}}(l, c', (c :: r), s)} \qquad \frac{\underline{\mathsf{move}}(\mathsf{nil}, c, r, s, \mathsf{left})}{\underline{\mathsf{state}}(\mathsf{nil}, 0, (c :: r), s)}$$

Figure 14: Turing machine

5.1 (Potentially non-deterministic) Turing machines

Figure 14 describes a general algorithm for encoding Turing machines. The algorithm requires as input a single linear resource $\underline{\mathsf{state}}(\mathsf{nil}, 0, \mathsf{nil}, \mathsf{a})$, where a is a distinguished start state and 0 is the empty tape symbol, as well as a transition relation encoded as persistent propositions $\mathsf{transition}(s, c, s', c', m)$, which should be read as "from state s reading character c, go to state s', write character c', and move as indicated by m. States are encoded as linear propositions $\underline{\mathsf{state}}(l, c, r, s)$, where l, c, and r represent the left, middle, and right sides of the tape (l and r are lists, and nil stands for an infinite blank tape). The transition relation is totally unspecified, so that certain state + tape symbol combinations may have no forward translation, terminating the algorithm in a stuck state, and certain combinations may have multiple forward translations, meaning that the rules define a non-deterministic Turing machine, and execution will follow an arbitrary trace of the non-deterministic Turing machine.

5.2 Copying garbage collection

We can write a linear logical algorithm for copying garbage collection that is extremely imperative in flavor, capturing the low-level details of the algorithm. We represent the contents of a memory location as terms; a cell in memory can contain either $\mathsf{data}(x)$, which represents non-pointer data, $\mathsf{ptr}(k)$, which represents a pointer to cell k, and $\mathsf{forward}(l)$, which represents the special *forwarding pointers* that are introduced as part of the copying garbage collector. Because memory is imperative, it will be represented by ephemeral propositions $\underline{\mathsf{cell}}(i, d)$, which represents the cell i containing data d.

The *scan* and *next* pointers are explicitly represented as ephemeral propositions of the form $\underline{\mathsf{scan}}(i)$ and $\underline{\mathsf{next}}(j)$.

At the beginning of the algorithm, we have $\underline{\mathsf{scan}}(i_0)$ and $\underline{\mathsf{next}}(j_0)$. "From-space" is all the cells $\underline{\mathsf{cell}}(k, d)$ with $k < i_0$, and "to-space" is all the cells $\underline{\mathsf{cell}}(k, d)$ with $i_0 \leq k$. At the beginning the only cells in to-space are root pointers into "to space," and there is one such cell $\underline{\mathsf{cell}}(k, \mathsf{ptr}(k'))$ for every $i_0 \leq k < j_0$.

The invariants maintained across the program's execution are:

- The linear propositions $\underline{\mathsf{scan}}(i)$ and $\underline{\mathsf{next}}(j)$ exist in every state, $i_0 \leq i \leq j$, there is no cell $\underline{\mathsf{cell}}(k, d)$ such that $j \leq k$, and there is no more than one cell $\underline{\mathsf{cell}}(k, d)$ such that $k < l$.

$$\frac{\underline{\mathsf{scan}}(i) \quad \underline{\mathsf{cell}}(i, \mathsf{data}(x))}{\underline{\mathsf{cell}}(i, \mathsf{data}(x)) \quad \underline{\mathsf{scan}}(\mathsf{s}(i))}$$

$$\frac{\underline{\mathsf{scan}}(i) \quad \underline{\mathsf{cell}}(i, \mathsf{ptr}(k)) \quad \underline{\mathsf{cell}}(k, \mathsf{data}(x)) \quad \underline{\mathsf{next}}(j)}{\underline{\mathsf{cell}}(i, \mathsf{ptr}(j)) \quad \underline{\mathsf{cell}}(j, \mathsf{data}(x)) \quad \underline{\mathsf{cell}}(k, \mathsf{fwd}(j)) \quad \underline{\mathsf{scan}}(\mathsf{s}(i) \quad \underline{\mathsf{next}}(\mathsf{s}(j))}$$

$$\frac{\underline{\mathsf{scan}}(i) \quad \underline{\mathsf{cell}}(i, \mathsf{ptr}(k)) \quad \underline{\mathsf{cell}}(k, \mathsf{ptr}(l)) \quad \underline{\mathsf{next}}(j)}{\underline{\mathsf{cell}}(i, \mathsf{ptr}(j)) \quad \underline{\mathsf{cell}}(j, \mathsf{ptr}(l)) \quad \underline{\mathsf{cell}}(k, \mathsf{fwd}(j)) \quad \underline{\mathsf{scan}}(\mathsf{s}(i)) \quad \underline{\mathsf{next}}(\mathsf{s}(j)}$$

$$\frac{\underline{\mathsf{scan}}(i) \quad \underline{\mathsf{cell}}(i, \mathsf{ptr}(k)) \quad \underline{\mathsf{cell}}(k, \mathsf{fwd}(l))}{\underline{\mathsf{cell}}(i, \mathsf{ptr}(l)) \quad \underline{\mathsf{cell}}(k, \mathsf{fwd}(l)) \quad \underline{\mathsf{scan}}(\mathsf{s}(i))}$$

