Linear Logic and Coordination for Parallel Programming

Flávio Manuel Fernandes Cruz

CMU-CS-15-148
March 2016

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh PA, USA

Consortium of:
Universidades do Minho, Aveiro e Porto
Portugal

Thesis Committee
Frank Pfenning, Carnegie Mellon University (Co-Chair)
Seth Goldstein, Carnegie Mellon University (Co-Chair)
Ricardo Rocha, University of Porto (Co-Chair)
Umut Acar, Carnegie Mellon University
Luis Barbosa, University of Minho
Carlos Guestrin, University of Washington

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Support for this research was provided by the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme; by the Portuguese funding agency - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program, under grants SFRH / BD / 51566 / 2011 and within projects POCI-01-0145-FEDER-006961 and FCOMP-01-0124-FEDER-037281.
The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, government or any other entity.

Keywords: Programming Languages, Parallel Programming, Coordination, Linear Logic, Logic Programming, Implementation

Abstract

Parallel programming is known to be difficult to apply, exploit and reason about. Programs written using low level parallel constructs tend to be problematic to understand and debug. Declarative programming is a step towards the right direction because it moves the details of parallelization from the programmer to the runtime system. However, this paradigm leaves the programmer with little opportunities to coordinate the execution of the program, resulting in suboptimal declarative programs. We propose a new declarative programming language, called Linear Meld (LM), that provides a solution to this problem by supporting data-driven dynamic coordination mechanisms that are semantically equivalent to regular computation.

LM is a logic programming language designed for programs that operate on graphs and supports coordination and structured manipulation of mutable state. Coordination is achieved through two mechanisms: (i) coordination facts, which allow the programmer to control how computation is scheduled and how data is laid out, and (ii) thread facts, which allow the programmer to reason about the state of the underlying parallel architecture. The use of coordination allows the programmer to combine the inherent implicit parallelism of LM with a form of declarative explicit parallelism provided by coordination, allowing the development of complex coordinated programs that run faster than regular programs. Furthermore, since coordination is indistinguishable from regular computation, it allows the programmer to reason both about the problem at hand and also about parallel execution.

We have written several graph algorithms, search algorithms and machine learning algorithms in LM. For some programs, we have written informal proofs of correctness to show that programs are easily proven correct. We have also engineered a compiler and runtime system that is able to run LM programs on multi core architectures with decent performance and scalability.

Acknowledgments

I would like to thank my advisors, for their help and encouragement during the development of this thesis. I thank Prof. Ricardo Rocha for his help during implementation and debugging, for his suggestions and for his immense attention to detail. I thank Prof. Frank Pfenning for helping me understand proof theory and for instilling a sense of rigor about my work. I thank Prof. Seth Goldstein for his great insights and for giving me the encourage to pursue the ideas I had during the last 4 years. I have certainly become a better researcher because of you all.

Besides my advisors, I would like to express my gratitude to the following persons. A special acknowledgment to Prof. Fernando Silva, for encouraging me to apply to the PhD program and for believing that I could do it. To Michael Ashley-Rollman, for helping me understand his work on ensemble programming. This thesis would not be possible without his initial contribution.

I am thankful to the Fundacao para a Ciencia e Tecnologia (FCT) for the research grant SFRH/BD/51566/2011, to the Center for Research and Advanced Computing Systems (CRACS), and to the Computer Science Department at CMU for their financial support.

To my fellow friends from DCC-FCUP, specially Miguel Areias, Joao Santos, and Joana Corte-Real, for their friendship and excellent work environment, which certainly helped me during the time I spent in Porto.

To my roommate Salil Joshi, for his friendship during my time at CMU. To all my other friends at CMU for their intellectually stimulating conversations.

My deepest gratitude for my parents and my sisters, for their support and for teaching me the value of hard work and persistence. Finally, I would also like to thank my wife Telma, for her affection and encouragement.

Flavio Cruz

Contents

List of Figures

List of Tables

List of Equations

Chapter 1

Introduction

Multi core architectures have become more widespread recently and are forcing the development of new software methodologies that enable developers to take advantage of increasing processing power through parallelism. However, parallel programming is difficult, usually because programs are written in imperative and stateful programming languages that make use of low level synchronization primitives such as locks, mutexes and barriers. This tends to make the task of managing multi threaded execution complicated and error-prone, resulting in race hazards and deadlocks. In the future, *many-core* processors will make this task even more daunting.

Past developments in parallel and distributed programming have given birth to several programming models. At one end of the spectrum are the lower-level programming abstractions such as *message passing* (e.g., MPI [GFB+04]) and *shared memory* (e.g., Pthreads [But97] or OpenMP [CJvdP07]). While such abstractions give a lot of control to the programmer and provide excellent performance, these APIs are hard to use and debug, which makes it difficult to prove that a program is correct, for instance. On the opposite end, we have many declarative programming models [Ble96] that can exploit some form of implicit parallelism. Implicit parallelism allows the runtime system to automatically exploit parallelism by deciding which tasks to run in parallel. However, it is not always obvious which tasks to run in parallel and which tasks to run sequentially. This important issue, known as the *granularity problem*, needs to be handled well because creating too many parallel tasks will make the program run very slowly since there is a significant overhead for creating parallel tasks. It is then fundamental that there is a good mapping between tasks and available processors. Another different, but related problem, is that declarative programming paradigms offer little to no control to the programmer over how parallel execution is scheduled or how data is laid out, making it hard to improve efficiency. Even if the runtime system reasonably solves the granularity problem, there is a lack of specific information about the program that a compiler cannot easily deduce. Furthermore, the program's data layout is also critical for performance since poor data locality will degrade performance even if the task granularity is optimal. If the programmer could provide such information, then execution would improve in terms of run time, memory usage, or scalability.

In the context of the Claytronics project [GCM05], Ashley-Rollman et al. [ARLG+09, ARRS+07] created Meld, a logic programming language suited to program massively distributed systems made of modular robots with a dynamic topology. Meld programs can derive actions that are used by the robots to act on the outside world. The distribution of computation is done by first

partitioning the program state across the robots and then by making the computation local to the robot. Because Meld programs are sets of logical clauses, they are more amenable to proofs than programs written using lower-level programming abstractions.

In this thesis, we present Linear Meld (LM), a new language for parallel programming over graph data structures that extends the original Meld programming language with linear logic and coordination. Linear logic gives the language a structured way to manage state, allowing the programmer to assert and retract logical facts. While the original Meld sees a running program as an ensemble of robots, LM sees the program as a graph of node data structures, where each node performs computation independently of other nodes and is able to communicate with its neighborhood of nodes. Using the graph as the main program abstraction, LM also solves the granularity program by allowing nodes to be grouped into tasks that can be ran in a single thread of control.

LM introduces a new mechanism, called coordination, that is semantically equivalent to regular computation and allows the programmer to reason about parallel execution. Coordination introduces the concept of *coordination facts*, which are logical facts used for scheduling and data partitioning purposes, and *thread facts*, which allow the programmer to reason about the state of the underlying parallel architecture. The use of these new facilities moves the LM language from the paradigm of implicit parallelism to some form of declarative explicit parallelism, but without the pitfalls of imperative parallel programming. In turn, this makes LM a novel declarative language that allows the programmer to optionally control how execution and data is managed by the execution system.

Our main goal with LM is to efficiently execute provably correct declarative graph-based programs on multi core machines. To show this, we wrote many graph-based algorithms, proved program correctness for some programs and developed a compiler and runtime system where we have seen good experimental results. Finally, we have also used coordination in some programs and we were able to see interesting improvements in terms of run time, memory usage and scalability.

1.1 Thesis Statement

In this thesis, we describe a new linear logic programming language, called Linear Meld (LM), designed to write **declarative parallel graph based programs** for multi core architectures. Our goal with LM is to prove the following thesis:

> Linear logic and coordination provide a suitable basis for a programming language designed to express parallel graph-based programs which, without a significant loss of efficiency, are scalable and easy to reason about.

LM is a novel programming language that makes **coordination** a first-class programming construct that is **semantically equivalent to computation**. Coordination allows the programmer to write declarative code that reasons about the underlying parallel architecture in order to improve the run time and scalability of programs. Since LM is based on logical foundations, programs are amenable to reasoning, even in the presence of coordination.

We support our thesis through seven major contributions:

Linear Logic We integrated linear logic into our language, so that program state can be encoded naturally. The original Meld was fully based on classical logic where everything that is derived is true forever. Linear logic turns some facts into resources that can be consumed when a rule is applied. To the best of our knowledge, LM is the first linear logic based language implementation that attempts to solve real world problems.

Coordination Facts LM offers execution control to the programmer through the use of coordination facts. These coordination facts change how the runtime system schedules computation and partitions data and is semantically equivalent to standard computation facts. We can increase the priority of certain nodes according to the state of the computation and to the state of the runtime in order to make better scheduling decisions so that programs can be more scalable and run faster.

Thread-Based Facts While LM can be classified as a programming language with implicit parallelism, it introduces thread facts to support some form of declarative explicit parallelism. Thread facts allow the programmer to reason about the state of the underlying parallel architecture, namely the execution thread, by managing facts about each thread. Thread facts can be used along with regular facts belonging to nodes and coordination facts, allowing for the implementation of customized parallel scheduling algorithms. We show how some programs take advantage of this facility to create optimal scheduling algorithms that are not possible in the limited context of implicit parallelism.

Provability We leveraged the logical foundations of LM to show how to prove the correctness of programs. We also show that coordination does not change those correctness proofs but only improves run time, scalability or memory usage.

Efficient Runtime and Compilation We have implemented a runtime system with support for efficient data structures for handling linear facts and a compiler that is designed to transform inference rules into C++ code that make use of those data structures. We have achieved a sequential performance that is less than one order of magnitude slower than hand-written sequential C++ programs and is competitive with some more mature frameworks such as GraphLab [LGK+10] or Ligra [SB13].

Multi core Parallelism The logical facts of the program are partitioned across the graph of the nodes. Since the logical rules only make use of facts from a single node, computation can be performed locally, independently of other nodes of the graph. We view applications as a communicating graph data structure where each processing unit performs work on a different subset of the graph, thus enabling concurrency. This allows LM to solve the granularity problem by allowing computation to be grouped into sub-graphs that can be processed by different processing cores. Even if there is poor work imbalance between cores, it is possible to easily move nodes between cores to achieve better load balancing and scheduling.

Experimental Results We have implemented a compiler and a virtual machine prototype from scratch that executes on multi core machines. We have implemented programs such as belief propagation, belief propagation with residual splash, PageRank, graph coloring, N queens, shortest path, diameter estimation, MapReduce, game of life, quick-sort, neural

network training, among others. Our experiments performed on a machine with 32 cores shows that our implementation provides good scalability with up to 32 threads of execution.

Chapter 2

Parallelism, Declarative Programming, Coordination and Provability

In this chapter, we introduce background concepts related to parallel programming, declarative programming with a focus on logic programming, coordination and provability of parallel programs. Some sections of this chapter may be skipped if the reader is already familiar with the topics at hand.

2.1 Explicit Parallel Programming

A popular paradigm for implementing parallelism is *explicit parallel programming*, where parallelism must be explicitly defined and all the mechanisms for coordinating parallel execution must be specified by the programmer. Explicit parallel programming is made possible through the use of imperative programming extended with either a multi-threaded or message passing model of programming.

Multi-Threaded Model In multi-threaded parallelism, the programmer must coordinate the execution of multiple threads of control by sharing state through a shared memory area. The existence of a shared memory area makes it easy to share data between workers, however, access to data from multiple workers needs to be protected, otherwise it might become inconsistent. Many constructs are available to ensure *mutual exclusion* such as *locks* [SGG08], *semaphores* [Dij02], *mutexes* [SGG08], and *condition variables* [Hoa74].

Message Passing For message passing, communication is not done through a shared memory area, but by allowing, as the name says, messages to be sent between threads of control or processes. Message passing is well suited for programming clusters of computers, where it is not possible to have a shared memory area, however message processing is more costly than shared memory area due to the extra work required to send and serialize messages. The most well known framework for message passing is the Message Passing Interface (MPI) [For94].

Since explicit parallel programming is hard to get right due to the non-deterministic nature of parallel execution, other, less explicit, parallel programming models were devised in order to

alleviate some of the problems inherent to the model. One such problem is the *fork/join* model which has made popular with the Cilk [BJK+95] programming language. In Cilk, the programmer writes C programs that can spawn parallel tasks and then explicitly join (or wait) until the spawned tasks are complete. An advantage of Cilk over more pure explicit parallel programming is that the parallel control is performed by the compiler and runtime system, allowing for automatic control over the threads of control through the use of techniques such as *work stealing*, where spawned tasks can be stolen by other processors. However, the programmer still must explicitly indicate where procedures can be spawned in parallel and where they should join.

Another model that uses the fork/join model is the X10 [CGS+05] programming language where spawned procedures are called activities. However, X10 innovates by introducing the concept of places, which are processes that run on different machines and do not share any memory area. This is also commonly called the *partitioned global address space* (PGAS) model, since the memory area is partitioned among processes in order to increase locality. While X10 solves many of the communication and synchronization issues between threads of control and processes, it is still the job of the programmer to effectively use the parallel programming constructs in order to take advantage of parallelism.

2.2 Implicit Parallel Programming

Another form of parallel programming is implicit parallel programming, a style of programming commonly present in declarative programming languages. The declarative style of programming allows the programmer to concentrate more on what the problem is, rather than telling the computer the steps it needs to do to solve the problem. In turn, this gives freedom to the language implementation to work out the details of parallelism, solving many of the issues that arise when writing explicit parallel programs. The programmer writes code without having to deal with parallel programming constructs and the compiler automatically parallelizes the program in order to take advantage of multiple threads of execution. Unfortunately, this comes at a cost of low programmer control over how computation is performed, which may not be optimal under every circumstance. However, declarative programming remains attractive because formal reasoning is performed at a higher level and does not have to deal with low level parallel programming constructs such as locks or message passing.

2.2.1 Functional Programming

Functional programming languages are considered declarative programming languages and are based on the *lambda calculus*, a model for function application and abstraction.

In pure functional programming languages, the side effect free nature of computation allows multiple expressions to evaluate safely in parallel [LRS+03]. This has been realized in in languages such as Id [Nik93] and SISAL [GDF+01] with relative success. However, this kind of parallelism still remains elusive in the functional programming community since practical functional programs have, in general, a high number of fine-grained parallel tasks, which makes it harder for a compiler to schedule computations efficiently [SJ09]. Alternative approaches such

as *data parallelism* [Ble96, CLJ+07, JLKC08], *semi-explicit parallelism* [MML+10, FRRS10], and *explicit parallelism* [HMPJH05] have shown to be more effective in practice.

Data parallelism attempts to partition data among a set of processing units and then apply the same operation on the data. The simplest form of data parallelism is *flat data parallelism*, where the data is flat (list or array) and is easily partitioned. However, functional applications are composed of many recursive data manipulation operations, which is not a good fit for flat data parallelism. However, in the NESL language [Ble96], a new compiler transformation was proposed that could take a program using *nested data parallelism* and turn it into a program using flat data parallelism, which is easier to parallelize. This approach has been later implemented in more modern languages such as Haskell [CLJ+07]. The main advantage of this approach is that it remains true to the original goal of implicit parallelism.

In semi-explicit parallelism, the programmer uses an API to tell the runtime system which computations should be carried out in parallel, thus avoiding the fine-grain granularity problem. An example of such API are the so-called called *sparked computations* [MML+10], which have been made available in the Haskell programming language and express the possibility of performing speculative computations which are going to be needed in the future. In a sense, sparked computations can be seen as *lazy futures* [BH77]. This type of construct was also realized in the Manticore language [FRRS10], which introduces the `pval` binding form for creating potential parallel computations. Interestingly, Manticore also combines support for parallel arrays in the style of NESL and explicit parallelism in the form of a `spawn` keyword that allows the programmer to create threads that communicate using message passing.

2.2.2 Logic Programming

Another example of declarative programming that is suitable for implicit parallel programming is logic programming. Logic programming is usually based on classical logics, such as the *predicate and propositional calculus*, or in non-classical logics such as *constructive* or *intuitionistic* logic, where the goal is to construct different models of logical truth.

Logical systems define their own meaning of truth and a set of consistent rules to manipulate truth. Proofs about statements in a logical system are then constructed by using the rules appropriately. For instance, the system of *propositional logic* contains the *modus ponens* rule, where truth is manipulated as follows: if P implies Q and P is known to be true, then Q must also be true. For example, if we know that "it is raining" and that "if it is raining then the grass is wet", we can prove that "the grass is wet" by using the *modus ponens* rule.

Logic programming arises when the proof search strategy in a logical system is fixed. In the case of propositional logic, the following programming model can be devised:

- A set R of implications of the form "P implies Q" represents the program;
- A set D of truths of the form "P" represents the database of facts;
- Computation proceeds by proof search where implications in R are used to derive new truths using facts from D;
- Every new truth generated using *modus ponens* is added back to D.

This particular proof search mechanism is called *forward-chaining* (or *bottom-up*) *logic pro-*

gramming, since it starts from the initial facts and then uses inference rules (*modus ponens*) to derive new truth. The proof search may then stop if the search has found a particular proposition to be true or a state of *quiescence* is reached (it is not possible to derive any more truths).

An alternative proof search strategy is *backward-chaining* (or *top-down*) *logic programming*. In this style of programming, we want to know if a particular proposition is true and then work backwards using the inference rules. For instance, consider the implications: (i) "if it is raining then the grass is wet" and (ii) "if the weather forecast for today is rain then it is raining" and the proposition (iii) "the weather forecast for today is rain". If we want to know if (iv) "the grass is wet", we select the implication (i) and attempt to prove "it is raining" since it is required by (i) to prove (iv). Next, the goal proposition becomes "it is raining" and the conclusion of implication (ii) matches and thus we have to prove "the weather forecast for today is rain", which can be proved immediately using proposition (iii).

Prolog [CR93] was one of the first logic programming languages to become available, yet it still one of the most popular logic programming languages in use today. Prolog is based on *First Order Logic*, a logical system that extends propositional logic with predicates and variables. Prolog is a backward-chaining logic programming language where a program is composed of a set of rules of the form "P implies Q" that can be activated by inputing a query. Given a query $q(\hat{x})$, a Prolog interpreter will work backwards by matching $q(\hat{x})$ against the head Q of a rule. If found, the interpreter will then try to match the body P of the rule, recursively, until it finds the program base facts. If the search procedure succeeds, the interpreter finds a valid substitution for the $\hat{x}$ variables in the given query.

Datalog [RU93, Ull90] is a forward-chaining logic programming language originally designed for deductive databases. Datalog has been traditionally used in deductive databases, but is now being increasingly used in other fields such as sensor nets [CPT+07], cloud computing [ACC+10], or social networks [SPSL13]. In Datalog, the program is composed of a database of facts and a set of rules. Datalog programs first populate the database with the starting facts and then saturate the database using rule inference. In Datalog, logical facts are persistent and once a fact is derived, it cannot be deleted. However, there has been a growing interest in integrating linear logic [Gir87] into bottom-up logic programming, allowing for both fact assertion and retraction [CCP03, LPPW05a, SP08, CRGP14], which is one of the topics of this thesis.

In logic programming languages such as Prolog, researchers took advantage of the nondeterminism of proof-search to evaluate subgoals in parallel. The most famous models are *OR-parallelism* and *AND-parallelism* [GPA+01]. When performing proof search with two implications of the form "P implies Q" and "R implies Q" then we have OR-parallelism because the proof search can select "P implies Q" and try to prove P but also select "R implies Q" and try to prove R. In AND-parallelism, there is an implication of the form "P and R implies Q" and, to prove Q, it is possible to prove P and Q at the same time. AND-parallelism becomes more complicated when P and Q actually depend on each other, for example, if $P = \texttt{prop}_1(\hat{x})$ and $R = \texttt{prop}_2(\hat{x}, \hat{y})$ then the set variables $\hat{x}$ must be the same in the two propositions. This issue does not arise in OR-parallelism, however AND-parallelism may be better when the program is more deterministic (i.e., it contains less alternative implications) since rule's bodies have more subgoals than the heads.

In Datalog programs, parallelism arises naturally because new logical facts may activate multiple inference rules and thus generate more facts [GST90, SL91, WS88]. A trivial parallelization

can be done by splitting the rules among processing units, however this may require sharing of logical facts depending on the rule partitioning [WS88]. Another alternative is to partition the logical facts among the machines [ZWC91, LCG+06], where rules are restricted in order to facilitate fact partitioning and communication. The LM language presented in this thesis follows this latter approach.

Origins of LM

LM is a direct descendant of Meld by Ashley-Rollman et al. [ARLG+09, ARRS+07], a logic programming language developed in the context of the Claytronics project [GCM05]. Meld is a language suited for programming massively dynamic distributed systems made of modular robots. While mutable state is not supported by Meld, Meld performs *state management* on the persistent facts by keeping a consistent database of facts whenever there is a change in the starting facts. If an axiom is no longer true, everything derived from that axiom is retracted. Likewise, when a fact becomes true, the database is immediately updated to take the new logical fact into account. To take advantage of these state management facilities, Meld supports *sensing* and *action* facts. Sensing facts are derived from the state of the world (e.g., temperature, new neighbor node) and action facts have an effect on the world (e.g., movement, light emission).

Meld was inspired by the P2 system [LCG+06], which includes a logic programming language called NDlog [Loo06] for writing network algorithms declaratively. Many ideas about state management were already present in NDlog. NDlog is essentially a Datalog based language with a few extensions for declarative networking.

2.2.3 Data-Centric Languages

Recently, there has been increasing interest in declarative data-centric languages. MapReduce [DG08], for instance, is a popular data-centric programming model that is optimized for large clusters. The scheduling and data sharing model is simple: in the *map phase*, data is transformed at each node and the result reduced to a final result in the *reduce phase*. In order to facilitate the writing of programs over large datasets, SQL-like languages such as PigLatin [ORS+08] have been developed. PigLatin builds on top of MapReduce and allows the programmer to write complex data-flow graphs, raising the abstraction and ease of programmability of MapReduce programs. An alternative to PigLatin/MapReduce is Dryad [IBY+07] that allows programmers to design arbitrary computation patterns using DAG abstractions. It combines computational vertices with communication channels (edges) that are automatically scheduled to run on multiple computers/cores.

LM, the language presented in this thesis, also follows the data-centric approach since it is designed for writing programs over graph data structures. In Section 3.6, we discuss data-centric languages in more detail and present some other programming systems for data-centric parallel computation and how they compare to LM.

2.2.4 Granularity Problem

In order for implicit parallelism to be effective in practice, its implementation must know which computations should be done in parallel and which computations are better performed sequentially due to the overhead of task synchronization. This is commonly known as the *granularity problem* and it is a crucial issue for good parallel performance. If the implementation uses coarse grained tasks, then this may lead to poor load balancing and thus poor scalability since the tasks take too long. On the other hand, if fine grained tasks are used, then scalability will suffer due to the overhead of synchronization.

In functional programming, every expression can potentially be computed in parallel. If such parallelism is explored eagerly, it will rapidly lead to an enormous overhead due to the amount of fine grained parallelism. The same problem arises in logic programming [TZ93] where multiple subgoals can be proven in parallel at each point in the program, generating large proof trees. It is then fundamental that declarative programming languages target the right level of granularity in order to achieve good scalability and performance.

Several solutions have been proposed and implemented to tackle this granularity problem. As seen in the previous sections, some languages provide explicit keywords that allow the programmer to give hints about which parts of the program can be parallelized. A more general solution has been devised by Acar et al. [ACR11] with the Parallel Algorithm Scheduling Library (PASL), a library that attempts to provide a general solution to task scheduling and granularity control by providing a combination of user-provided asymptotic cost functions and execution time profiling. PASL comes with a novel work stealing algorithm [ACR13] based on private deques that further reduces the overhead of task creation and task sharing. LM also solves the granularity problem but by modeling the application as a graph of nodes where nodes can compute independently of other nodes, allowing for nodes, which represent tasks, to be grouped together into subgraphs to provide the right level of granularity.

2.3 Coordination

Although declarative languages give limited opportunities to coordinate programs, there are other programming languages that directly support different kinds of coordination [PA98]. Most of the following languages cannot be easily classified into either imperative or declarative languages, but tend to be a mix of both.

Coordination programming languages tend to divide execution in two parts: *computation*, where the actual computation is performed, and *coordination*, which deals with communication and cooperation between processing units. This paradigm attempts to clearly distinguish between these two parts by providing abstractions for coordination as a way to provide architecture and system-independent forms of communication.

We can identify two main types of coordination models:

Data-Driven: In a data-driven model, the state of the computation depends on both the data being received or transmitted by the available processes and the current configuration of the coordinated processes. The coordinated process is not only responsible for reading and manipulating the data but is also responsible for coordinating itself and/or other processes.

Each process must intermix the coordination directives provided by the coordination model with the computation code. While these directives have a clear interface, it is the programmer's responsibility to use them correctly.

Task-Driven: In this model, the coordination code is more cleanly separated from the computation code. While in data-driven models, the content of the data exchanged by the processes will affect how the processes coordinate with each other, in a task-driven model, the process behavior depends only on the coordination patterns that are setup before hand. This means that the computation component is defined as a black box and there are clearly defined interfaces for input/output. These interfaces are usually defined as a full-fledged coordination language and not as simple directives present in the data-driven models.

Linda [ACG86] is probably the most well-known coordination model. Linda implements a data-driven coordination model and features a *tuple space* that can be manipulated using the following coordination directives: `out(t)` writes a tuple `t` into the tuple space; `in(t)` reads a tuple using the template `t`; `rd(t)` retrieves a copy of the tuple `t` from the tuple space; and `eval(p)` puts a process `p` in the tuple space and executes it in parallel. Linda processes do not need to know the identity of other processes because processes only communicate through the tuple space. Linda can be implemented on top of many popular languages by simply creating a communication and storage mechanism for the tuple space and then adding the directives as a language library.

Another early coordination language is Delirium [LS90]. Unlike Linda, which is embedded into another language, Delirium actually embeds operators written in other languages inside the Delirium language. The advantages of Delirium are improved abstraction and easier debugging because sequential operators are isolated from the coordination language.

The original Meld [ARLG+09] can also be seen as a kind of data-driven coordination language. The important distinction is that in Meld there is no explicit coordination directives. When Meld rules are activated they may derive facts that need to be sent to a neighboring robot. In turn, this will activate computation on the neighbor. Robot communication is implemented by *localizing* the program rules and then by creating *communication rules*. The LM language also implements communication rules, however it goes further because some facts, e.g., coordination facts, can change how the processing units schedule computation, which may in turn change the program's final results or execution time. This results in a more complete inter-play between coordination code and data.

There are also several important programming models, which are not programming languages on their own right, but provide ways for a programmer to specify how computation should be coordinated. On example is the Galois [PNK+11] system, which is a graph-based programming model that applies operators over the nodes of a graph. Operator activities may be scheduled through *runtime coordination*, where activities are ordered in a specific fashion. In the context of Galois, Nguyen et al. [NP11] expanded the concept of runtime coordination with the introduction of an approach to specify scheduling algorithms. The scheduling language specifies three base schedulers that can be composed and synthesized without requiring users to write complex concurrent code. Another language that builds on top of Galois is Elixir [PMP12], which allows the user to specify how operator application should be scheduled from a high level specification, that is then compiled into low level code. In Section 7.9, we delve deeply into these and other

systems.

2.4 Provability

Many techniques and formal systems have been devised to help reason about parallel programs. One such example is the Owicki-Gries [OG76] deductive system for proving properties about imperative parallel programs (deadlock detection, termination, etc). It extends Hoare logic [Hoa69] with a stronger set axioms such as parallel execution, critical section and auxiliary variables. The formal system can be successfully used in small imperative programs, although using it on languages such as C is difficult since these do not restrict the use of shared variables.

Some formal systems do not build on top of a known programming paradigm, but instead create an entirely new formal system for describing concurrent systems. Process calculus such as π-calculus [Mil99] is a good example of this. The π-calculus describes the interactions between processes through the use of channels for communication. Interestingly, channels can also be transmitted as messages, allowing for changes in the network of processes. The π-calculus is powerful enough to prove that two processes behave the same through the use of bi-simulation equivalence.

Mobile UNITY [RM97] is a proof logic designed for applications with *mobility* of processes. The basic UNITY model [Mis89] assumes that statements could be executed nondeterministically in order to create parallelism. Mobile UNITY extends UNITY by adding locations to processes and removing the nondeterministic aspect from local processes. Processes can then communicate or move between locations.

The Meld language, as a logic programming language, has been used to produce proofs of correctness. A Meld program is amenable to mechanized analysis via theorem checkers such as Twelf [PS99], a logic system designed for analyzing program logics and logic program implementations. For instance, a meta-module based shape planner program was proven to be correct under the assumption that actions derived by the program are always successfully applied in the outside world [DARR+08, ARLG+09]. While the fault tolerance aspect is lax, the planner will always reach the target shape in finite time.

In this thesis, we limit ourselves to informal correctness proofs for LM programs. Unfortunately, there are no available meta reasoning tools for linear logic based programs and proof mechanization is beyond the scope of this work.

2.5 Chapter Summary

In this chapter, we presented the background concepts of this thesis, including parallel programming, declarative programming, coordination, and provability. In the next chapters, we delve more deeply on some of these concepts by including a separate section about recent work that is related to the material covered by each chapter.

Chapter 3

Linear Meld: The Base Language

Linear Meld (LM) is a forward chaining logic programming language where a program is defined by a *database of facts* and by a set of *derivation rules*. LM supports structured manipulation of mutable state through the use of linear logic [Gir87] by supporting two types of logical facts, namely, *persistent facts* (which cannot be retracted) and *linear facts* (which can be retracted).

LM programs are evaluated as follows. Initially, we populate the database with the program's initial facts and then determine which derivation rules can be applied by using the facts from the database. Once a rule is applied, we derive new facts, which are then added to the database. If a rule uses linear facts, they are consumed and thus deleted from the database. The program stops when we reach *quiescence*, that is, when we can no longer obtain new facts through rule derivation.

LM views the database of facts as a graph data structure where each node contains facts belonging to that node. The database of facts can also be seen as a collection of node objects and each node contains several attributes represented as facts that describe the node. Derivation rules are restricted so that they are only allowed to manipulate facts belonging to the same node. This restriction allows nodes of the graph to compute independently of other nodes since they can only perform computation with their current attributes. Communication between nodes arises when a rule derives a fact that belongs to another node. These design decisions allow LM to be concurrent and to also avoid the granularity problem that is usually present in declarative languages. Since computation is performed locally at each node of the graph, it is possible to group nodes into computation units that can processed independently by the same processing thread, which allows the implementation to easily control the granularity of the program.

3.1 A Taste Of LM

In order to understand how LM programs are written, we now present and discuss three LM programs.

3.1.1 First Example: Message Routing

Figure 3.1 shows the first LM program, a message routing program that simulates message transmission through a network of nodes. Lines 1-3 declare the predicates used in the program's rules. Note that the first argument of every predicate must be typed as node because the first argument indicates where the fact lives in the graph. Predicate edge is a *persistent predicate* while message and processed are *linear predicates*. Persistent predicates model facts that are never retracted from the database, while linear predicates model linear facts which are retracted when used in rules. To improve readability of LM rules, persistent predicates are preceded with a ! symbol. Predicate edge represents the connections between nodes, predicate message contains the message content and the route list, and predicate processed keeps count of the number of messages routed at each node. Along with the type, a predicate argument can also be named (e.g., see Neighbor and Content) for documentation purposes.

```
1  type edge(node, node Neighbor).                          // Predicate declaration
2  type linear message(node, string Content, list node Routing).
3  type linear processed(node, int Total).
4
5  message(A, Content, [B | L]),                                        // Rule 1
6  !edge(A, B),
7  processed(A, N)
8     -o processed(A, N + 1),
9        message(B, Content, L).
10
11 message(A, Content, []),                                            // Rule 2
12 processed(A, N)
13    -o processed(A, N + 1).
```

Figure 3.1: *LM code for routing messages in a graph.*

The message routing program in Fig. 3.1 implements two rules. An LM rule has the form $L_1, \cdots, L_n$—o $R_1, \cdots, R_m$, where $L_1, \cdots, L_n$ is the *left-hand side* (LHS) of the rule and $R_1, \cdots, R_m$ is *right-hand side* (RHS) of the rule. The meaning of a rule is then as follows: if facts $L_1, \cdots, L_n$ are present in the database then consume all the facts L_i that are linear facts and derive the facts $R_1, \cdots, R_m$ from the RHS. Note that in the rules, the LHS of each rule uses only facts from the same node (represented by A in both rules of Fig. 3.1), but the rule's RHS may derive facts that belong to other nodes (case of B in the first rule) if the variable is instantiated in the LHS.

The first rule (lines 5-9) grabs the head node B in the route list (third argument of message using the syntax [B | L] to represent the head and tail of a list) and ensures that a communication edge exists (through edge(A, B)). If so, the number of processed messages is increased by consuming processed(A, N) and deriving processed(A, N + 1), along with a new message to the head node B. When the route list is empty, the message has reached its destination and thus it is simply consumed (second rule in lines 11-13).

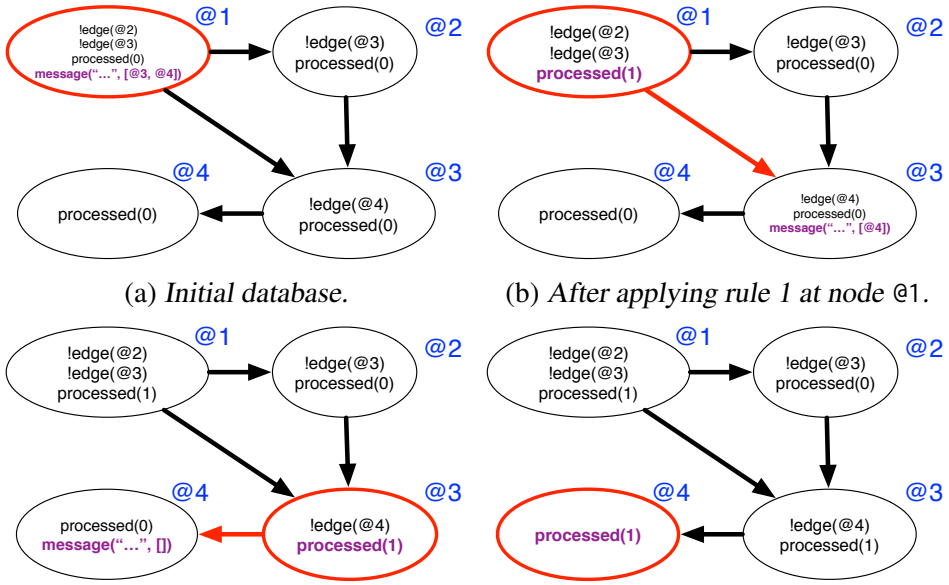
The initial facts of the program are presented in Fig. 3.2. The edge facts describe the structure of the graph, where the node in the first argument has a direct connection with the node in the second argument. Node literals are represented using the syntax @N, where N is an non-negative number. We also declare the message that needs to be routed with the content "hello world"

and a route list with the nodes `@3` and `@4`.

```
1  !edge(@1, @2). !edge(@2, @3).
2  !edge(@3, @4). !edge(@1, @3).
3  processed(@1, 0). processed(@2, 0).
4  processed(@3, 0). processed(@4, 0).
5  message(@1, "hello world", [@3, @4]).
```

Figure 3.2: *Initial facts for the message routing program. There is only one message ("hello world") to route through nodes* `@3` *and* `@4`.

In Fig. 3.3 we present an execution trace of the message routing program. The database is represented as a graph structure where the edges represent the edge initial facts. To simplify the figure, we dropped the first argument of each fact since the first argument corresponds to the node where the fact is placed. In Fig. 3.3(a) the database is initialized with the program's initial facts. Note that the initial `message` fact is instantiated at node `@1`. After applying rule 1, we get the database represented in Fig. 3.3(b), where the message has been derived at node `@3`. After applying rule 1 again, the message is then routed to node `@4` (Fig. 3.3(c)) where it will be consumed (Fig. 3.3(d)).



(a) *Initial database.* (b) *After applying rule 1 at node* `@1`.

(c) *After applying rule 1 at node* `@3`. (d) *After applying rule 2 at node* `@4`.

Figure 3.3: *An execution trace for the message routing program. The message "hello world" travels from node* `@1` *to node* `@4`.

The attentive reader will wonder how much concurrency is available in this particular routing implementation. For a single message, there is no concurrency because the message follows a pre-determined path as it travels from node to node. However, concurrency arises when the program needs to route multiple messages. The amount of concurrency is then dependent on the messages routing lists being non-overlapping. If messages travel in different paths, then there is more concurrency because more nodes are routing messages at the same time.

3.1.2 Second Example: Key/Value Dictionary

```
1  type left(node, node Child).                                    // Predicate declaration
2  type right(node, node Child).
3  type linear value(node, int Key, string Value).
4  type linear replace(node, int Key, string Value).
5
6  replace(A, K, New),                                          // Rule 1: we found our key
7  value(A, K, Old)
8     -o value(A, K, New).
9
10 replace(A, RKey, RValue),                                       // Rule 2: go left
11 value(A, Key, Value),
12 RKey < Key,
13 !left(A, B)
14    -o value(A, Key, Value),
15       replace(B, RKey, RValue).
16
17 replace(A, RKey, RValue),                                      // Rule 3: go right
18 value(A, Key, Value),
19 RKey > Key,
20 !right(A, B)
21    -o value(A, Key, Value),
22       replace(B, RKey, RValue).
```

Figure 3.4: *LM program for replacing a key's value in a BST dictionary.*

Our second example, shown in Fig. 3.4, implements the key update operation for a binary search tree (BST) represented as a key/value dictionary. Each LM node represents a binary tree node. We first declare all the predicates in lines 1-4. Predicates `left` and `right` represent the child nodes of each BST node. Linear predicate `value` stores the key/value pair of a node, while the linear predicate `replace` represents an update operation where the key in the second argument is to be updated to the value in the third argument.

The algorithm uses three rules for the three possible cases of updating a key's value. The first rule (lines 6-8) performs the update by removing `replace(A, K, New)` and `value(A, K, Old)` and deriving `value(A, K, New)`. The second rule (lines 10-15) recursively picks the left branch for the update operation by deleting `replace(A, RKey, RValue)` and re-deriving it at node `B`. Similarly, the third rule (lines 17-22) recursively descends the right branch. The derivation of `replace` facts on node `B` can also be seen as a implicit *message passing* from `A` to `B`, since `B` is a different node than `A` and the LHS of each rule can only manipulate facts from the same node.

The initial facts of the program are presented in Fig. 3.5 and describe the initial binary tree configuration, including keys and values, and the `replace(@1, 6, "x")` fact instantiated at the root node `@1` that manifests the intent to change the value of key 6 to "x".