Figure 15: Copying garbage collection

- Every cell $\underline{\mathsf{cell}}(k, d)$ where $k < i_0$ is either data ($d = \mathsf{data}(x)$), a pointer to from-space ($d = \mathsf{ptr}(l)$ where $l < i_0$), or a forwarding pointer into to-space ($d = \mathsf{fwd}(l)$ where $i_0 \leq l < j$).
- Every scanned cell $\underline{\mathsf{cell}}(k, d)$ in to-space where $i_0 \leq k < i$ is either data ($d = \mathsf{data}(x)$) or a pointer to to-space ($d = \mathsf{ptr}(l)$ where $i_0 \leq l < j$).
- Every unscanned cell $\underline{\mathsf{cell}}(k, d)$ in to-space where $i \leq k < j$ is either data ($d = \mathsf{data}(x)$) or a pointer to from-space ($d = \mathsf{ptr}(l)$ where $l < i_0$).
- Every cell in from-space pointed at by a scanned cell is a forwarding pointer to the to-space cell containing the data formerly contained in that cell.

Correctness of the algorithm follows from these invariants; and the algorithm will terminate when the scan pointer i is equal to the next pointer j, because the invariants maintain that there is no cell $\underline{\mathsf{cell}}(j, d)$ at the location held by the scan pointer, and at this point the cells $\underline{\mathsf{cell}}(k, d)$ where $i_0 \leq k < j$ form a new heap.

5.2.1 Extending to structured data

An interesting aspect of this logical presentation is that it is open-ended, and so we can define an extension to the algorithm just by defining more rules. For instance, Figure 16 shows an extension of the copying garbage collector to vectors, where a vector is a tagged cell $\underline{\mathsf{cell}}(l, \mathsf{vec}(n))$ followed by n other cells. Vectors can be nested if we include the rule marked with a (*), but doing so requires the slight "cheat" that we have to add two natural numbers, and therefore cannot represent the length of a vector as a successor numeral if we want addition to take constant time.

The essence of the extension is a proposition $\underline{\mathsf{write.vec}}(k, n, j)$ that indicates the need to copy n items from k to the free space beginning at location j. $\underline{\mathsf{write.vec}}$ needs to know where the free space starts, so it "captures" the $\underline{\mathsf{next}}(j)$ linear proposition, and the invariant concerning $\underline{\mathsf{next}}$ must be modified to say that either $\underline{\mathsf{next}}(j)$ or $\underline{\mathsf{write.vec}}(k, n, j)$ exists at all times. We do not allow pointers into vectors, so we have an invariant that when $\underline{\mathsf{write.vec}}(k, n, j)$ "scans" the contents of the vector in from-space to copy it into to-space, there will be no forwarding pointers encountered.

$$\frac{\mathsf{scan}(i)\quad \mathsf{cell}(i, \mathsf{vec}(n))}{\mathsf{cell}(i, \mathsf{vec}(n))\quad \mathsf{scan}(\mathsf{s}(i))}$$

$$\frac{\mathsf{scan}(i)\quad \mathsf{cell}(i, \mathsf{ptr}(k))\quad \mathsf{cell}(k, \mathsf{vec}(n))\quad \mathsf{next}(j)}{\mathsf{cell}(i, \mathsf{ptr}(j))\quad \mathsf{cell}(j, \mathsf{vec}(n))\quad \mathsf{cell}(k, \mathsf{fwd}(j)\quad \mathsf{scan}(\mathsf{s}(i))\quad \mathsf{write.vec}(\mathsf{s}(k), n, \mathsf{s}(j))}$$

$$\frac{\mathsf{write.vec}(k, 0, j)}{\mathsf{next}(j)}$$

$$\frac{\mathsf{write.vec}(k, \mathsf{s}(n), j)\quad \mathsf{cell}(k, \mathsf{data}(x))}{\mathsf{cell}(j, \mathsf{data}(x))\quad \mathsf{write.vec}(\mathsf{s}(k), n, \mathsf{s}(j))}$$

$$\frac{\mathsf{write.vec}(k, \mathsf{s}(n), j)\quad \mathsf{cell}(k, \mathsf{ptr}(l))}{\mathsf{cell}(j, \mathsf{ptr}(l))\quad \mathsf{write.vec}(\mathsf{s}(k), n, \mathsf{s}(j))}$$

$$\frac{\mathsf{write.vec}(k, \mathsf{s}(n), j)\quad \mathsf{cell}(k, \mathsf{vec}(m))}{\mathsf{cell}(j, \mathsf{vec}(l))\quad \mathsf{write.vec}(\mathsf{s}(k), n+m, \mathsf{s}(j))}\ (*)$$

Figure 16: Extension of copying garbage collection to vectors

Another interesting aspect of this presentation is that when write.vec "captures" the next proposition until it is done copying, it prevents certain rules from firing (those that have a next as a premise) but not others. Essentially, we see that, interleaved with the copying of a vector, it is possible for the scan pointer to move forward as long as only data, vector declarations, or pointers to already-copied data are encountered. This is one of the intuitions behind distributed garbage collection, and it is clearly suggested by the way the system is written here.