Figure 3.6 represents the trace of the algorithm. The program database is partitioned by the seven nodes using the first argument of each fact. In Fig. 3.6 (a), we present the database filled with the program's initial facts. Next, we follow the right branch using rule 3 since $6 > 3$ (Fig. 3.6 (b)). We use the same rule again in Fig. 3.6 (c) where we finally reach the key 6. Here, we apply rule 1 and `value(@7, 6, "g")` is updated to `value(@7, 6, "x")`.

```
1  !left(@1, @2).
2  !right(@1, @3).
3  !left(@2, @4).
4  !right(@2, @5).
5  !left(@3, @6).
6  !right(@3, @7).
7
8  value(@1, 3, "a").
9  value(@2, 1, "b").
10 value(@3, 5, "c").
11 value(@4, 0, "d").
12 value(@5, 2, "e").
13 value(@6, 4, "f").
14 value(@7, 6, "g").
15
16 // Update key 6 to value "x".
17 replace(@1, 6, "x").
```

Figure 3.5: *Initial facts for replacing a key's value in a BST dictionary.*

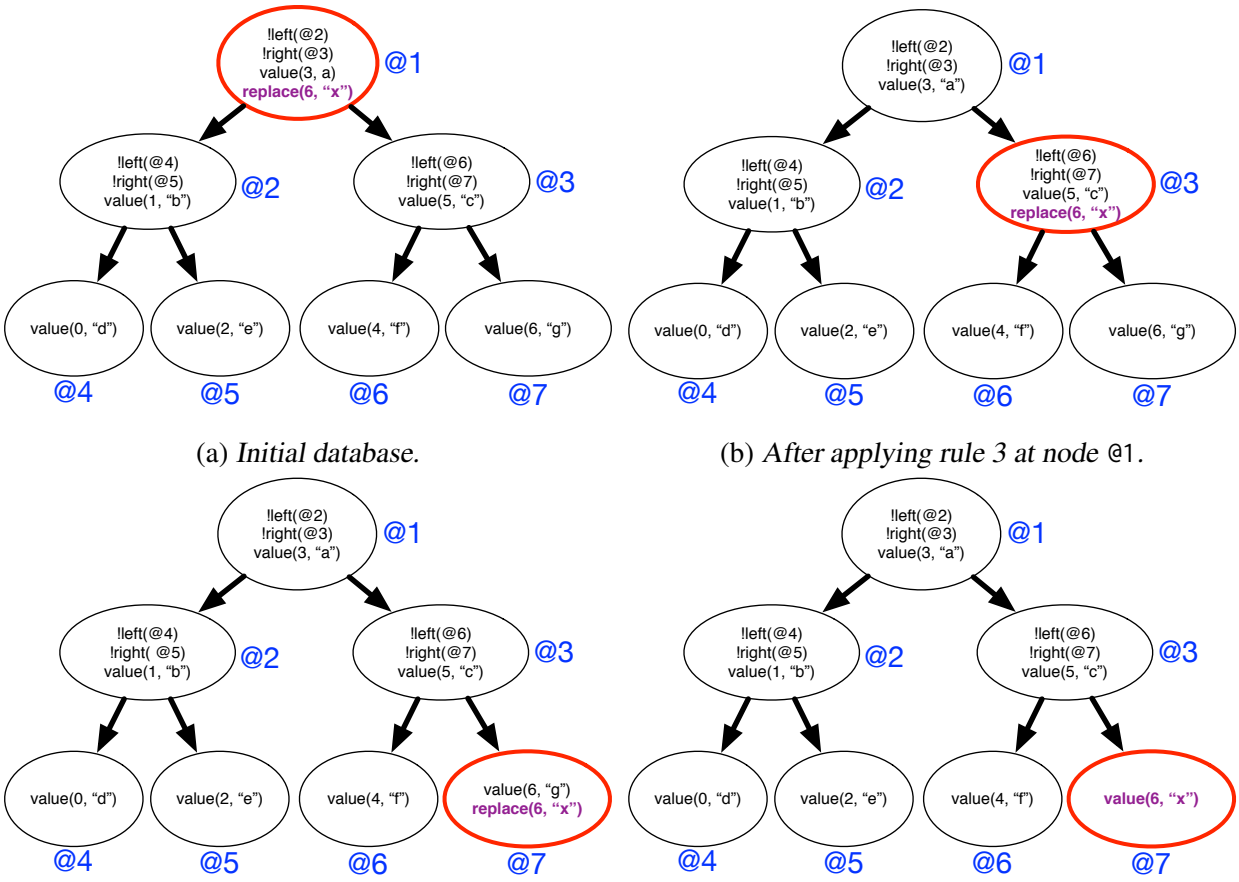
Like the message routing example shown in the previous section, the amount of concurrency present in this example depends on the number of `replace` facts. If there are many `replace` operations to perform on different parts of the BST, then there is more concurrency. On the other hand, if only a few BST nodes are being updated, then concurrency is reduced. This is the same behavior that one would expect from an parallel implementation of the BST data structure in an imperative language.

3.1.3 Third Example: Graph Visit

Our third example, shown in Fig. 3.7, presents another LM program that, for a given graph of nodes, performs a visit to all nodes reachable from node `@1`. The first rule of the program (lines 6-9) visits node `A` for the first time: fact `visited(A)` is derived and a *comprehension* construct is used to go through all the edge facts and then derive `visit(B)` at each neighbor node `B`. The second rule of the program (lines 11-13) applies when the node is already visited more than once: we keep the `visited` fact and delete `visit`. The initial facts shown in Fig. 3.8 use the `visit(@1)` fact to start the program.

Fig. 3.9 shows a possible execution trace for the visit program. After applying the first rule at node `@1` we get the database in Fig 3.9 (b). Note that node `@1` is now visited and nodes `@2` and `@4` now have a `visit` fact. At this point, we could either apply rule 1 at node `@2` or at node `@4`. For this specific trace, we apply the rule at node `@2`, resulting in Fig. 3.9 (c). Node `@4` now has two `visit` facts, thus we can apply rule 1 followed by rule 2, therefore consuming both `visit` facts and deriving `visited`. In addition, we can also apply rule 1 at node `@3` to reach the state of Fig. 3.9 (d).

The graph visit has the potential to have plenty of concurrency. Once the visit starts from the initial node, it expands to other nodes, allowing several nodes to derive rules concurrently. At some point, the amount of concurrency is reduced because more and more nodes have been



(a) *Initial database.*

(b) *After applying rule 3 at node* `@1`.

(c) *After applying rule 3 at node* `@3`.

(d) *After applying rule 1 at node* `@7`.

Figure 3.6: *An execution trace for the binary tree dictionary algorithm. The first argument of each fact was dropped and the address of the node was placed beside it.*

```
type edge(node, node).                                  // Predicate declaration
type linear visit(node).
type linear unvisited(node).
type linear visited(node).

visit(A),                                               // Rule 1: visit node
unvisited(A) -o
   visited(A),
   {B | !edge(A, B) -o visit(B)}.

visit(A),                                               // Rule 2: node already visited
visited(A)
   -o visited(A).
```

Figure 3.7: *LM code for the graph visit program.*

visited. The level of concurrency depends on the structure of the graph. In the worst case, if the graph is a chain of nodes then the program becomes effectively sequential. In the best case, if the

```
!edge(@1, @2).
!edge(@1, @4).
!edge(@2, @3).
!edge(@2, @4).
!edge(@2, @1).
!edge(@3, @2).
!edge(@4, @1).
!edge(@4, @2).

unvisited(@1).
unvisited(@2).
unvisited(@3).
unvisited(@4).

visit(@1).
```

Figure 3.8: *Initial facts for the graph visit program. Nodes reachable from node* `@1` *will be marked as* `visited`.

graph is densely connected (each node connects to most nodes in the graph), then it is possible to run the program on most nodes at the same time.

The goal of introducing higher-level declarative languages is to facilitate reasoning about the properties of a program. In the case of the visit program, the most important goal is to prove that, if the graph is connected, then all the nodes will become `visited`, regardless of the order in which we apply the rules. First, we define a visit graph:

Definition 3.1.1 (Visit graph)
A visit graph is an ordered pair $G = (N, E)$ comprising a set N of nodes together with a set E of edges. Set E contains pairs (A, B) that correspond to facts $!\texttt{edge}(A, B)$. For every pair $(A, B) \in E$ there is also a pair $(B, A) \in E$, representing an undirected edge.

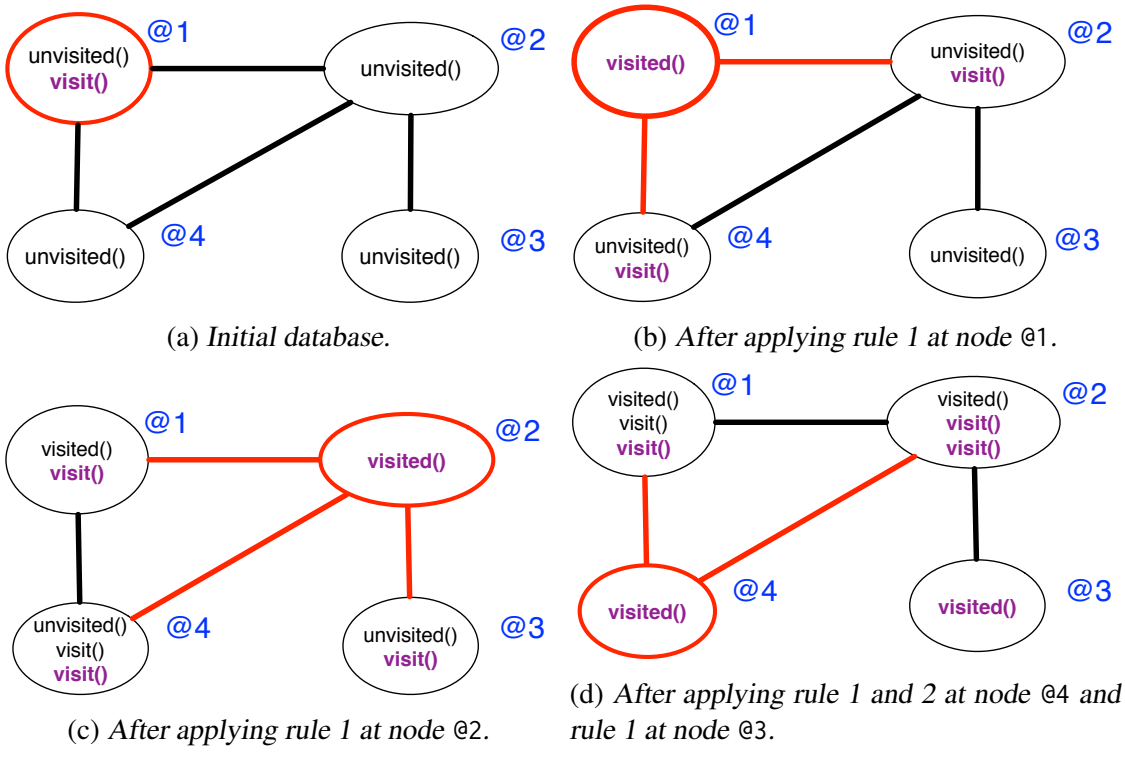
To prove the correctness property of the program, we first define the main *invariant* of the program:

Invariant 3.1.1 (Node state)
A node is either `visited` or `unvisited` but not both.

Proof. From the initial facts we check if all nodes start as `unvisited`. This is clearly true for the previous example. Rule 1 changes a node from `unvisited` to `visited`, while rule 2 keeps the node `visited`, proving the invariant.

□

Invariants are conditions that never change, no matter which rules are derived during execution. With this invariant, it is now possible to classify nodes of the graph G according to their state:



(a) *Initial database.*

(b) *After applying rule 1 at node* `@1`.

(c) *After applying rule 1 at node* `@2`.

(d) *After applying rule 1 and 2 at node* `@4` *and rule 1 at node* `@3`.

Figure 3.9: *A possible execution trace for the visit program. Note that the* `edge` *facts were omitted for simplicity.*

Definition 3.1.2 (Node sets)
`visited` nodes are contained in set V, while `unvisited` nodes are in set U. From the node state invariant, we know that $V \cup U = N$ and $V \cap U = \emptyset$.

We can now prove an important lemma about sets V and U:

Invariant 3.1.2 (Visited set)
After each rule derivation, visited set V always increases or stays the same size. The inverse is true for set U.

Proof. Initially, $V = \emptyset$ and $U = N$. By rule 1, V increases by 1 while U decreases by 1. With rule 2, set membership remains unchanged. □

In turn, since set membership changes from U to V, we now prove the following:

Lemma 3.1.1 (Edge visits)
The program generates at most one `visit` per directed edge and for a node $a \in N$ that receives a `visit` fact, then for all $b \in N$ where $(a, b) \in E$, exactly one `visit` fact is generated at b.

Proof. From the visited set invariant, we know that once nodes become members of set V, they no longer return to set U, therefore rule 1 applies once per node. This rule generates a `visit` fact per neighbor node. □

In order to prove that all the nodes in the graph are visited, we need to make sure that the graph is connected.

Definition 3.1.3 (Connected graph)
A connected graph is a graph where every pair of nodes has a path between them.

Finally, we prove that all nodes will become `visited`.

Theorem 3.1.1 (Graph visit correctness)
If graph G is connected, set V will eventually include all nodes in N, while $U = \emptyset$.

Proof. Proof by induction.

- Base case: initial fact `visit(@1)` adds node `@1` to V. By Lemma 3.1.1, a `visit` fact is generate for all edges of `@1`.
- Inductive case: assume visited set V' and unvisited set U'. Since the graph is connected, there must be a node $a \in V'$ that is connected to a node $b \in U'$. Using the Edge visits lemma, a `visit`(b) fact is generated, swapping b from U' to V'.

Eventually, set V will include all nodes in N. Otherwise, there would be unreachable nodes in the graph and that would be a contradiction since the graph is connected. □

3.2 Types and Locality

Each fact is an association between a *predicate* and a tuple of values. A predicate is a pair with a name and a tuple of types (the argument types). LM rules are type-checked using the predicate declarations in the header of the program. LM has a simple type system that includes the following basic types: *node*, *int*, *float*, *string*, *bool*. The following structured types are also supported: *list* X, for lists of type X; *struct* $X_1, \ldots, X_n$, for composite values made of n elements; and *array* X, for arrays of type X.

LM allows the definition of new type names from simpler types using the declaration `type simple-type new-type` in the header of the program. The type `new-type` can then be used as any other type. Note that LM uses *structural equivalence* to check if two types are the same, therefore `simple-type` and `new-type` are type equivalent.

Type checking LM programs is straightforward due to its simple type system and mandatory predicate declarations. For each rule, the variables found in the LHS are mapped to types based on their use on atomic proposition arguments. Some constraints of the form `X = expression` that force an equality between `X` and `expression` may actually represent an *assignment* if `X` is not defined by any LHS atomic proposition. In this case, all the variables in `expression` must be typed and `X` is assigned the value of `expression` during run time. Any variable used in the RHS of the rule must be defined in the LHS, because otherwise derived facts would not be *grounded*, that is, some arguments would be undefined or uncomputable. For comprehensions (and aggregates), type checking is identical, however, the LHS of each construct must declare explicitly the variables in scope.

Another important component of type checking is *locality checking*. The first argument of each atomic proposition in the LHS must use the same variable in order to enforce locality and allow concurrency. This *home variable* is always typed as a *node* and represents a node in the graph. In the rule's RHS, other home variables are allowed, as long as they have been defined in the LHS. For comprehensions, the LHS must use the same home argument as the rule's LHS.

3.3 Operational Semantics

Nodes are selected for computation non-deterministically (i.e., any node can be picked to run). This means that the programmer cannot expect that facts coming from different nodes will be considered as a whole since the process is non-deterministic. The operational semantics promises that rule derivations are performed atomically, therefore, if a rule derives many facts belonging to a node then that node will receive them all at once. Under these restrictions, computation can then be parallelized by processing nodes concurrently.

Each rule in LM has a defined priority that is inferred from its position in the source file. Rules at the beginning of the file have higher priority. At the node level, we consider all the new facts that have not been considered yet to create a queue of *candidate rules*. The queue of candidate rules is then applied (in priority order) and updated as new facts are derived or consumed. As an example, consider the following three rules:

```
f(A), g(A) -o f(A).

h(A) -o g(A).

g(A) -o 1.
```

If the database contains the facts `h(@1)` and `f(@1)`, then the second rule is applied, retracting `h(@1)` and deriving `g(@1)`. Next, the first and third rules are candidate rules, but since the first rule has higher priority, it is applied first, resulting in a database with a single fact `f(@1)`, where no rule can be applied. Later, in Section 5.3, we give more details about how our implementation manages the set of candidate rules.

3.4 LM Abstract Syntax

Previously, we presented the LM syntax and usage through the presentation of three program examples. We now delve more deeply into LM by presenting the underlying abstract syntax of the language in Table 3.1.

A LM program $Prog$ consists of a list of derivation rules Σ and a database D. A database fact $l(\hat{t})$ is an association between a predicate l and a list of literals $\hat{t}$. Literals t can be either a number $\mathsf{number}(N)$, a node $\mathsf{node}(N)$, a list $\mathsf{list}(\hat{t})$, among other literals such as strings, arrays, booleans, etc.

Each derivation rule R can be written as $LHS \multimap RHS$ with the meaning described in Section 3.1.1. Rules without an LHS are called *initial facts*. All the variables in the rule's scope must be introduced explicitly in the abstract syntax using the $\forall_x.R$ production. However, when the programmer writes LM programs, those variables are introduced implicitly. A rule can also use a *selector* of the form $[\ S \Rightarrow y;\ LHS\ ] \multimap RHS$ which allows the programmer to force a specific ordering during rule derivation. Selectors are described in more detail in Section 3.4.1.

Program	$Prog$	::=	Σ, D
List Of Rules	Σ	::=	$\cdot \parallel \Sigma, R$
Database	D	::=	$\Gamma; \Delta$
Known Linear Facts	Δ	::=	$\cdot \parallel \Delta, l(\hat{t})$
Known Persistent Facts	Γ	::=	$\cdot \parallel \Gamma, !p(\hat{t})$
Literal List	$\hat{t}$	::=	$\cdot \parallel t, \hat{t}$
Literal	t	::=	$\mathsf{number}(N) \parallel \mathsf{node}(N) \parallel \mathsf{list}(\hat{t}) \parallel \ldots$
Rule	R	::=	$LHS \multimap RHS \parallel \forall_x.R \parallel [\ S \Rightarrow y;\ LHS\ ] \multimap RHS$
LHS Expression	LHS	::=	$L \parallel P \parallel C \parallel LHS, LHS \parallel \exists_x.LHS \parallel \mathbf{1}$
Selector Operation	S	::=	`asc` $\parallel$ `desc` $\parallel$ `random`
RHS Expression	RHS	::=	$L \parallel P \parallel C \parallel RHS, RHS \parallel EE \parallel CE \parallel AE \parallel \mathbf{1}$
Linear Atomic Proposition	L	::=	$l(\hat{x})$
Persistent Atomic Prop.	P	::=	$!p(\hat{x})$
Term List	$\hat{x}$	::=	$\cdot \parallel x, \hat{x} \parallel t, \hat{x}$
Constraint	C	::=	$e\ O\ e$
Expression	e	::=	$x \parallel t \parallel \mathsf{fun}(\hat{e}) \parallel e\ M\ e \parallel e\ O\ e$
Expression List	$\hat{e}$	::=	$\cdot \parallel e, \hat{e}$
Math Operation	M	::=	$+ \parallel \times \parallel / \parallel - \parallel \%$
Boolean Operation	O	::=	$= \parallel <> \parallel > \parallel \geq \parallel < \parallel \leq$
Exists Expression	EE	::=	$\mathsf{exists}_{\widehat{x}}.SRHS$
Comprehension	CE	::=	$\{\ \widehat{x} \mid SLHS \multimap SRHS\ \}$
Aggregate	AE	::=	$[\ A \Rightarrow y;\ \widehat{x} \mid SLHS \multimap SRHS_1 \rightsquigarrow SRHS_2\ ]$
Aggregate Operation	A	::=	`min` $\parallel$ `max` $\parallel$ `sum` $\parallel$ `count` $\parallel$ `collect`
Sub-LHS	$SLHS$	::=	$L \parallel P \parallel SLHS, SLHS \parallel \exists_x.SLHS$
Sub-RHS	$SRHS$	::=	$L \parallel P \parallel SRHS, SRHS \parallel \mathbf{1}$

Table 3.1: *Core abstract syntax of LM.*

The LHS of a rule may contain linear (L) and persistent (P) *atomic propositions* and constraints (C). Atomic propositions are template facts that instantiate variables from facts in the database (example in line 11 of Fig. 3.7). Variables can be used again in the LHS for matching and also in the RHS when instantiating facts. Constraints C are boolean expressions that must be true in order for the rule to be derived. Each constraint starts with a boolean operation $e\ O\ e$, where each expression e may be a literal, a variable, a function call $\mathsf{fun}(\hat{e})$ or a mathematical operation $e\ M\ e$.

The RHS of a rule contains linear (L) and persistent (P) atomic propositions which are uninstantiated facts. The RHS can also have *exists expressions* (EE), *comprehensions* (CE) and *aggregates* (AE). All those expressions may use all the variables instantiated in the rule's LHS and are explained in Section 3.4.2. To introduce variables in the scope of the RHS, it is possible to use the $\exists_x.LHS$ production, which can be used for sub-computations for instantiating the atomic propositions of the RHS. This production is heavily used by the compiler to move variables defined in the rule's LHS to the RHS which are only used in the RHS, however it is still possible for the programmer to define RHS's variables explicitly using an equality constraint of the form $x = e$ (represented by C in the syntax).

In order to understand how LM rules are translated into the abstract syntax, consider again the two rules in the graph visit program shown in Fig. 3.7:

```
visit(A),
unvisited(A)
   -o visited(A),
      {B | !edge(A, B) -o visit(B)}.

visit(A),
visited(A)
   -o visited(A).
```

First, we have to de-sugar the code and introduce the variable `A`, that is not explicitly quantified, as follows:

$$\forall_A.\mathtt{visit}(A),\ \mathtt{unvisited}(A) \multimap \mathtt{visited}(A),\ \{\ B \mid !\mathtt{edge}(A, B) \multimap \mathtt{visit}(B)\ \} \tag{3.1}$$

$$\forall_A.\mathtt{visit}(A),\ \mathtt{visited}(A) \multimap \mathtt{visited}(A) \tag{3.2}$$

For the initial facts, they are translated as rules where the LHS is $\mathbf{1}$:

$$\mathbf{1} \multimap !\texttt{edge}(@1, @2) \tag{3.3}$$
$$\mathbf{1} \multimap !\texttt{edge}(@2, @3) \tag{3.4}$$
$$\mathbf{1} \multimap !\texttt{edge}(@1, @4) \tag{3.5}$$
$$\mathbf{1} \multimap !\texttt{edge}(@2, @4) \tag{3.6}$$
$$\mathbf{1} \multimap \texttt{unvisited}(@1) \tag{3.7}$$
$$\mathbf{1} \multimap \texttt{unvisited}(@2) \tag{3.8}$$
$$\mathbf{1} \multimap \texttt{unvisited}(@3) \tag{3.9}$$
$$\mathbf{1} \multimap \texttt{unvisited}(@4) \tag{3.10}$$
$$\mathbf{1} \multimap \texttt{visit}(@1) \tag{3.11}$$

3.4.1 Selectors

During rule derivation, the facts to be used in the LHS of the rule are picked non-deterministically. While our system uses an implementation dependent order for efficiency reasons, sometimes it is important to sort facts by one of the arguments. The abstract syntax for this construct is $[\ S \Rightarrow y;\ LHS\ ] \multimap RHS$, where S is the selector operation and y is the variable in LHS that represents the value to be sorted according to S. An example using concrete syntax is as follows:

```
[asc => W | !edge(A, B, W), select(A)] -o best-neighbor(A, B, W).
```

In this case, we sort the `edge` facts by `W` in ascending order and then try to match them. Other operations available are `desc` and `random` (to force no pre-defined order at the implementation level).

3.4.2 Exists Expressions

Exists constructs (EE) are based on the linear logic construct of the same name and are used to create fresh node addresses. We can use the new node address to instantiate new facts. As an example, consider extending the key/value dictionary example described in Fig. 3.4 with an insertion operation for a node that has no left branch:

```
insert(A, IKey, IValue),
value(A, Key, Value),
no-left-branch(A),
IKey < Key
   -o value(A, Key, Value),
      exists B. (value(B, IKey, IValue), !left(A, B)).
```

The `exists` construct creates a new node `B` containing the linear fact `value(B, IKey, IValue)` (the newly inserted key/value pair) and the persistent fact `!left(A, B)` that connects `A` to `B` is also added to node `A`.

3.4.3 Comprehensions

When consuming a linear fact we might want to generate several new facts depending on the contents of the database. To solve this particular end, we use comprehensions, which are sub-rules that are applied with all possible combinations of facts from the database. In a comprehension $\{\ \widehat{x} \mid SLHS \multimap SRHS\ \}$, $\widehat{x}$ is a list of variables in the scope of $SLHS$ and $SRHS$, where $SLHS$ is the comprehension's left-hand side and $SRHS$ is the comprehension's right-hand side. $SLHS$ is used to generate all possible combinations for $SRHS$ according to the list of variables $\hat{x}$ and to the facts in the database. We have already seen an example of a comprehension in the graph visit program (Fig. 3.7 line 9):

```
visit(A),
unvisited(A)
   -o visited(A),
      {B | !edge(A, B) -o visit(B)}.
```

In this example, the comprehension matches `!edge(A, B)` using all the combinations available in the database for node `B` and for each combination it derives `visit(B)`.

3.4.4 Aggregates

Another useful feature is the ability to reduce several facts into a single fact. LM features aggregates (AE), a special kind of sub-rule that works somewhat like comprehensions. In the abstract syntax $[\ A \Rightarrow y;\ \widehat{x} \mid SLHS \multimap SRHS_1 \rightsquigarrow SRHS_2\ ]$, A is the aggregate operation, $\widehat{x}$ is the list of variables introduced in $SLHS$, $SRHS_1$ and $SRHS_2$ and y is the variable in $SLHS$ that represents the values to be aggregated using A. Like comprehensions, we use $\widehat{x}$ to try all the combinations of $SLHS$, but, in addition to deriving $SRHS_1$ for each combination, we aggregate the values represented by y into a new y variable that is used to derive $SRHS_2$.

To understand how aggregates work, let's consider the following rule:

```
count-neighbors(A) -o [count => T; B | !edge(A, B) -o 1 -> num-neighbors(A, T)].
```

The rule uses a `count-neighbors` proposition to iterate over all `!edge` facts (the $SLHS$) of node `A`. Since the $SRHS_1$ of the aggregate is `1`, nothing is derived for each `!edge`. Since we use a `count` aggregate, for each `!edge` fact, the variable `T` is incremented by one and the total result is used to derive a single `num-neighbors(A, T)` fact (the $SRHS_2$).

To further understand aggregates, let's consider a rule from the PageRank program to be presented in Section 3.5.2:

```
update(A),
!numInbound(A, T)
   -o [sum => V; B, Val, Iter | neighbor-pagerank(A, B, Val, Iter), V =
         Val/float(T) -o neighbor-pagerank(A, B, Val, Iter) -> sum-ranks(A, V)].
```

The rule aggregates the values `V` by iterating over `neighbor-pagerank(A, B, Val, Iter)` (the $SLHS$) and by re-deriving the fact `neighbor-pagerank(A, B, Val, Iter)` (the $SHRS_1$). Once all values are inspected, the atomic proposition `sum-ranks(A, V)` present in $SRHS_2$ is derived once with `V` representing the sum of all the neighbor values `Val/float(T)`. LM provides several aggregate operations, including the `min` (minimum value), `max` (maximum value), `sum`, `count` (count combinations) and `collect` (collect items into a list).

3.4.5 Directives

LM also supports a small set of extra-logical directives that are not represented by the abstract syntax but may be used by the programmer to change the compilation and runtime behavior of the program. The full list is presented in Appendix D.

3.5 Applications

In this section, we present more LM solutions to well-known problems. We start with straightforward graph-based problems such as bipartitness checking and the PageRank program. Next, we present a version of the Quick-Sort algorithm which, from a first impression, was not expected to fit well under the programming paradigm offered by LM. Informal correctness and termination proofs are also included to further show that it is easy to reason about LM programs.

3.5.1 Bipartiteness Checking

The problem of checking if a graph is bipartite can be seen as a 2-color graph coloring problem. The code for this algorithm is shown in Fig. 3.10. The code declares five predicates, namely: `edge`, to specify the structure of the graph; `uncolored`, to mark nodes as uncolored; `colored`, to mark nodes as colored and the node's color; `fail`, to mark an invalid bipartite graph; and `visit` to perform the coloring of the graph. Initially, all nodes in the graph start as `uncolored`, because they do not have a color yet. The initial fact `visit(@1, 1)` is instantiated at node `@1` (line 19) in order to start the coloring process by assigning it with color 1.

If a node is `uncolored` and needs to be marked with a color `P` then the rule in lines 9-11 is applied. We consume the `uncolored` fact and derive a `colored(A, P)` to effectively color the node with `P`. We also derive `visit(B, next(P))` in neighbor nodes to color them with the other color. The coloring can fail if a node is already colored with a color `P` and needs to be colored with a different color (line 15) or if it has already failed (line 17).

In order to show that the code in Fig. 3.10 works as intended, we first setup some invariants that hold throughout the execution of the program. Assume that the set of nodes in the graph is represented as N.

Invariant 3.5.1 (Node state)
The set of nodes N is partitioned into 4 different states that represent the 4 possible states that a node can be in, namely:

- U (`uncolored` nodes)
- F (`fail` nodes)
- C_1 (`colored`$(A, 1)$ nodes)
- C_2 (`colored`$(A, 2)$ nodes)

Proof. Initially, all nodes start in set U (line 20 of Fig. 3.10. All the 4 rules of the programs either keep the node in the same set or exchange the node with another set. □

```
type edge(node, node).                                          // Predicate declaration
type linear uncolored(node).
type linear colored(node, int).
type linear fail(node).
type linear visit(node, int).

fun next(int X) : int = if X <> 1 then 1 else 2 end.            // Function declaration

visit(A, P), uncolored(A)                                      // Rule 1: coloring a node
   -o {B | !edge(A, B) -o visit(B, next(P))},
      colored(A, P).

visit(A, P), colored(A, P) -o colored(A, P).          // Rule 2: node is already colored

visit(A, P1), colored(A, P2), P1 <> P2 -o fail(A).    // Rule 3: graph is not bipartite

visit(A, P), fail(A) -o fail(A).                 // Rule 4: graph is still not bipartite

visit(@1, 1).                                                          // Initial facts
unvisited(A).
```

Figure 3.10: *Bipartiteness Checking program.*

A bipartite graph is one where in every $\texttt{edge}(a, b)$, there is a valid assignment that makes a member of set C_1 or C_2 and node b member of either C_2 or C_1 respectively.

Variant 3.5.1 (Bipartiteness Convergence)
We now reason from the application of the program rules. After each application of an inference rule, one of the following events will happen:

1. Set U will decrease and set C_1 or C_2 will increase, with a potential increase in the number of `visit` facts.
2. Set C_1 or C_2 will stay the same, while the number of `visit` facts will be reduced.
3. Set C_1 or C_2 will decrease and set F will increase, while the number of `visit` facts will be reduced.
4. Set F will stay the same, while the number of `visit` facts decreases.

Proof. Trivially from the rules. □

From this variant, it can be inferred that set U never increases in size and in a node transition from `uncolored` to `colored`, the database may increase in size. For every other rule application, the database of facts always decreases. This also means that the program will eventually terminate, since it is limited by the number of `visit` facts that can be generated.

Theorem 3.5.1 (Bipartiteness Correctness)

If the graph is connected and bipartite then the nodes will be partitioned into sets C_1 and C_2, while sets F and U will be empty.

Proof. By induction, we prove that uncolored nodes become part of either C_1 or C_2 and, if there is an edge between nodes in the two sets then they have different colors.

In the base case, we start with empty sets but node `@1` is made member of C_1. Rule 1 sends `visit` facts to the neighbors of `@1`, forcing them to be members of C_2.

In the inductive case, we have sets C_1' and C_2' with some nodes already colored. From Variant 3.5.1, we know that U always decreases. Since the graph is bipartite, events 3 and 4 never happen since there is a possible partitioning of nodes. With event 1, we have set $C_1 = C_1' \cup \{n\}$, (or C_2) where n is the new colored node. With event 2, the sets remain the same. Since the graph is connected, every node will be colored, therefore event 1 will happen for every node of the graph.

□

Theorem 3.5.2 (Bipartiteness Failure)
If the graph is connected but not bipartite then some nodes will be part of set F.

Proof. Assume that the algorithm completely partitions the nodes into sets C_1 and C_2 and thus F is empty. Since the graph is connected, we know that the algorithm tries to build a valid partitioning represented by C_1 and C_2. This is a contradiction because the graph is not bipartite (by definition) and thus at least one node will be part of set F with rule 3.

□

3.5.2 Synchronous PageRank

PageRank [Pag01] is a well known graph algorithm that is used to compute the relative relevance of web pages. The standard formulation of the PageRank algorithm uses three $n \times n$ matrices (where n is the number of pages):

- An adjacency matrix A, where A_{ij} is 1 if page i has an outgoing link to j;
- A transition matrix P, where $P_{ij} = A_{ij}/deg(i)$ and $\deg(i)$ is the number of outgoing links for page i;
- A 'Google matrix' G, where $G = \alpha P + (1-\alpha)I$, where α is the *damping factor* and I is a $n \times n$ matrix where every I_{ij} is 1. The damping factor is the probability of a user jumping to a random page instead of following the links of the current page.

The PageRank vector x of size $1 \times n$ is the solution to the following linear system:

$$x = Gx \tag{3.12}$$

To solve this problem, it is possible to use an iterative method by starting with an initial vector $x(0)$ where all pages start with the same value (that adds up to 1) and then perform the following computation:

$$x(t+1) = Gx(t) \tag{3.13}$$

To compute the PageRank value of a single page, it is possible to use the following formula:

$$x_i(t+1) = G_i x(t) \tag{3.14}$$

where G_i is the row of the G matrix that corresponds to the inbound links of page i. Note that the full $x(t)$ vector is not required since most elements of G_i are 0.

The LM code for this iterative computation is shown in Fig. 3.11 and the code starts by declaring six predicates, namely: `outbound`, that specifies outbound links where the third argument represents a value in the transition matrix P; `numInbound`, with the number of inbound links; `pagerank`, to represent the PageRank of a page; `accumulator`, to accumulate the incoming neighbor's PageRank values; `neighbor-pagerank`, to represent an incoming PageRank value; and `start` to initialize a node. Since this program is synchronous, the PageRank values of all nodes must be computed before the next PageRank is computed.

We use constant definitions provided by LM (lines 8-10) to refer to constant values used throughout the program. The `damping` constant is the damping factor α used in the PageRank calculations. The constant `iterations` reads the number of iterations to execute from the program's input arguments and `pages` is assigned to `@world`, a special constant that evaluates to the number of nodes in the program (in this case, the number of pages).

The initial PageRank representing $x_i(0)$ is initialized in the first rule (lines 12-13) along with the accumulator. All the initial PageRank values form the initial $x(0)$ vector. The second rule of the program (lines 15-17) propagates a newly computed PageRank value to all neighbors and represents a step in the iterative method for a column in the G matrix. The fact `neighbor-pagerank` informs the neighbor node about the PageRank value of node `A` for iteration `Iter + 1`. For every iteration, each node will accumulate all the `neighbor-pagerank` facts into the `accumulator` fact (lines 19-21). When all inbound neighbor PageRank values are accumulated, the third rule (lines 23-27) is derived and a PageRank value is generated for iteration `Iter`.

The synchronous version of the PageRank algorithm has a large amount of concurrency. First, the program starts on all nodes of the graph, which makes the program trivial to parallelize. However, because this is a synchronous algorithm, there is a data dependency between PageRank iterations, i.e., nodes can only compute their next iteration after they receive all the neighbor's PageRank values from the previous iteration.

Appendix F.1 further expands this section with the asynchronous version of PageRank and also includes a proof of correctness.

3.5.3 Quick-Sort

The Quick-Sort algorithm is a divide and conquer sorting algorithm that works by splitting a list of items into two sublists and then recursively sorting the two sublists. To split a list, it picks a

```
1  type outbound(node, node, float).                                  // Predicate declaration
2  type numInbound(node, int).
3  type linear pagerank(node, float Rank, int Iteration).
4  type linear accumulator(node, float Acc, int Left, int Iteration).
5  type linear neighbor-pagerank(node, node Neighbor, float Rank, int Iteration).
6  type linear start(node).
7
8  const damping = 0.85.      // probability of user following a link in the current page
9  const iterations = str2int(@arg1).                                 // iterations to compute
10 const pages = @world.                                    // number of pages in the graph.
11
12 start(A), !numInbound(A, T)              // Rule 1: initialize pagerank and accumulator
13    -o accumulator(A, 0.0, T, 1), pagerank(A, 1.0 / float(pages), 0).
14
15 pagerank(A, V, Iter),                                       // Rule 2: propagate pagerank
16 Iter < iterations
17    -o {B, W | !outbound(A, B, W) -o neighbor-pagerank(B, A, V * W, Iter + 1)}.
18
19 neighbor-pagerank(A, B, V, Iter),                // Rule 3: accumulate neighbor's value
20 accumulator(A, Acc, T, Iter)
21    -o accumulator(A, Acc + V, T - 1, Iter).
22
23 accumulator(A, Acc, 0, Iter),                            // Rule 4: generate new pagerank
24 !numInbound(A, T),
25 V = damping + (1.0 - damping) * Acc,
26 Iter <= iterations
27    -o pagerank(A, V, Iter), accumulator(A, 0.0, T, Iter + 1).
28
29 start(A).                                                                  // Initial facts
```

Figure 3.11: *Synchronous PageRank program.*

pivot element and puts the items that are smaller than the pivot into the first sublist and the items greater than the pivot into the second list.

The Quick-Sort algorithm is interesting because it does not map immediately to the graph-based model of LM and will demonstrate that LM supports applications with dynamic graphs. The LM program starts with a single node where the initial unsorted list is located. Then the list is split as usual and two nodes are created that will recursively sort the sublists. Interestingly, this looks similar to a call tree in a functional programming language.

Figure 3.12 presents the code for the Quick-Sort algorithm in LM. The code uses six predicates described as follows: `down` represents a list that needs to be sorted; `up` is the result of sorting an `down` list; `sorted` represents a sorted sublist; `back` connects a node that is sorting a sublist to its parent node; `split` is used for splitting a list into two sublists using a pivot element; and `waitpivot` waits for two sorted sublists.

For each sublist, we start with a `down` fact that later must be, eventually, transformed into an `up` fact with the sublist sorted. In line 36 we start with the initial list at node `@0`. If the list has a small number of items (two or less), then rules at lines 8-14 will immediately sort it, otherwise the rule in line 16 is applied to split the list in 2 sublists. The fact `split` first splits the list

using the pivot `Pivot` using rules in lines 23-27. When there is nothing else to split, the rule in lines 19-21 uses an exists construct to create nodes `B` and `C` and then the sublists are sent to nodes `B` and `C` using `down` facts. `back` facts are also derived to be used to send the sorted list back to the parent node using the rule in line 34.

When two sublists are sorted, two `sorted` facts are derived that must be matched against `waitpivot` in the rule in lines 29-32. The sorted sublists are appended and send up to the parent node via the derivation of an `up` fact (line 32).

```
1  type linear down(node, list int).                                  // Predicate declaration
2  type linear up(node, list int).
3  type linear sorted(node, node, list int).
4  type linear back(node, node).
5  type linear split(node, int list int, list int, list int).
6  type linear waitpivot(node, int, node, node).
7
8  down(A, []) -o up(A, []).                                            // Rule 1: empty list
9
10 down(A, [X]) -o up(A, [X]).                                  // Rule 2: single element list
11
12 down(A, [X, Y]), X < Y -o up(A, [X, Y]).                       // Rule 3: two element list
13
14 down(A, [X, Y]), X >= Y -o up(A, [Y, X]).                      // Rule 4: two element list
15
16 down(A, [Pivot | Xs])                        // Rule 5: lists with more than two elements
17    -o split(A, Pivot, Xs, [], []).
18
19 split(A, Pivot, [], Smaller, Greater) -o       // Rule 6: create nodes to sort sublists
20    exists B, C. (back(B, A), back(C, A),
21                  down(B, Smaller), down(C, Greater), waitpivot(A, Pivot, B, C)).
22
23 split(A, Pivot, [X | Xs], Smaller, Greater), X <= Pivot         // Rule 7: split case 1
24    -o split(A, Pivot, Xs, [Y | Smaller], Greater).
25
26 split(A, Pivot, [X | Xs], Smaller, Greater), X > Pivot          // Rule 8: split case 2
27    -o split(A, Pivot, Xs, Smaller, [Y | Greater]).
28
29 waitpivot(A, Pivot, NodeSmaller, NodeGreater),                 // Rule 9: merge sublists
30 sorted(A, NodeSmaller, Smaller),
31 sorted(A, NodeGreater, Greater)
32    -o up(A, Smaller ++ [Pivot | Greater]).
33
34 up(A, L), back(A, B) -o sorted(B, A, L).                 // Rule 10: send list to parent
35
36 down(@0, initial_list).                                                // Initial facts
```

Figure 3.12: *Quick-Sort program written in LM.*

The use of the exists construct allows the programmer to create new nodes where facts can be derived. In the case of the Quick-Sort, it allows the program to create a tree of nodes where sorting can take place concurrently.