6 Conclusion

We have described a bottom-up logic programming language that has a notion of ephemeral truth as well as the more familiar notion of persistent truth, and we have defined a cost semantics that allows for reasoning about the running time of programs written in this language. The language can be used to express and analyze a number of algorithms that have a notion of stateful change or non-deterministic update, Turing machines, and garbage collection. Our system is unique among similar work in having a proof-theoretic semantics based on focusing and linear logic. In the future, we are interested in pursuing a number of extensions to the language described here, including priorities similar to those in [6], temporal stratification and stratified negation similar to [14], and a notion of equality to describe algorithms that use union-find.

A anonymous reviewer of the companion ICALP paper mentioned that the complexity behavior of additives might be worth considering; in fact, while additives as premises would easily break any reasonable complexity results, additives could be added between atomic linear conclusions without much trouble and without disturbing the cost semantics. For instance, the rule $f : \mathsf{a}, \mathsf{b} \;/\; \mathsf{c}, (\mathsf{d} \;\&\; \mathsf{e})$ has a conclusion that either provides the linear resource d *or* the linear resource e in its conclusion, *but not both*. The check for overlap on line 3 of `search` would have to be modified, as would the

function `consume` (the consumption of d would have to remove e, and vice versa), but this could reasonably be added to the programming language without doing violence to the implementation or the cost semantics.

6.1 Acknowledgments.

We would like to thank Michael Ashley-Rollman, Dan Licata, and the anonymous reviewers for their comments on drafts of this paper. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship by the first author.

We wish to dedicate this paper to Harald Ganzinger, with whom the second author discussed some of the core ideas presented here, and whose untimely passing prevented him from participating further in this research.

References

[1] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The Deductive Database System LDL++. *Theory Pract. Log. Program.*, 3(1):61–94, 2003.

[2] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131, Carnegie Mellon University, April 2003.

[3] Kaustuv Chaudhuri. *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University, December 2006.

[4] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. *Automated Reasoning*, volume 4130, chapter A Logical Characterization of Forward and Backward Chaining in the Inverse Method, pages 97–111. Springer Berlin / Heidelberg, 2006.

[5] Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 514–528, London, UK, 2001. Springer-Verlag.

[6] Harald Ganzinger and David A. McAllester. Logical algorithms. In *ICLP '02: Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, London, UK, 2002. Springer-Verlag.

[7] Sergio Greco and Carlo Zaniolo. Greedy algorithms in Datalog. *Theory Pract. Log. Program.*, 1(4):381–407, 2001.

[8] Georg Lausen, Bertram Ludäscher, and Wolfgang May. On active deductive databases: The statelog approach. In *ILPS '97: International Seminar on Logic Databases and the Meaning of Change, Transactions and Change in Logic Databases*, pages 69–106, London, UK, 1998. Springer-Verlag.

[9] Yanhong A. Liu and Scott D. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 172–183, New York, NY, USA, 2003. ACM.

[10] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 35–46, New York, NY, USA, 2005. ACM.

[11] David A. McAllester. On the complexity analysis of static analyses. *J. ACM*, 49(4):512–537, 2002.

[12] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Automatic complexity analysis. In *ESOP '02: Proceedings of the 11th European Symposium on Programming Languages and Systems*, pages 243–261, London, UK, 2002. Springer-Verlag.

[13] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. A Succinct Solver for ALFP. *Nordic J. of Computing*, 9(4):335–372, 2002.

[14] Christos Nomikos, Panos Rondogiannis, and Manolis Gergatsoulis. Temporal stratification tests for linear and branching-time deductive databases. *Theor. Comput. Sci.*, 342(2-3):382–415, 2005.

[15] Frank Pfenning. Linear logical algorithms. In *Workshop on Programming Logics in memory of Harald Ganzinger*, Saarbrücken, June 2005. Invited talk.

[16] Robert J. Simmons and Frank Pfenning. Linear Logical Algorithms. In *ICALP '08: Proceedings of the 35th International Colloquium on Automata, Languages and Programming,*, London, UK, 2008. Springer-Verlag.

[17] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework II: Judgments and properties. Technical Report CMU-CS-02-102, School of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003.