The amount of concurrency available in the Quick-Sort program depends on the quality of the selected pivot. If the pivot splits the list in equal parts, then there is more concurrency because it is now possible to work on the two halves of the list concurrently. If a bad pivot is selected, then we may end up in situations where the pivot is the smallest (or largest) element of the list, splitting the list into an empty list and a list with $n - 1$ elements. It is clear that the amount of work required to sort the empty list is much smaller than the work required to sort the larger list. Repeatedly choosing a bad pivot will effectively turn Quick-Sort into a sequential algorithm. This is not surprising since it is directly related to the Quick-Sort's average and worst case complexity, $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^2)$, respectively.

The proof of correctness for Quick-Sort follows a different style than the proofs done so far. Instead of proving invariants, we prove what happens to the database given the presence of some logical facts.

Lemma 3.5.1 (Split lemma)
If a $\texttt{split}(A, Pivot, L, Small, Great)$ fact exists then it will be consumed to derive a $\texttt{split}(A, Pivot, [], Small'{+}{+}Small, Great'{+}{+}Great)$ fact, where the elements of $Small'$ are lesser or equal than $Pivot$ and the elements of $Great'$ are greater than $Pivot$.

Proof. By induction on the size of L. □

Theorem 3.5.3 (Sort theorem)
If a $\texttt{down}(A, L)$ fact exists then it will be consumed and a $\texttt{up}(A, L')$ fact will be derived, where L' is the sorted list of L.

Proof. By induction on the size of L.

The base cases are proven trivially (rules 1-4).

In the inductive case, only rule 5 applies:

```
down(A, [Pivot | Xs]) -o split(A, Pivot, Xs, [], []).
```

which necessarily derives a $\texttt{split}(A, Pivot, Xs, [], [])$ fact. By applying the split lemma, a $\texttt{split}(A, Pivot, [], Smaller, Greater)$ fact is generated, from which only rule 6 can be used:

```
split(A, Pivot, [], Smaller, Greater) -o
   exists B, C. (back(B, A), back(C, A),
                 down(B, Smaller), down(C, Greater), waitpivot(A, Pivot, B, C)).
```

which necessarily derives $\texttt{back}(B, A)$, $\texttt{back}(C, A)$, $\texttt{down}(B, Smaller)$, $\texttt{down}(C, Greater)$ and also a $\texttt{waitpivot}(A, Pivot, B, C)$ fact. The semantics of LM ensure that B and C are fresh node addresses, therefore those new facts will be derived on nodes with no facts. The lists $Smaller$ and $Greater$ are both smaller (in size) than L, so, by the the induction hypothesis, an $\texttt{up}(B, Smaller')$ and an $\texttt{up}(C, Greater')$ facts are derived. These last two facts will be used in the following rule:

```
up(A, L), back(A, B) -o sorted(B, A, L).
```

which generates a `sorted`$(A, B, Smaller')$ and a `sorted`$(A, C, Greater')$ facts. In the continuation, there is only one rule that accepts `sorted` and `waitpivot` facts:

```
waitpivot(A, Pivot, NodeSmaller, NodeGreater),
sorted(A, NodeSmaller, Smaller),
sorted(A, NodeGreater, Greater)
   -o up(A, Smaller ++ [Pivot | Greater]).
```

returning up$(A, Smaller' + +[Pivot|Greater'])$. We know that $Smaller' + +[Pivot|Greater']$ is sorted since $Small'$ contains the sorted list of elements lesser or equal than $Pivot$ and $Greater'$ the elements greater than $Pivot$.

□

3.6 Related Work

3.6.1 Graph-Based Programming Models

Many programming systems have designed for writing graph-based programs. Some good examples are the Dryad, Pregel, GraphLab, Ligra, Grace, Galois and SociaLite systems.

The Dryad system [IBY+07] combines computational vertices with communication channels (edges) to form a data-flow graph. The program is scheduled to run on multiple processors or cores and data is partitioned during runtime. Routines that run on computational vertices are sequential, with no synchronization. Dryad is better suited for a wider range of problem domains than LM, however Dryad is not a language but a framework upon which languages such as LM could be built.

The Pregel system [MAB+10] is also graph based, although programs have a more strict structure. They must be represented as a sequence of iterations where each iteration is composed of computation and message passing. Pregel is especially suited for large graphs since it aims to scale to large architectures. LM does not impose iteration restrictions on programs, however it is not as well suited for large graphs as Pregel.

GraphLab [LGK+10] is a C++ framework for developing parallel machine learning algorithms. While Pregel uses message passing, GraphLab allows nodes to have read/write access to different scopes through different concurrent access models in order to balance performance and data consistency. While some programs only require access to the local node's data, others may need to update edge information. Each consistency model will provide different guarantees that are better adapted to some algorithms. GraphLab also provides different schedulers that dictate the order in which nodes are computed. The downside of GraphLab is that GraphLab has a steep learning curve and programs must be written in C++, requiring a lot of boilerplate code. LM programs tend to be smaller and easier to reason about since they are written at a higher abstraction level.

Ligra [SB13] is a lightweight framework for large scale graph processing on a single multi core machine. Ligra exploits the fact that most huge graph datasets available today can be made to fit in the main memory of commodity servers. Ligra is a simple framework that exposes two

main interfaces: `EdgeMap` and `VertexMap`. The former applies a function to a subset of edges of the graph, while the latter applies a function to a subset of vertices. The functions passed as arguments are applied to either a single edge or a single vertex and the user must ensure that the function can be executed in parallel. The framework allows the use of *Compare-and-Swap* (CAS) instructions when implementing functions in order to avoid race conditions.

Grace [WXDG13] is another graph-based framework for multi core machines. Unlike Ligra, Grace programs are implemented from the point of view of a vertex. Each vertex and edge can be customized with different data depending on the target application. By default, programs are executed iteratively: for each iteration the vertex program reads incoming edge messages, performs computation and sends messages to the outbound edges. Since iterative programs require synchronization after each iteration, Grace allows the user to relax these constraints and implement customizable execution policies by implementing code for describing which vertices are allowed to run and in which order. The order is dictated by assigning a *scheduling priority value*.

Galois [PNK+11] is a parallel programming model with irregular applications based on graphs, trees and sets. A Galois parallel algorithm is viewed as a parallel application of an *operator* over an irregular data structure which generate *activities* on the data structure. Such operator may, for instance, be applied to the node of the graph in order to change its data or change the structure of its neighborhood, allowing for data structure changes. In Galois, *active elements* are nodes of the graph where computation needs to be performed. Each operator performed on an active element needs to acquire the *neighborhood*, which are the nodes connected to the active element that are involved in the operator's computation. Furthermore, operators are selected for execution according to some specific *ordering*. From the point of view of the programmer, the active elements are represented in a work-list, while operators can be implemented on top of work-list's iterators. Galois supports speculative execution by allowing operator rollback when an operator requires a node that is held by another operator.

Ligra, Grace and Galois are not programming language but frameworks built on top of other programming languages such as C and Java. Naturally, programs written in these frameworks need to take into account several implementation details such as *compare-and-set* instructions, work-lists, and scheduling order. LM programs are more abstract since reasoning is performed around logical rules and logical facts which are much closer to the problem domain of graph-based programs.

SociaLite [SPSL13] is a Datalog-based language for writing algorithms for social-network analysis. SociaLite takes Datalog programs and compiles them to efficient distributed Java code. The language places some restrictions on rules in order to make distribution possible, however, the programmer is free to tell the system how to shard database's facts. Like LM, these restrictions deal with the first argument of each fact, however, the first argument is not related to a graph abstraction but is instead related to fact distribution, allowing programmers to optimize how messages are sent between machines.

3.6.2 Sensor Network Programming Languages

Many programming languages have been designed to help developers write programs that run on sensor networks, which are networks that can be represented as graphs. Programming languages such as Hood [WSBC04], Tinydb [MFHH05] or Regiment [NMW07] have been proposed, pro-

viding support for data collection and aggregation over the network. These systems assume that the network remains static and nodes stay in place.

Other languages such as Pleiades [KGMG07], LDP [RGL+08] or Proto [BB06] go beyond static networks and support dynamic reconfiguration. In Pleiades, the programmer writes an application from the point of view of the whole sensor network and the compiler transforms it into code that can be run on each individual node. LDP is a language derived from a method for distributed debugging, that allows it to efficiently detect conditions on variably-sized groups of nodes. It is based on the tick model, generating a new set of condition matchers throughout the ensemble on each tick. Like Pleiades and LDP, Proto also compiles global programs into locally executed code.

Finally, the original Meld [ARLG+09] is also a language designed for dynamic networks, namely, ensembles of robots. As we have seen before, Meld is a logic programming language where programs are sets of logical rules that infer over the state of the ensemble. Meld supports action facts and sensing facts, which allow the robots to act on the world or sense the world, respectively. Like Pleiades programs, Meld programs are also written from the point of the view of the whole ensemble.

3.6.3 Constraint Handling Rules

Since LM is a bottom-up linear logic programming language, it also shares similarities with Constraint Handling Rules (CHR) [BF05, BF13]. CHR is a concurrent committed-choice constraint language used to write constraint solvers. A CHR program is a set of rules and a set of constraints. Constraints can be consumed or generated during the application of rules. Unlike LM, in CHR there is no concept of rule priorities, but there is an extension to CHR that supports them [DKSD07]. There is also another CHR extension that adds persistent constraints and it has been proven to be sound and complete [BRF10] in relation to the standard formulation of CHR.

3.6.4 Graph Transformation Systems

Graph Transformation Systems (GTS) [EP04], commonly used to model distributed systems, perform rewriting of graphs through a set of graph productions. GTS also introduces concepts of concurrency, where it may be possible to apply several transformations at the same time. In principle, it should be possible to model LM programs as graph transformations: we directly map the LM graph of nodes to GTS's initial graph and consider logical facts as nodes that are connected to LM's nodes. Each LM rule is then a graph production that manipulates the node's neighbors (the database) or sends new facts to other nodes. On the other hand, it is also possible to embed GTS inside CHR [RF11].

3.7 Chapter Summary

In this chapter, we gave an overview of the LM language, including its syntax and operational semantics. We also explained how to write programs using all the facilities provided by LM,

including linear facts, comprehensions, and aggregates. We also explained how to informally prove the correctness of several LM programs.

Chapter 4

Logical Foundations: Abstract Machine

This chapter provides an overview of the proof theoretic basis behind LM and the dynamic semantics of the language. First, we will present the subset of linear logic on which LM is built. Second, we present the high level dynamic semantics, i.e., how rules are evaluated and how node communication is done, followed by the low level dynamics, a close representation of how the virtual machine runs and we prove that the low level dynamic semantics are sound in relation to the high level dynamic semantics. The presence of rule priorities, comprehensions and aggregates makes the low level dynamic semantics not complete, since these are features are not represented directly by the underlying linear logic system. This is the same approach used in Prolog, which includes useful features such as the *cut operator* and a depth-first search strategy that make the language sound but not complete. Note that the semantics presented in this chapter do not take into account coordination or thread-based facts. This chapter may be skipped without loss of understanding of the main thesis of the dissertation.

4.1 Linear Logic

Logic, as classically understood, treats true propositions as *persistent truth*. When a persistent proposition is needed to prove other propositions, it can be reused as many times as we wish because it is true indefinitely. This is also true in the constructive or intuitionistic school of logic. Linear logic is a *substructural logic* (lacks weakening and contraction) developed by Girard [Gir95] that extends persistent logic with linear propositions which can be understood as ephemeral resources that can be used only once to prove other propositions. Due to the resource interpretation of the logic, linear logic presents a good basis for implementing a structured way of managing state in programming languages [Mil85]. Linear logic has also been used in game semantics [LS91, Bla92], concurrent programming [LPPW05b, MZ10, PCPT12], knowledge representation [Bos11], or narrative generation [MFBC14, MBFC13].

In the context of the Curry-Howard correspondence [How80], linear logic has been applied in programming languages as a mechanism to implement *linear types*. Linear types force objects to be used exactly once. Surprisingly, such types add mutable state to functional languages because they enforce a linear view of state, allowing the language to naturally support concurrency, input/output and data structure's updates. Arguably, the most popular language that features

uniqueness types is the Clean programming language [AP95]. Monads [Wad97], made popular with the Haskell programming language, are another interesting way to add state to functional languages. Monads tend to be more powerful than linear types as they also ensure equational reasoning in the presence of mutable data structures and I/O effects.

Linear logic programming is a different approach than either monads or linear types. While the latter are mechanisms that enhance functional programming with state, the former uses state as a foundation, since computation is driven forward through the manipulation of state.

Traditional forward-chaining logic programming languages like Datalog only use persistent logic, however many ad-hoc extensions [Liu98, LHL95] have been devised to support state updates, but most are extra-logical which makes the programs harder to reason about. LM uses linear logic as its foundation, therefore state updates are natural to the language.

In linear logic, truth is treated as a resource that is consumed once it is used. For instance, in the graph visit program in Fig. 3.7, the `unvisited(A)` and `visit(A)` linear facts are consumed in order to prove `visit(A)`. If those facts were persistent, then the rule would make no sense, because the node would be `visited` and `unvisited` at the same time.

4.1.1 Sequent Calculus

We now describe the linear logic fragment used as a basis for LM. Note that in this thesis we follow the intuitionistic approach and use the sequent calculus [Gen35] to specify the logic. Our initial sequent is written as $\Psi; \Gamma; \Delta \vdash C$ and can be read as "assuming persistent resources Γ and linear resources Δ then C is true". More specifically, Ψ is the typing context which contains unique variables, Γ is a multi-set of persistent resources, Δ is a multi-set of linear resources while C is the proposition we want to prove. The sequent can also be decomposed into a *succedent* (C) and the *antecedents* (contexts that appear before $\vdash$).

We now present the connectives and their associated rules for the linear logic fragment. First, we have the *simultaneous conjunction* $A \otimes B$ that packages linear resources together. In the right rule, $A \otimes B$ is true if both A and B are true, and, in the left rule, it is possible to split $A \otimes B$ apart.

$$\frac{\Psi; \Gamma; \Delta \vdash A \quad \Psi; \Gamma; \Delta' \vdash B}{\Psi; \Gamma; \Delta, \Delta' \vdash A \otimes B} \otimes R \qquad \frac{\Psi; \Gamma; \Delta, A, B \vdash C}{\Psi; \Gamma; \Delta, A \otimes B \vdash C} \otimes L$$

Note that the inference rules above can be decomposed into premises (the sequents above the separator line) and conclusion (the sequent below the line).

Next, we have the *additive conjunction* $A \,\&\, B$ that allows us to select between A or B. In the right rule we must prove A and B using the same resources, while in the left rule, we can select one of the resources.

$$\frac{\Psi; \Gamma; \Delta, A \vdash C}{\Psi; \Gamma; \Delta, A \,\&\, B \vdash C} \,\&L_1 \qquad \frac{\Psi; \Gamma; \Delta, B \vdash C}{\Psi; \Gamma; \Delta, A \,\&\, B \vdash C} \,\&L_2 \qquad \frac{\Psi; \Gamma; \Delta \vdash A \quad \Psi; \Gamma; \Delta \vdash B}{\Psi; \Gamma; \Delta \vdash A \,\&\, B} \,\&R$$

To express inference, we introduce the *linear implication* connective written as $A \multimap B$. For the right rule, we prove $A \multimap B$ by assuming A and then proving B, while in the left rule, we obtain B by using some linear resources to prove A.

$$\frac{\Psi;\Gamma;\Delta, A \vdash B}{\Psi;\Gamma;\Delta \vdash A \multimap B} \multimap R \quad \frac{\Psi;\Gamma;\Delta \vdash A \quad \Psi;\Gamma;\Delta', B \vdash C}{\Psi;\Gamma;\Delta, \Delta', A \multimap B \vdash C} \multimap L$$

Next, we introduce persistent resources written as $!A$. For the right rule, we prove $!A$ by proving it without any linear resources. Likewise, to use a persistent resource, we simply drop the !. There is also a copy rule that moves persistent resources from Γ to Δ. Remember that Γ contains persistent resources.

$$\frac{\Psi;\Gamma;\cdot \vdash A}{\Psi;\Gamma;\cdot \vdash !A} !R \quad \frac{\Psi;\Gamma, A;\Delta \vdash C}{\Psi;\Gamma;\Delta, !A \vdash C} !L \quad \frac{\Psi;\Gamma, A;\Delta, A \vdash C}{\Psi;\Gamma, A;\Delta \vdash C} \text{copy}$$

Another useful connective is the *multiplicative unit* of the $\otimes$ connective. It is written as $\mathbf{1}$ and is best understood as something that does not need any resource to be proven.

$$\frac{}{\Psi;\Gamma;\cdot \vdash \mathbf{1}} \mathbf{1}R \quad \frac{\Psi;\Gamma;\Delta \vdash C}{\Psi;\Gamma;\Delta, \mathbf{1} \vdash C} \mathbf{1}L$$

Next, we introduce the *quantification* connectives, namely *universal quantification* $\forall_{n:\tau}.A$ and *existential quantification* $\exists_{n:\tau}.A$ ($n : \tau$ means that n has type τ). These connectives use the typing context Ψ to introduce and read term variables. The right and left rules of those two connectives are dual.

$$\frac{\Psi, m : \tau;\Gamma;\Delta \vdash A\{m/n\}}{\Psi;\Gamma;\Delta \vdash \forall_{n:\tau}.A} \forall R \quad \frac{\Psi \vdash M : \tau \quad \Psi;\Gamma;\Delta, A\{M/n\} \vdash C}{\Psi;\Gamma;\Delta, \forall_{n:\tau}.A \vdash C} \forall L$$

$$\frac{\Psi \vdash M : \tau \quad \Psi;\Gamma;\Delta \vdash A\{M/n\}}{\Psi;\Gamma;\Delta \vdash \exists_{n:\tau}.A} \exists R \quad \frac{\Psi, m : \tau;\Gamma;\Delta, A\{m/n\} \vdash C}{\Psi;\Gamma;\Delta, \exists_{n:\tau}.A \vdash C} \exists L$$

The judgment $\Psi \vdash M : \tau$ introduces a new term M with type τ that does not depend on Γ or Δ but may depend on the variables in Ψ. In rules $\forall R$ and $\exists L$, the new m variable introduced in Ψ must always be *fresh*. We complete the linear logic system with the *cut rules* and the *identity rule*:

$$\frac{\Psi;\Gamma;\Delta \vdash A \quad \Psi;\Gamma;\Delta', A \vdash C}{\Psi;\Gamma;\Delta, \Delta' \vdash C} cut_A \quad \frac{\Psi;\Gamma;\cdot \vdash A \quad \Psi;\Gamma, A;\Delta \vdash C}{\Psi;\Gamma;\Delta \vdash C} cut!_A$$

$$\frac{}{\Psi;\Gamma;A \vdash A} id_A$$

4.1.2 From The Sequent Calculus To LM

The connection between LM and the sequent calculus fragment is presented in the Table 4.1. In the table, we show how each connective is translated into LM's abstract syntax and then into LM programs. In order to understand how LM rules are related to the sequent calculus, consider the first rule of the graph visit program shown in Fig 3.7:

Connective	Description	LM Syntax	LM Place	LM Example
$fact(\hat{x})$	Linear atomic propositions.	$fact(\hat{x})$	LHS or RHS	`path(A, P)`
$!fact(\hat{x})$	Persistent atomic propositions.	$!fact(\hat{x})$	LHS or RHS	`!edge(X, Y, W)`
1	Represents rules with an empty RHS.	1	RHS	`1`
$A \otimes B$	Connect two expressions.	A, B	LHS and RHS	`path(A, P), edge(A, B, W)`
$\forall x.A$	To represent variables defined inside the rule.	Please see $A \multimap B$	Rule	`path(A, B) -o reachable(A, B)`
$\exists x.A$	Instantiates new node variables.	$\text{exists}_{\hat{x}}.B$	RHS	`exists A.(path(A, P))`
$A \multimap B$	$\multimap$ means "linearly implies". A is the rule's LHS and B is the RHS.	$A \multimap B$	Rule	`path(A, B) -o reachable(A, B)`
$!C$	Constraint.	$A = B$	LHS	`A = B`
$\mathcal{R}_{comp}^{(\widehat{V},M)}$	For comprehensions (M is not used). For aggregates (M accumulates). $\widehat{V}$ captures rule variables.	$\{\ \hat{x} \mid A \multimap B\ \}$	RHS	`{B | !edge(A, B) -o visit(B)}`

Table 4.1: *Connectives from linear logic and their use in LM.*

```
visit(A),
unvisited(A)
   -o visited(A),
      {B | !edge(A, B) -o visit(B)}.
```

This rule is translated to a sequent calculus proposition, as follows:

$$\forall_A.(\texttt{visit}(A) \otimes \texttt{unvisited}(A) \multimap \texttt{visited}(A) \otimes \mathcal{R}_{comp}^{(A)}) \tag{4.1}$$

First, the rule's variable A is included using the $\forall$ connective. The rule's LHS and RHS are connected using the $\multimap$ connective. The comprehension is transformed into $\mathcal{R}_{comp}^{(A)}$, which is a *recursive* term that is assigned to an unique name, namely, comp. This name is related to the following persistent term:

$$!\forall_A.(\mathcal{R}_{comp}^{(A)} \multimap (\mathbf{1} \ \& \ (\forall_B.(!\texttt{edge}(A,B) \multimap \texttt{visit}(B)) \otimes \mathcal{R}_{comp}^{(A)}))) \tag{4.2}$$

Notice that the enclosing $\forall$ includes all the arguments of the unique name in order to pass around variables from outside the definition of the comprehension, in this case variable A. The persistent term allows the implication of the comprehension to be derived as many times as needed. However, the argument list can also be used to implement aggregates. Recall the PageRank aggregate example shown before:

```
update(A),
!numInbound(A, T)
   -o [sum => V; B, Val, Iter | neighbor-pagerank(A, B, Val, Iter), V =
         Val/float(T) -o neighbor-pagerank(A, B, Val, Iter) -> sum-ranks(A, V)].
```

This rule is translated into a linear logic proposition as shown next:

$$\forall_A.\forall_T.(\texttt{update}(A) \otimes !\texttt{numInbound}(A,T) \multimap \mathcal{R}_{agg}^{(A,T,0)}) \tag{4.3}$$

The persistent term for agg is defined as follows:

$$!\forall_A.\forall_T.\forall_S.(\mathcal{R}_{agg}^{(A,T,S)} \multimap \texttt{sum-ranks}(A,S)\&$$
$$(\forall_V.\forall_B.\forall_{Val}.\forall_{Iter}.(\texttt{neighbor-pagerank}(A,B,Val,Iter) \otimes !V = Val/\texttt{float}(T) - o$$
$$\texttt{neighbor-pagerank}(A,B,Val,Iter) \otimes \mathcal{R}_{agg}^{(A,T,S+V)}))) \quad (4.4)$$

The argument S of $\mathcal{R}_{agg}$ accumulates the PageRank values of the neighborhood by consuming `neighbor-pagerank` and re-deriving a new $\mathcal{R}_{agg}$ with $S{+}V$. Once the aggregate is complete, we simply select $\texttt{sum-ranks}(A,S)$ instead. As an aside, note how the constraint are translated into a persistent term of the form $!V = Val/\texttt{float}(T)$ since it does not require any fact to be proven true. This recursive mechanism is inspired in Baelde's work on *fix points* [BM07, Bae12], which allows the introduction of recursive definitions into a consistent fragment of linear logic.

4.2 High Level Dynamic Semantics

In this section, we present the high level dynamic (HLD) semantics of LM. HLD formalizes the mechanism of matching rules and deriving new facts. HLD semantics present a simplified overview of the dynamics of the language that are closer to the sequent calculus (presented in Section 4.1.1) than the implementation principles of the virtual machine. The low level dynamic (LLD) semantics are much closer to a real implementation and represent the operational semantics of the language. Both the HLD and LLD semantics model local computation at the node level.

Note that neither HLD nor LLD model the use of variable bindings when matching facts from the database. The formalization of bindings tends to complicate the formal system and it is not necessary for a good understanding of the system. Therefore, when we write an atomic proposition as p, we assume a proposition of the form $p(\widehat{x})$, where $\widehat{x}$ is the list of terms for that proposition (either values or variables), which is used for matching during the derivation process.

Starting from the sequent calculus, we consider Γ and Δ the database of our program. Γ contains the database of persistent facts while Δ the database of linear facts. We assume that the rules of the program are persistent linear implications of the form $!(A \multimap B)$ that can be used several times. However, we do not put the rules in the Γ context but in a separate context Φ. The persistent terms associated with each comprehension and aggregate are put in the Π dictionary that maps the unique name of the comprehension/aggregate to the persistent term.

The main idea of the dynamic semantics is to ignore the right side of the sequent calculus and then use *focusing* [And92] on the implications in Φ so that we only have atomic facts (e.g., the database of facts). Focusing is a proof search mechanism that prunes the proof search by constructing proofs using two alternating phases called *chaining* and *inversion*. During inversion, propositions are decomposed into smaller propositions using *invertible rules* (which are sequent rules where the premises of the rule are derivable whenever the conclusion is derivable). In chaining, a proposition is in focus and *non-invertible rules* are applied to that proposition. Propositions have either negative or positive *polarity*, if either the right rule of a given connective is invertible or not, respectively. Atomic propositions can be either negative or positive. For our case, we make them positive so that the succedent of the sequent is ignored and chaining

proceeds by focusing on implications (rules) $A \multimap B \in \Phi$ where the antecedents A must be already in Δ.

4.2.1 Step

Operationally, LM proceeds in *steps*. A step happens at some node i and proceeds by picking one rule to apply, matching the rule's LHS against the database, removing all those facts and then deriving all the constructs in the rule's RHS. We assume the existence of n nodes in the program and that Γ and Δ are split into $\Gamma_1, \ldots, \Gamma_n$ and $\Delta_1, \ldots, \Delta_n$ respectively. After each step, the database of each fact is updated accordingly.

Steps are defined as $\mathsf{step}\ \Gamma; \Delta; \Phi \Longrightarrow \Gamma'; \Delta'$, where Γ' and Δ' are the new database contexts and Φ are the derivation rules of the program. The step rule is as follows:

$$\frac{\mathsf{run}^{\Gamma_i;\Pi}\ \Delta_i; \Phi \to \Xi'; [\Gamma'_1; \cdots; \Gamma'_n]; [\Delta'_1; \cdots; \Delta'_n]}{\mathsf{step}\ [\Gamma_1; \cdots; \Gamma_i; \cdots; \Gamma_n]; [\Delta_1; \cdots; \Delta_i; \cdots; \Delta_n]; \Phi} \ \mathsf{step}$$
$$\Longrightarrow$$
$$[\Gamma_1, \Gamma'_1; \cdots; \Gamma_i, \Gamma'_i; \cdots; \Gamma_n, \Gamma'_n]; [\Delta_1, \Delta'_1; \cdots; (\Delta_i - \Xi'), \Delta'_i; \cdots; \Delta_n, \Delta'_n]$$

A step is then a local derivation of some rule at a given node i. The effects of a step result in the addition and retraction of facts from node i and also the assertion of facts in *remote* nodes. In the rule above, this is represented by the contexts Γ'_j and Δ'_j.

4.2.2 Application

A step is performed through $\mathsf{run}^{\Gamma;\Pi}\ \Delta; \Phi \to \Xi'; \Gamma'; \Delta'$. Γ, Δ, Φ and Π have the meaning explained before, while Ξ', Γ' and Δ' are output multi-sets from applying one of the rules in Φ and are sometimes written as $\mathcal{O}$ for conciseness. Ξ' is the multi-set of retracted linear facts, Γ' is the set of derived persistent facts and Δ' is the multi-set of derived linear facts. Note that for HLD semantics there is no concept of rule priority, therefore a rule is picked non-deterministically.

The judgment $\mathsf{app}^{\Gamma;\Pi}_{\Psi}\ \Delta; \Pi \to \mathcal{O}$ applies one derivation rule. First, it non-deterministically chooses the correct term values to the $\forall$ variables and then splits the Δ context into Δ_1 and Δ_2, namely the multi-set of linear facts consumed to match the rule's LHS (Δ_1) and the remaining linear facts (Δ_2). The Ψ context is used to store a substitution that maps variables to values. In the next section, we will see how LLD semantics deterministically calculate Δ_1 and Ψ.

$$\frac{\mathsf{m}^{\Gamma}\ \Delta_1 \to A \quad \mathsf{der}^{\Gamma;\Pi}\Delta_2; \Delta_1; \cdot; \cdot; B \to \mathcal{O}}{\mathsf{app}^{\Gamma;\Pi}_{\Psi}\ \Delta_1, \Delta_2; \Pi \to \mathcal{O}} \ \mathsf{app\ rule}$$

$$\frac{\mathsf{app}^{\Gamma;\Pi}_{\Psi, m:M:\tau}\ \Delta; \Pi \to \mathcal{O} \quad \Psi \vdash M : \tau}{\mathsf{app}^{\Gamma;\Pi}_{\Psi}\ \Delta; \Pi \to \mathcal{O}} \ \mathsf{app}\ \forall$$

$$\frac{\mathsf{app}^{\Gamma;\Pi}_{\cdot}\ \Delta; \Pi \to \mathcal{O}}{\mathsf{run}^{\Gamma;\Pi}\ \Delta; R, \Phi \to \mathcal{O}} \ \mathsf{run\ rule}$$

4.2.3 Match

The $\mathsf{m}^{\Gamma}\ \Delta \to C$ judgment uses the right (R) rules of the sequent calculus in order to match (prove) the term C using Γ and Δ. We must consume all the linear facts in the multi-set Δ when matching C. The context Γ may be used to match persistent terms in C but such facts are never retracted since they are persistent.

$$\frac{}{\mathsf{m}^{\Gamma}\ \cdot \to \mathbf{1}}\ \mathsf{m}\mathbf{1}$$

$$\frac{}{\mathsf{m}^{\Gamma}\ p \to p}\ \mathsf{m}p \qquad \frac{}{\mathsf{m}^{\Gamma,p}\ \cdot \to\ !p}\ \mathsf{m}!p$$

$$\frac{\mathsf{m}^{\Gamma}\ \Delta_1 \to A \quad \mathsf{m}^{\Gamma}\ \Delta_2 \to B}{\mathsf{m}^{\Gamma}\ \Delta_1, \Delta_2 \to A \otimes B}\ \mathsf{m}\otimes$$

4.2.4 Derivation

After successfully matching a rule's LHS, we next derive the RHS. The derivation judgment has the form $\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; \Omega \to \mathcal{O}$ with the following meaning:

- Γ the multi-set of persistent resources in the database;
- Π dictionary of persistent terms for comprehensions and aggregates;
- Δ the multi-set of linear resources in the database not yet retracted;
- Ξ the multi-set of linear resources that have been retracted while matching the rule's LHS, matching comprehensions or aggregates;
- Γ_1 the multi-set of persistent facts that have been derived using the current rule;
- Δ_1 the multi-set of linear facts that have been derived using the current rule;
- Ω an ordered list which contains the propositions of the rule's RHS that need to asserted into the database;
- $\mathcal{O}$ the output contexts, including consumed facts and derived persistent and linear facts.

The following derivation rules are a direct translation from the sequent calculus:

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; p, \Delta_1; \Omega \to \mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; p, \Omega \to \mathcal{O}}\ \mathsf{der}\ p$$

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1, p; \Delta_1; \Omega \to \mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; !p, \Omega \to \mathcal{O}}\ \mathsf{der}\ !p$$

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; A, B, \Omega \to \mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; A \otimes B, \Omega \to \mathcal{O}}\ \mathsf{der}\otimes$$

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; \Omega \to \mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; \mathbf{1}, \Omega \to \mathcal{O}}\ \mathsf{der}\ \mathbf{1}$$

$$\frac{}{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi'; \Gamma'; \Delta'; \cdot \to \Xi'; \Gamma'; \Delta'}\ \mathsf{der\ end}$$

The main rule for deriving aggregates is der agg_1. It looks into Π for the appropriate persistent term and applies $\mathcal{R}_{agg}$ to the implication and then selects the recursive case. On the other hand, the rule der agg_2 is identical but instead decides to derive the final part of the aggregate by selecting the additive conjunction's left hand side. The HLD semantics do not take into account the contents of the database to determine how many times a comprehension should be applied.

$$\frac{\begin{array}{c}\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\Sigma'+\sigma)}))))\\ \mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; \forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\Sigma+\sigma)})\{\widehat{V}/\widehat{v}\}\{\Sigma/\Sigma'\}, \Omega \to \mathcal{O}\end{array}}{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; \mathcal{R}_{agg}^{(\widehat{V},\Sigma)}, \Omega \to \mathcal{O}}\ \mathsf{der\ agg}_1$$

$$\frac{\begin{array}{c}\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\Sigma'+\sigma)}))))\\ \mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; C\{\widehat{V}/\widehat{v}\}\{\Sigma/\Sigma'\}, \Omega \to \mathcal{O}\end{array}}{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; \mathcal{R}_{agg}^{(\widehat{V},\Sigma)}, \Omega \to \mathcal{O}}\ \mathsf{der\ agg}_2$$

We do not include comprehensions here because they are a special case of aggregates.

Finally, because both comprehensions and aggregates create implications $A \multimap B$ and use the $\forall$ connective, we add a derivation rules der $\multimap$ and der $\forall$.

$$\frac{\mathsf{m}^{\Gamma}\ \Delta_a \to A \quad \mathsf{der}^{\Gamma;\Pi}\Delta_b; \Xi, \Delta_a; \Gamma_1; \Delta_1; B, \Omega \to \mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta_a, \Delta_b; \Xi; \Gamma_1; \Delta_1; A \multimap B, \Omega \to \mathcal{O}}\ \mathsf{der}\ \multimap$$

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; A\{V/x\}, \Omega \to \mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_1; \Delta_1; \forall_x.A, \Omega \to \mathcal{O}}\ \mathsf{der}\ \forall$$

4.3 Low Level Dynamic Semantics

The Low Level Dynamic (LLD) semantics remove all the non-deterministic choices in the previous dynamics and makes them deterministic. The new semantics will do the following:

- Match rules by priority order;
- Determine the set of linear facts needed to match either the rule's LHS or the LHS of comprehensions/aggregates without guessing;
- Apply as many comprehensions as the database allows.
- Apply as many aggregates as the database allows.

While the implementation presented in Chapter 5 follows the LLD semantics, there are several optimizations not implemented in LLD, such as:

- Indexing: the implementation uses indexing for looking up facts using a specific argument;
- Better candidate rules: when selecting a rule to execute, the implementation filters out rules which do not have enough facts to be derived;
- Multiple rule derivation: the LLD semantics only execute one rule at the time, while the implementation is able to derive a rule multiple times when there are no conflicting rule;
- Matching and substitution: in the implementation, matching is done implicitly using variables and comparisons, while LLD uses the Ψ context to hold substitutions.

The complete set of inference rules for the LLD semantics are presented in Appendix C.

LLD is specified as an *abstract machine* and is represented as a sequence of state transitions of the form $S_1 \quad \mapsto \quad S_2$. HLD had many different proof trees for a given triplet $\Gamma; \Delta; \Phi$ because HLD allows choices to be made during the inference rules. For instance, in HLD any rule could be selected to be executed. In LLD there is only one state sequence possible for a given $\Gamma; \Delta; \Phi$ since there is no guessing involved. LLD semantics present a complete step by step mechanism that is needed to correctly evaluate a LM program. For instance, when LLD tries to apply a rule, it will check if there are enough facts in the database and backtrack until a rule can be applied.

4.3.1 Application

LLD shares exactly the same inputs and outputs as HLD. The inputs correspond to the Γ and Δ fact contexts and the list of rules Φ, while the outputs correspond to the newly asserted facts in Γ' and Δ' and the retracted facts which are put in the Ξ' context.

The first difference between LLD and HLD start when picking a rule to derive. Instead of guessing, LLD treats the list of rules as a stack and picks the first rule R_1 to execute (the rule with the highest priority). The remaining rules are stored as a *continuation*. If R_1 cannot be matched because there are not enough facts in the database, we backtrack and use the rule continuation to pick the next rule and so on, until one rule can be successfully applied.

The machine starts with a database $(\Gamma; \Delta)$ and a list of rules Φ. The initial state is always $\texttt{infer}\ \Delta; \Phi; \Gamma$. We start by picking the first rule R_1 from Φ:

$$\texttt{infer}\ \Delta; R_1, \Phi; \Gamma \quad \mapsto \quad \texttt{apply}\ \cdot; \Delta; \Pi; \Gamma; R \qquad \text{(select rule)}$$

If, after trying all the rules, there are no remaining candidate rules, the machine enters into the next state, which means that no more rules are possible for this node and the machine should perform local computation on another node.

$$\texttt{infer}\ \Delta; \cdot; \Gamma \quad \mapsto \quad \mathsf{next}_{\Gamma;\Delta} \qquad \text{(fail)}$$

In order to try a particular rule, we either need to unfold the $\forall$ connective, by adding its variable to the Ψ context, or, initiate the matching process when reaching the $\multimap$ connective. The variables in the Ψ context, which are initially assigned to an unknown value $\_$, will later be assigned to a concrete value as the matching process goes forward.

$$\texttt{apply}\ \Psi; \Delta; \Pi; \Gamma; \forall_{x:\tau}.A \quad \mapsto \quad \texttt{apply}\ \Psi, x : \_ : \tau; \Delta; \Pi; \Gamma; A \qquad \text{(open rule)}$$

$$\texttt{apply}\ \Psi; \Delta; \Pi; \Gamma; A \multimap B \quad \mapsto \quad {}^{\Psi}\blacktriangleright^{\mathsf{m}^{\Gamma}\ \cdot \to \mathbf{1}}_{A \multimap B}\ (\Delta; \Phi); \cdot; \Gamma; \Delta; A \qquad \text{(init rule)}$$

4.3.2 Continuation Frames

The most interesting aspects introduced by the LLD machine are the *continuation frame* and the *continuation stack*. A continuation frame acts as a choice point that is created during rule matching whenever we try to match an atomic proposition against the database. The frame considers all the facts relevant to the proposition given the current context Ψ.

The frame contains enough state to resume the matching process at the time of its creation, therefore we can easily backtrack to the choice point and select the next candidate fact from the database. We keep the continuation frames in a continuation stack for backtracking purposes. If, at some point there are no candidate facts because the current variable assignments are not usable, we update the top frame to try the next candidate fact. If all candidates are exhausted, we pop the top frame and continue with the next available frame.

By using this match mechanism, we determine which facts need to be used to match a rule. Our LM implementation works like LLD, by iterating over the available facts at each choice point and then committing to the rule if the matching process succeeds. However, while the implementation only attempts to match rules when the database has all the facts required by the rule's LHS, LLD is more naïve in this aspect because it tries all rules in order.

4.3.3 Structure of Continuation Frames

We have two continuation frame types, depending on the type of the candidate facts.[1]

Linear continuation frames

There are two types of continuation frames. Linear frames use the form $(\Delta; \Delta''; p(\widehat{x}); \Omega; \Psi)$, $\mathsf{m}^{\Gamma}\ \Delta' \to \Omega'$ where:

$p(\widehat{x})$ atomic proposition that created this frame. The predicate for the proposition is p;

Δ multi-set of linear facts that are not of predicate p plus all the other candidate facts of the predicate p we have already tried, including a fact p, which is the current candidate fact;

Δ'' facts of predicate p that match $p(\widehat{x})$ which we haven't tried yet. It is a multi-set of linear facts;

Ω ordered list of remaining terms needed to match;

Δ' multi-set of linear facts we have consumed to reach this point;

Ω' terms matched already using Δ' and Γ;

Ψ substitution of variable assignments (includes variable and value).

[1] All continuation frames have an implicit Ψ context that models variable assignments, including variable names, values and their locations in the terms. This is important if we want to model variable assignments and matchings.

Persistent continuation frame

Persistent frames are slightly different since they only need to keep track of remaining persistent candidates. They are structured as $[\Gamma''; \Delta; !p(\widehat{x}); \Omega; \Psi]^{\mathsf{m}^\Gamma\ \Delta' \to \Omega'}$[2]:

$!p(\widehat{x})$ persistent atomic proposition that created this frame;

Γ'' remaining candidate facts that match $!p(\widehat{x})$;

Δ multi-set of linear facts not consumed yet;

Ω ordered list of terms needed to match past this frame;

Δ' multi-set of linear facts consumed up-to this frame;

Ω' terms matched up-to this point using Δ' and Γ;

Ψ substitution of variable assignments (includes variable and value).

We now introduce some definitions which helps define what means for a continuation frame to be well-formed.

Term equivalence

The first definition defines the equality between two multi-sets of terms. Two multi-sets A and B are equal, $A \equiv^\theta B$, when there a substitution θ that allows $A\theta$ and $B\theta$ to have the same constituent atoms. Each continuation frame builds a substitution Ψ which can be used to determine if two terms are equal.

$$\frac{A \equiv^\theta B \quad p\theta \triangleq q\theta}{p, A \equiv^\theta q, B} \equiv p \qquad \frac{A \equiv^\theta B \quad !p\theta \triangleq !q\theta}{!p, A \equiv^\theta !p, B} \equiv !p \qquad \frac{A \equiv^\theta B}{\mathbf{1}, A \equiv^\theta B} \equiv \mathbf{1}\ L$$

$$\frac{A \equiv^\theta B}{A \equiv^\theta \mathbf{1}, B} \equiv \mathbf{1}\ R \qquad \frac{}{\cdot \equiv^\theta \cdot} \equiv \cdot \qquad \frac{A, B, C \equiv^\theta D}{A \otimes B, C \equiv^\theta D} \equiv \otimes\ L \qquad \frac{A \equiv^\theta B, C, D}{A \equiv^\theta B \otimes C, D} \equiv \otimes\ R$$

Theorem 4.3.1 (Match equivalence)
If two multi-sets are equivalent, $A_1, \ldots, A_n \equiv^\theta B_1, \ldots, B_m$, and we can match $A_1 \otimes \cdots \otimes A_n$ in HLD such that $\mathsf{m}^\Gamma\ \Delta \to (A_1 \otimes \cdots \otimes A_n)\theta$ then $\mathsf{m}^\Gamma\ \Delta \to (B_1 \otimes \cdots \otimes B_m)\theta$ is also true.

Proof. By straightforward induction on the first assumption. □

Definition 4.3.1 (Split contexts)
$split(\Omega)$ is defined as $split(\Omega) = times(flatten(\Omega))$, where:

[2] We will sometimes use p in place of $p(\widehat{x})$ for brevity.

$$flatten(\cdot) = \cdot \tag{4.5}$$
$$flatten(\mathbf{1}, \Omega) = flatten(\Omega) \tag{4.6}$$
$$flatten(A \otimes B, \Omega) = flatten(A), flatten(B), flatten(\Omega) \tag{4.7}$$
$$flatten(p, \Omega) = p, flatten(\Omega) \tag{4.8}$$
$$flatten(!p, \Omega) = !p, flatten(\Omega) \tag{4.9}$$

And $times(A_1, \ldots, A_n) = A_1 \otimes \cdots \otimes A_n$.

Theorem 4.3.2 (Split equivalence)
$split(\Omega) \equiv^{\theta} \Omega$.

Proof. Induction on the structure of Ω. □

Well-Formed Continuation Frames

We now define the concept of a well-formed frame given initial linear and persistent contexts and a term A that needs to be matched.

Definition 4.3.2 (Well-formed frame)
Consider a triplet $A; \Gamma; \Delta_{\mathcal{I}}$ where A is a term, Γ is a set of persistent resources and $\Delta_{\mathcal{I}}$ a multi-set of linear resources. A frame f is well-formed iff:

1. Linear frame $f = \underset{\mathsf{m}^{\Gamma}\ \Delta' \to \Omega'}{(\Delta, p_1; \Delta''; p; \Omega; \Psi)}$
   (a) $p, \Omega, \Omega' \equiv^{\Psi} A$ (the remaining terms and already matched terms are equivalent to the initial LHS A);
   (b) $\Delta' = \Delta'_1, \ldots, \Delta'_n$ and $\Omega' = \Omega'_1 \otimes \cdots \otimes \Omega'_n$;
   (c) $\Delta, \Delta'', \Delta, p_1 = \Delta_{\mathcal{I}}$ (available facts, candidate facts for p, consumed facts and the linear fact used for p, respectively, are the same as the initial $\Delta_{\mathcal{I}}$);
   (d) $\mathsf{m}^{\Gamma}\ \Delta' \to \Omega'$ is valid.
2. Persistent frame $f = \underset{\mathsf{m}^{\Gamma}\ \Delta' \to \Omega'}{[\Gamma''; \Delta; !p; \Omega; \Psi]}$
   (a) $!p, \Omega, \Omega' \equiv^{\Psi} A$;
   (b) $\Delta' = \Delta'_1, \ldots, \Delta'_n$ and $\Omega' = \Omega'_1 \otimes \cdots \otimes \Omega'_n$;
   (c) $\Delta, \Delta' = \Delta_{\mathcal{I}}$;
   (d) $\Gamma'' \subset \Gamma$ (remaining candidates are a subset of Γ);
   (e) $\mathsf{m}^{\Gamma}\ \Delta' \to \Omega'$ is valid.

Definition 4.3.3 (Well-formed stack)
A continuation stack $\mathcal{C}$ is well-formed iff every frame is well-formed. For two consequent linear frames $f_1 = (\Delta, \Delta_q, q_1, p_1; \Delta_p; p; \Omega_1)$ $_{\mathsf{m}^\Gamma\ \Delta_1' \to \Omega_1'}$ and $f_2 = (\Delta, \Delta_p, q_1; \Delta_q; q; \Omega_2)$ $_{\mathsf{m}^\Gamma\ \Delta_1', p_1 \to \Omega_1' \otimes p}$ where f_2 is on top of f_1, we also know that $\Omega_1 \equiv^\Psi \Omega_2, p$. Identical relationships exist between different pairs of frames.

4.3.4 Match

The matching state for the LLD machine uses the continuation stack to try different combinations of facts until a match is achieved. The state is structured as $\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to \Omega'}_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma; \Delta; \Omega$, where:

$A \multimap B$ rule being matched: A is the rule's LHS and B the RHS;

$\mathcal{R}$ rule continuation to be used if the current rule fails. Contains the original $\Delta_\mathcal{I}$ and the rest of the rules Φ;

$\mathcal{C}$ ordered list of frames representing the continuation stack used for matching A;

Δ multi-set of linear facts still available to complete the matching process;

Ω ordered list of deconstructed RHS terms to match;

Δ' multi-set of linear facts from the original $\Delta_\mathcal{I}$ that were already consumed ($\Delta', \Delta = \Delta_\mathcal{I}$);

Ω' parts of A already matched. They are in the form $P_1 \otimes \cdots \otimes P_n$. The idea is to use term equivalence and the fact that $\Omega, \Omega' \equiv^\Psi A$ to justify $\mathsf{m}^\Gamma\ \Delta' \to A$ when the matching process completes.

Not shown in the matching state is the context Ψ that maps variables to values. At the start of matching, the $\widehat{x}$ variables are set as *undefined*. Matching then uses facts from Δ and Γ to match the terms of the rule's LHS represented as Ω. During the matching process, continuation frames are pushed into $\mathcal{C}$ and if matching fails, we use $\mathcal{C}$ to restore the process using different candidate facts. New facts also update the variables in the Ψ context by assigning them concrete values.

Linear atomic propositions

The first 2 state transitions are used when the head of Ω is a linear atomic proposition $p(\widehat{x})$. In the first transition we find p_1 and Δ'' as facts from the database that match $p(\widehat{x})$'s hidden and partially initialized arguments. Context Δ'' is stored in the second argument of the new continuation frame but is passed along with Δ since the facts have not been consumed yet (just fact p_1).

The second transition deals with the case where there are no candidate facts and thus a different machine state is used for enabling backtracking.

Note that the proposition $p_1, \Delta'' \prec p(\widehat{x})$ indicates that facts Δ'', p_1 satisfy the constraints of $p(\widehat{x})$ while $\Delta \not\prec p(\widehat{x})$ indicates that no fact in Δ satisfies $p(\widehat{x})$. In the first rule, the substitution context Ψ is extended with a new set of variable assignments θ which take into account the new linear proposition.

$$^{\Psi}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta,p_1,\Delta'';p(\widehat{x}),\Omega\ \mapsto\ {}^{\mathsf{extend}(\Psi,\theta)}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta',p_1\to\Omega'\otimes p(\widehat{x}\theta)}_{A\multimap B}\ \mathcal{R};(\Delta,p_1;\Delta'';p(\widehat{x});\Omega;\mathsf{extend}(\Psi,\theta)),\mathcal{C};\Gamma;\Delta,\Delta'';\Omega \quad (p_1,\Delta''\prec p(\widehat{x})\quad \Delta\not\preceq p(\widehat{x}))$$ (match p ok)

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;p(\widehat{x}),\Omega\ \mapsto\ \triangleleft_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma\quad(\Delta\not\preceq p(\widehat{x}))$$ (match p fail)

Persistent atomic propositions

The next 2 state transitions are used when the head of Ω contains a persistent atomic proposition $!p(\widehat{x})$. They are identical to the previous 2 transitions but they deal with the persistent context Γ.

$$^{\Psi}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;!p(\widehat{x}),\Omega\ \mapsto\ {}^{\mathsf{extend}(\Psi,\theta)}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'\otimes!p(\widehat{x}\theta)}_{A\multimap B}\ \mathcal{R};[\Gamma'';\Delta;!p(\widehat{x});\Omega;\mathsf{extend}(\Psi,\theta)],\mathcal{C};\Gamma,p_1,\Gamma'';\Delta;\Omega \quad (!p_1,\Gamma''\prec !p(\widehat{x})\quad \Gamma\not\preceq !p(\widehat{x}))$$ (match !p ok)

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;!p(\widehat{x}),\Omega\ \mapsto\ \triangleleft_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma\quad(\Gamma\not\preceq !p(\widehat{x}))$$ (match !p fail)

Other propositions

Finally, we have the transitions that deconstruct the other LHS terms and a final transition to initiate the RHS derivation.

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;\mathbf{1},\Omega\ \mapsto\ \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;\Omega$$ (match $\mathbf{1}$)

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;X\otimes Y,\Omega\ \mapsto\ \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;X,Y,\Omega$$ (match $\otimes$)

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;\cdot\ \mapsto\ \curvearrowright^{\vdots}_{\Delta'}\ \Gamma;\Delta;B$$ (match end)

4.3.5 Backtracking

The backtracking state of the machine reads the top of the continuation stack $\mathcal{C}$ and restores the matching process with a different candidate fact from the continuation frame. The state is written as $\triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma$, where:

$A \multimap B$ the rule being matched;

$\mathcal{R}$ next available rules if the current rule does not match; the rule continuation;

$\mathcal{C}$ the continuation stack for matching A;

Linear continuation frames

The next two state transitions handle linear continuation frames on the top of the continuation stack. The first transition selects the next candidate fact p_1 from the second argument of the linear frame and updates the frame. Otherwise, if we have no more candidate facts, we pop the continuation frame and backtrack to the remaining continuation stack.

$$\triangleleft_{A \multimap B}\ \mathcal{R}; (\Delta; p_2, \Delta''; p; \Omega), \mathcal{C}; \Gamma \quad \mapsto \quad \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta', p_2 \to \Omega' \otimes p}_{A \multimap B}\ \mathcal{R}; (\Delta, p_2; \Delta''; p; \Omega), \mathcal{C}; \Gamma; \Delta; \Omega \quad \text{(next p)}$$
$$\mathsf{m}^\Gamma\ \Delta' \to \Omega' \qquad\qquad \mathsf{m}^\Gamma\ \Delta' \to \Omega'$$

$$\triangleleft_{A \multimap B}\ \mathcal{R}; (\Delta; \cdot; p; \Omega), \mathcal{C}; \Gamma \quad \mapsto \quad \triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma \qquad \text{(next frame)}$$
$$\mathsf{m}^\Gamma\ \Delta' \to \Omega'$$

Persistent continuation frames

We also have the same two kinds of inference rules for persistent continuation frames.

$$\triangleleft_{A \multimap B}\ \mathcal{R}; [!p_2, \Gamma''; \Delta; !p; \Omega], \mathcal{C}; \Gamma \quad \mapsto \quad \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to \Omega' \otimes !p_2}_{A \multimap B}\ \mathcal{R}; [\Gamma''; \Delta; !p; \Omega], \mathcal{C}; \Gamma; \Delta; \Omega \quad \text{(next !p)}$$
$$\mathsf{m}^\Gamma\ \Delta' \to \Omega' \qquad\qquad \mathsf{m}^\Gamma\ \Delta' \to \Omega'$$

$$\triangleleft_{A \multimap B}\ \mathcal{R}; [\cdot; \Delta; !p; \Omega], \mathcal{C}; \Gamma \quad \mapsto \quad \triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma \qquad \text{(next !frame)}$$
$$\mathsf{m}^\Gamma\ \Delta' \to \Omega'$$

Empty continuation stack

Finally, if the continuation stack is empty, we simply force execution to try the next inference rule in Φ.

$$\triangleleft_{A \multimap B}\ (\Delta; \Phi); \cdot; \Gamma \quad \mapsto \quad \texttt{infer}\ \Delta; \Phi; \Gamma \qquad \text{(rule fail)}$$

4.3.6 Derivation

Once the list of terms Ω of the LHS is exhausted, we derive the rule's RHS. The derivation state simply iterates over B, the rule's RHS, and derives terms into the corresponding new contexts. The state is represented as $\curvearrowright_{\Xi}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \Gamma; \Delta; \Omega$ with the following meaning:

- Γ set of persistent facts;
- Δ multi-set of remaining linear facts;
- Ξ multi-set of linear facts consumed up-to this point;
- $\Gamma_{\mathcal{N}}$ set of persistent facts derived;
- $\Delta_{\mathcal{N}}$ multi-set of linear facts derived;
- Ω remaining terms to derive as an ordered list. We start with B if the original rule is $A \multimap B$.

Atomic propositions

When deriving either p or $!p$ we have the following two inference rules:

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; p, \Omega \;\; \mapsto \;\; \curvearrowright_{\Xi}^{\Gamma_1;\Delta_1,p} \Gamma; \Delta; \Omega \qquad \text{(new p)}$$

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; !p, \Omega \;\; \mapsto \;\; \curvearrowright_{\Xi}^{\Gamma_1,!p;\Delta_1} \Gamma; \Delta; \Omega \qquad \text{(new !p)}$$

RHS deconstruction

The following two inference rules deconstruct the RHS list Ω from terms created using either $\mathbf{1}$ or $\otimes$.

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; \mathbf{1}, \Omega \;\; \mapsto \;\; \curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; \Omega \qquad \text{(new } \mathbf{1}\text{)}$$

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; A \otimes B, \Omega \;\; \mapsto \;\; \curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; A, B, \Omega \qquad \text{(new } \otimes\text{)}$$

Aggregates

We also have a transition for aggregates. The aggregate starts with a set of values $\widehat{V}$ and an accumulator initialized as $\cdot$. The second state initiates the matching process of the LHS A of the aggregate (explained in the next section).

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; \mathcal{R}_{agg}^{(\widehat{V},\cdot;\cdot)}, \Omega \;\; \mapsto \;\; {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta;\Xi} \blacktriangleright_{\mathsf{agg};\cdot}^{\mathsf{m}^{\Gamma}\ \cdot \to \mathbf{1}} \cdot; \cdot; \Gamma; \Delta; A \qquad \text{(new agg)}$$
$$\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))$$

Successful rule

Finally, if the ordered list Ω is exhausted, then the whole execution process is done. Note how the output arguments match the input arguments of the der_{LLD} judgment.

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma;\Delta;\cdot \;\mapsto\; \circlearrowleft \Xi;\Gamma_1;\Delta_1 \qquad \text{(rule finished)}$$

4.3.7 Aggregates

The most intricate part of the derivation process is processing comprehensions and aggregates. For both of them, we need to perform as many derivations as the database allows, therefore we need to deterministically check the contents of the database until no more derivations are possible. The matching process is then similar to the process used for matching a rule's LHS as presented in Section 4.3.4, however we use two continuation stacks, $\mathcal{C}$ and $\mathcal{P}$. In $\mathcal{P}$, we put all the initial persistent frames and in $\mathcal{C}$ we put the first linear frame and then everything else.

In order to reuse the stacks $\mathcal{C}$ and $\mathcal{P}$, we need to update them by removing all the frames in $\mathcal{C}$ pushed after the first linear continuation frame. If we tried to use those frames, we would assumed that the linear facts used by the other frames were still in the database, but that is not true because they have been consumed during the first application of the comprehension. For example, if the LHS is `!a(X)` $\otimes$ `b(X)` $\otimes$ `c(X)` and the continuation stack has three frames (one per fact), we cannot backtrack to the frame of `c(X)` because, at that point, the matching process was assuming that the previous `b(X)` linear fact was still available. Moreover, we also need to remove the consumed linear facts from the frames of `b(X)` and `!a(X)` in order to make the stack fully consistent with the new database. We will see later on how to do that.

Example

As an example, consider the following snippet of code inspired in the PageRank program shown in Fig. F.1:

```
update(A),
!numInbound(A, T)
   -o [sum => V; B, W, Val | !edge(A, B), neighbor-pagerank(A, B, Val),
         V = Val/float(T) -o neighbor-pagerank(A, B, Val) -> sum-ranks(A, V)].
```

Let's assume that the rule above was successfully matched with `A = 1` and `T = 2` and the database contains the following facts: `!edge(1, 2)`, `!edge(1, 3)`, `neighbor-pagerank(1, 2, 0.5)` and `neighbor-pagerank(1, 3, 0.5)`. Figure 4.1 shows how the aggregate is computed using the continuation stack. An initial frame is created for `!edge(1, B)` which includes two edge facts and `!edge(1, 2)` is selected to continue the process. Since `B = 2`, the frame for `neighbor-pagerank(1, 2, Val)` includes only `neighbor-pagerank(1, 2, 0.5)` which completes the first application of the aggregate and the same linear fact is re-derived using the first aggregate's RHS. Computation then proceeds by backtracking to the first linear frame, the frame of `neighbor-pagerank(1, 2, Val)`, but there are no more available candidates, therefore the frame is removed and the next candidate `!edge(1, 3)` of frame `!edge(1, B)` is selected. Here, a frame is created for `B = 3` with `neighbor-pagerank(1, 3, 0.5)` and the second

application of the aggregate is completed. The process backtracks again twice but now there are no more candidates and the second aggregate's RHS derives `sum-ranks(1, 1.0)` because `V = 0.5 + 0.5`, completing the aggregate computation.

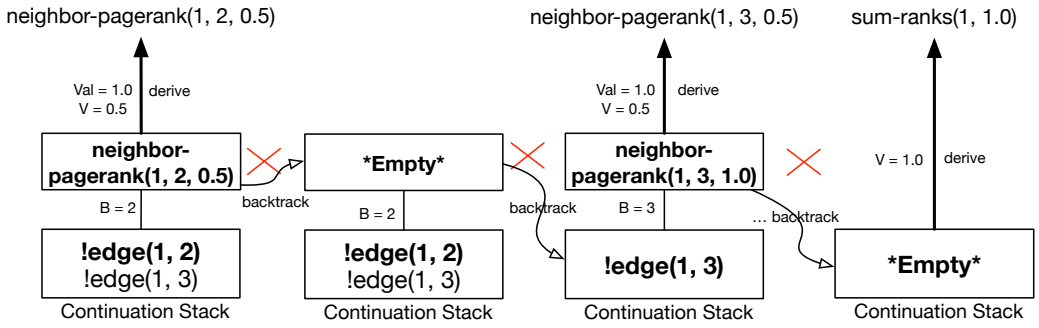


Figure 4.1: *Generating the PageRank aggregate.*

Matching

The matching state for aggregates is ${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta;\Omega$

- $\Omega_{\mathcal{I}}$ ordered list of remaining terms of the rule's RHS to be derived;
- $\Delta_{\mathcal{I}}$ multi-set of linear facts that were still available after matching the rule's LHS and all the previous aggregates. Note that $\Delta, \Xi = \Delta_{\mathcal{I}}$;
- Ξ multi-set of linear facts used during the matching process of the rule's LHS and all the previous aggregates;
- Γ_1 set of persistent facts derived up to this point in the rule's RHS;
- Δ_1 multi-set of linear facts derived up to this point in the rule's RHS;
- Δ' multi-set of linear facts consumed up to this point;
- Ω' terms matched using Δ' up to this point;
- agg aggregate that is being matched;
- Σ the list of aggregated values;
- $\mathcal{C}$ continuation stack that contains both linear and persistent frames. The first frame must be linear;
- $\mathcal{P}$ initial part of the continuation stack with only persistent frames;
- Δ multi-set of linear facts remaining up to this point in the matching process;
- Ω ordered list of terms that need to be matched for the comprehension to be applied.

Since aggregates accumulate values (from specific variables), we retrieved the value from the Ψ context. Remember that Ψ is used for the quantification connectives in the sequent calculus and in LLD is used to store current variable bindings.

Linear atomic propositions

The following two transitions deal with the case when there is a linear atomic propositions in the aggregates' LHS.

$$\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta,p_1,\Delta'';p,\Omega \;\mapsto\; \Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta',p\to\Omega'\otimes p}_{\mathsf{agg};\Sigma} (\Delta,p_1;\Delta'';p;\Omega),\mathcal{C};\mathcal{P};\Gamma;\Delta,\Delta'';\Omega \quad \text{(agg match p ok)}$$

$$\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta;p,\Omega \;\mapsto\; \Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\triangleleft_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma \qquad \text{(agg match p fail)}$$

Persistent atomic propositions

The transitions for dealing with persistent facts are similar to the previous ones.

$$\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \cdot;\mathcal{P};\Gamma,p_1,\Gamma'';\Delta;!p,\Omega \;\mapsto\; \Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'\otimes !p}_{\mathsf{agg};\Sigma} \cdot;[\Gamma'';\Delta;!p;\Omega],\mathcal{P};\Gamma,p_1,\Gamma'';\Delta;\Omega \quad \text{(agg match !p ok } \mathcal{P})$$

$$\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma,p_1,\Gamma'';\Delta;!p,\Omega \;\mapsto\; \Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'\otimes !p}_{\mathsf{agg};\Sigma} [\Gamma'';\Delta;!p;\Omega],\mathcal{C};\mathcal{P};\Gamma,p_1,\Gamma'';\Delta;\Omega \quad \text{(agg match !p ok } \mathcal{C})$$

$$\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta;!p,\Omega \;\mapsto\; \Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\triangleleft_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma \qquad \text{(agg match !p fail)}$$

LHS Deconstruction

$$\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta;X\otimes Y,\Omega \;\mapsto\; \Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta;X,Y,\Omega \quad \text{(agg match } \otimes)$$

$$\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta;\mathbf{1},\Omega \;\mapsto\; \Gamma_{\mathcal{N}};\Delta_{\mathcal{N}} \atop \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta;\Omega \quad \text{(agg match } \mathbf{1})$$

Successful match

When the aggregate's LHS finally matches, we retrieve the term for variable x (the aggregate variable) and add it to the list Σ.

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Delta;\cdot \quad\mapsto\quad \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma \qquad \text{(agg match end)}$$

Continuation stack update

As we said before, to update the continuation stacks, we need remove to all the frames except the first linear frame and remove the consumed linear facts from the remaining frames so that they are still valid for the next application of the aggregate. The judgment that updates the stack has the form $\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma$, where:

- $\Omega_{\mathcal{I}}$ ordered list of remaining terms of the rule's RHS to be derived;
- Δ multi-set of linear facts that were still available after matching the rule's LHS and the aggregate's LHS;
- Ξ multi-set of linear facts used during the matching process of the rule's LHS and all the previous aggregates;
- Δ' multi-set of linear facts consumed by the aggregate's LHS;
- $\Gamma_{\mathcal{N}}$ set of persistent facts derived by the rule's RHS and all the previous aggregates;
- $\Delta_{\mathcal{N}}$ multi-set of linear facts derived by the rule's RHS and all the previous aggregates;
- agg the current aggregate;
- Σ list of accumulated values;
- $\mathcal{C},\mathcal{P}$ continuation stacks for the comprehension;
- Γ set of usable persistent facts.

Remove linear continuation frames

To remove all linear continuation frames except the first one, we simply go through all the frames in the stack $\mathcal{C}$ until only one frame remains. This last frame and stack $\mathcal{P}$ are then updated by removing Δ' from its contexts.

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} \_, f, \mathcal{C};\mathcal{P};\Gamma \quad\mapsto\quad \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} f, \mathcal{C};\mathcal{P};\Gamma \qquad \text{(agg fix rec)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} f;\mathcal{P};\Gamma \quad\mapsto \Omega_{\mathcal{I}};\Delta;\Xi;\Delta' \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};V::\Sigma} f';\mathcal{P}';\Gamma;B\{\Psi(\widehat{x}),V/\widehat{x},\sigma\} \qquad \text{(agg fix end1)}$$
$$\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}^{(\widehat{v},\Sigma')}_{agg} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}^{(\widehat{v},\sigma::\Sigma')}_{agg}))))$$
$$f' = \mathsf{remove}(f,\Delta')$$
$$\mathcal{P}' = \mathsf{remove}(\mathcal{P},\Delta')$$
$$V = \Psi(\sigma)$$

$$\begin{array}{c}{}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta;\Xi;\Delta'} \bowtie_{\mathsf{agg};\Sigma} \cdot;\mathcal{P};\Gamma \;\mapsto\; {}_{\Omega_{\mathcal{I}};\Delta;\Xi,\Delta'} \curvearrowright^{\Gamma_{N1};\Delta_{N1}}_{\mathsf{agg};V::\Sigma} \cdot;\mathcal{P}';\Gamma;B\{\Psi(\widehat{x}),V/\widehat{x},\sigma\} \quad \text{(agg fix end2)}\\ \Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))\\ \mathcal{P}' = \mathtt{remove}(\mathcal{P},\Delta')\\ V = \Psi(\sigma)\end{array}$$

Aggregate backtracking

If the aggregate match fails, we need to backtrack to the next candidate fact. The backtracking state has the form ${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma$, where:

- $\Omega_{\mathcal{I}}$ ordered list of remaining terms of the rule's RHS to be derived;
- $\Delta_{\mathcal{I}}$ multi-set of linear facts that were still available after matching the rule's LHS and the aggregate's LHS;
- Ξ multi-set of linear facts used during the matching process of the rule's LHS and all the previous aggregates;
- $\Gamma_{\mathcal{N}}$ set of persistent facts derived by the rule's RHS and all the previous aggregates;
- $\Delta_{\mathcal{N}}$ multi-set of linear facts derived by the rule's RHS and all the previous aggregates;
- agg the current aggregate;
- Σ list of accumulated values.
- $\mathcal{C},\mathcal{P}$ continuation stacks for the comprehension;
- Γ set of usable persistent facts.

Using the $\mathcal{C}$ stack The following 4 state transitions use the $\mathcal{C}$ stack, the stack where the first continuation frame is linear, to perform backtracking.

$${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} (\Delta; \underset{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}{p_1,\Delta''};p;\Omega),\mathcal{C};\mathcal{P};\Gamma \;\mapsto\; {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta',p_1\to\Omega'\otimes p}_{\mathsf{agg};\Sigma} (\Delta, \underset{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}{p_1};\Delta'';p;\Omega),\mathcal{C};\mathcal{P};\Gamma;\Delta;p,\Omega \quad \text{(agg next p } \mathcal{C})$$

$${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} [p_1, \underset{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}{\Gamma'';\Delta;!p;\Omega}],\mathcal{C};\mathcal{P};\Gamma \;\mapsto\; {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'\otimes !p}_{\mathsf{agg};\Sigma} [\underset{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}{\Gamma'';\Delta;!p;\Omega}],\mathcal{C};\mathcal{P};\Gamma;\Delta;p,\Omega \quad \text{(agg next !p } \mathcal{C})$$

$${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} (\Delta; \underset{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}{\cdot};p;\Omega),\mathcal{C};\mathcal{P};\Gamma \;\mapsto\; {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma \qquad \text{(agg next frame } \mathcal{C})$$

$${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} [\underset{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}{\cdot};\Delta;!p;\Omega],\mathcal{C};\mathcal{P};\Gamma \;\mapsto\; {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma \qquad \text{(agg next !frame } \mathcal{C})$$

Using the $\mathcal{P}$ stack The following 2 state transitions rules use the $\mathcal{P}$ stack instead, the stack where all continuation frames are persistent.

$$\begin{array}{l}{}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\blacktriangleleft_{\mathsf{agg};\Sigma}\ \cdot;[p_1,\underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\Gamma''};\Delta;!p;\Omega],\mathcal{P};\Gamma\ \mapsto \\ \qquad\qquad {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'\otimes!p}_{\mathsf{agg};\Sigma}\ \cdot;[\underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\Gamma''};\Delta;!p;\Omega],\mathcal{P};\Gamma;\Delta;p,\Omega \quad \text{(agg next !p } \mathcal{P})\end{array}$$

$${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\blacktriangleleft_{\mathsf{agg};\Sigma}\ \cdot;[\cdot;\underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\Delta};!p;\Omega],\mathcal{P};\Gamma\ \mapsto\ {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\blacktriangleleft_{\mathsf{agg};\Sigma}\ \cdot;\mathcal{P};\Gamma \qquad \text{(agg next !frame } \mathcal{P})$$

Aggregate done If both the $\mathcal{C}$ and $\mathcal{P}$ stacks are empty, backtracking is impossible and the aggregate is done. The final aggregate's RHS is then derived along with the rest of the rule's RHS.

$$\begin{array}{c}{}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\blacktriangleleft_{\mathsf{agg};\Sigma}\ \cdot;\cdot;\Gamma\ \mapsto\ \curvearrowright^{\Gamma_{N1};\Delta_{N1}}_{\Xi}\ \Gamma;\Delta_{\mathcal{I}};(\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma,\Omega_N \qquad \text{(agg end)}\\ \Pi(\mathsf{agg})=\forall_{\widehat{v},\Sigma'}.(\mathcal{R}^{(\widehat{v},\Sigma')}_{agg}\multimap((\lambda x.Cx)\Sigma'\&(\forall_{\widehat{x},\sigma}.(A\multimap B\otimes\mathcal{R}^{(\widehat{v},\sigma::\Sigma')}_{agg}))))\end{array}$$

Aggregate Derivation

After updating the continuation stacks, the first aggregate's RHS is derived. The derivation state has the form ${}_{\Omega_{\mathcal{I}};\Delta;\Xi}\curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma;\Omega$, where:

- $\Omega_{\mathcal{I}}$ ordered list of remaining terms of the rule's RHS to be derived;
- Δ multi-set of remaining linear facts that can be used for the next aggregate applications.
- Ξ multi-set of linear facts consumed both by the rule's LHS and previous aggregate applications;
- $\Gamma_{\mathcal{N}}$ set of persistent facts derived by the rule's RHS, previous aggregates and current derivation;
- $\Delta_{\mathcal{N}}$ multi-set of linear facts derived by the rule's RHS, previous aggregates and current derivation;
- agg current aggregate symbol;
- Σ accumulated list of values of the aggregate;
- $\mathcal{C},\mathcal{P}$ new continuation stacks;
- Γ set of persistent facts;
- Ω ordered list of terms to derive.

$${}_{\Omega_{\mathcal{I}};\Delta;\Xi}\curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma;p,\Omega\ \mapsto\ {}_{\Omega_{\mathcal{I}};\Delta;\Xi}\curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}},p}_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma;\Omega \qquad \text{(agg new p)}$$

$${}_{\Omega_{\mathcal{I}};\Delta;\Xi}\curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma;!p,\Omega\ \mapsto\ {}_{\Omega_{\mathcal{I}};\Delta;\Xi}\curvearrowright^{\Gamma_{\mathcal{N}},p;\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma;\Omega \qquad \text{(agg new !p)}$$

$$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;X\otimes Y,\Omega \;\mapsto\; \Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright^{\Gamma_{\mathcal{N}},p;\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;X,Y,\Omega \qquad \text{(agg new } \otimes\text{)}$$

$$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\mathbf{1},\Omega \;\mapsto\; \Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright^{\Gamma_{\mathcal{N}},p;\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\Omega \qquad \text{(agg new } \mathbf{1}\text{)}$$

$$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma;\cdot \;\mapsto\; {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta;\Xi}\!\triangleleft_{\mathsf{agg};\Sigma} \mathcal{C};\mathcal{P};\Gamma \qquad \text{(agg next)}$$

This completes the specification of the LLD semantics.

4.3.8 State well-formedness

Given all the state transitions show above, we now define what it means for a given machine state to be well-formed.

Definition 4.3.4 (Well-formed LHS match)
${}^{\Psi}\blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;\Omega$ is well-formed in relation to a triplet $A;\Gamma;\Delta_{\mathcal{I}}$ iff:

- $\Delta,\Delta' = \Delta_{\mathcal{I}}$
- $\Omega',\Omega \equiv^{\Psi} A$;
- $\mathsf{m}^{\Gamma}\ \Delta' \to \Omega$ is valid;
- $\mathcal{C}$ is well-formed in relation to $A;\Gamma;\Delta_{\mathcal{I}}$ and:
  - If $\mathcal{C} = \cdot$
    $\Omega \equiv^{\Psi} A$.
  - If $\mathcal{C} = (\Delta_a, p_1; \Delta_b; p; \Omega_a; \Psi), \mathcal{C}'$:
    $\mathsf{m}^{\Gamma}\ \Delta_c\to\Omega_b$
    - $\Omega' \equiv^{\Psi} \Omega$;
    - $\Delta = \Delta_a, \Delta_b$;
    - $\Delta' = \Delta_c, p_1$.
  - If $\mathcal{C} = [\Gamma''; \Delta_a; !p; \Omega_a; \Psi], \mathcal{C}'$:
    $\mathsf{m}^{\Gamma}\ \Delta_a\to\Omega_b$
    - $\Omega \equiv^{\Psi} \Omega_a$;
    - $\Delta' = \Delta_a$;
    - $\Delta = \Delta_a$.

Definition 4.3.5 (Well-formed backtracking)
$\triangleleft_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma$ is well-formed in relation to a triplet $A;\Gamma;\Delta_{\mathcal{I}}$ iff $\mathcal{C}$ is well-formed in relation to $A;\Gamma;\Delta_{\mathcal{I}}$.

Definition 4.3.6 (Well-formed aggregate match)
The matching state ${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta' \to \Omega'}_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma;\Delta;\Omega$ is well-formed in relation to a triplet $A;\Gamma;\Delta_{\mathcal{I}}$ iff:

- $\mathcal{P}$ is composed solely of persistent frames;
- $\mathcal{C}$ is composed of either linear or persistent frames, but the first frame is linear;
- $\mathsf{m}^{\Gamma}\ \Delta' \to \Omega$ is valid;
- $\Delta, \Delta' = \Delta_{\mathcal{I}}$;
- $\Omega, \Omega' \equiv^{\Psi} A$;
- $\mathcal{C}$ and $\mathcal{P}$ are well-formed in relation to $A;\Gamma;\Delta_{\mathcal{I}}$ and follow the same rules presented before in "Well-formed LHS match" as a stack $\mathcal{C}, \mathcal{P}$.

Definition 4.3.7 (Well-formed aggregate backtracking)
${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \triangleleft_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma$ is well-formed in relation to a triplet $A;\Gamma;\Delta_{\mathcal{I}}$ iff:

- $\mathcal{P}$ is composed solely of persistent frames.
- $\mathcal{C}$ is composed of either linear or persistent frames, but the first frame is linear.
- $\mathcal{C}$ and $\mathcal{P}$ are well-formed in relation to $A;\Gamma;\Delta_{\mathcal{I}}$.

Definition 4.3.8 (Well-formed stack update)
${}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta;\Xi;\Delta'} \bowtie_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma$ is well-formed in relation to a triplet $A;\Gamma;\Delta_{\mathcal{I}}$ iff:

- $\mathcal{P}$ is composed solely of persistent frames.
- $\mathcal{C}$ is composed of either linear or persistent frames, but the first frame is linear.
- $\mathcal{C}$ and $\mathcal{P}$ are well-formed in relation to $A;\Gamma;\Delta_{\mathcal{I}}$.
- $\Delta, \Delta' = \Delta_{\mathcal{I}}$

Definition 4.3.9 (Well-formed aggregate derivation)
$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma}\ \mathcal{C};\mathcal{P};\Gamma;\Omega$ is well-formed in relation to a triplet $A;\Gamma;\Delta_{\mathcal{I}}$ iff:

- $\mathcal{P}$ is composed solely of persistent frames.
- $\mathcal{C}$ is empty or has a single linear frame;
- $\mathcal{C}$ and $\mathcal{P}$ are well-formed in relation to $A;\Gamma;\Delta_{\mathcal{I}}$.

For the theorems that follow, we always assume that the states are well-formed in relation to their main contexts and matching LHS, be it either a rule LHS or an aggregate LHS.

4.4 Soundness Proof

Now that we have presented both the HLD and LLD semantics, we are in position to start building our soundness theorem. The soundness theorem proves that if a rule was successfully derived in the LLD semantics then it can also be derived in the HLD semantics. Since the HLD semantics are so close to linear logic, we prove that our language has a determined, correct, proof search behavior when executing programs. However, the completeness theorem cannot be proven since LLD lacks the non-determinism inherent in HLD.

First and foremost, we need to prove some auxiliary theorems and definitions that will be used during the soundness theorem.

4.4.1 Soundness Of Matching

The soundness theorem will be proven into two main steps. First, we prove that performing a rule match at LLD is sound in relation to HLD and then we prove that the derivation of the rule's RHS is also sound.

In order to prove the soundness of matching, we want to reconstitute a valid match $\mathsf{m}^{\Gamma}\ \Delta \rightarrow A$ in HLD from machine steps in LLD. Our machine specification already includes a built-in $\mathsf{m}^{\Gamma}\ \Delta \rightarrow A$ judgment that can be used to prove soundness immediately. However, we need to prove that every state transition preserves the well-formedness of the machine from the previous definitions.

Theorem 4.4.1 (Rule transitions preserve well-formedness)
Given a rule $A \multimap B$, consider a triplet $T = A; \Gamma; \Delta_N$. If a state s_1 is well-formed in relation to T and $s_1 \mapsto s_2$ then s_2 is also well-formed.

Proof. Case by case analysis.

- match p ok: simple manipulation of multi-set equality and use of equivalence rules.
- match p fail: trivial.
- match !p ok: multi-set manipulation and use of equivalence rules.
- match !p fail: trivial.
- match $\mathbf{1}$: use of term equivalence rules.
- match !: use of term equivalence rules.
- next p: simple multi-set manipulation.
- next frame: trivial.
- next !frame: trivial.

□

Given this result, we now need to prove that from an initial matching state, we end up with a final matching state. The final matching state will include the soundness result since all state transitions preserve well-formedness. However, LLD may fail during matching, therefore the match lemma needs to handle unsuccessful matches. In order to be able to use induction, we

must assume a general matching state that already contains some continuation frames in stack $\mathcal{C}$. The lemma also needs to relate the matching state with the backtracking state since there is a need to backtrack during the matching process. Apart from an unsuccessful match, we deal with two situations during a successful match: (1) we succeed without needing to backtrack to a frame in stack $\mathcal{C}$ or (2) we need to backtrack to a frame in $\mathcal{C}$. The complete lemma is stated and proven below.

Lemma 4.4.1 (LHS match result)
Given a rule $A \multimap H$, consider a triplet $T = A; \Gamma; \Delta_N$ and a context $\Delta_N = \Delta_1, \Delta_2, \Xi$. If $s_1 = {}^{\Psi}\blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta' \to \Omega' \Psi}_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma; \Delta_1, \Delta_2; \Omega$ is well-formed in relation to T and $s_1 \quad \mapsto^* \quad s_2$ then:

- Match succeeds with no backtracking to frames of stack $\mathcal{C}$:
  - $s_2 = {}^{\mathsf{extend}(\Psi, \Psi 2)}\blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta', \Delta_2 \to \Omega' \Psi \otimes split(\Omega) \Psi_2}_{A \multimap B}\ \mathcal{R}; \mathcal{C}'', \mathcal{C}; \Gamma; \Delta_1; \cdot$
- Match fails:
  - $s_2 = \triangleleft_{A \multimap B}\ \mathcal{R}; \cdot; \Gamma$
- Match succeeds with backtracking to a linear frame:
  - $s_2 = {}^{\mathsf{extend}(\Psi_f, \Psi_2)}\blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f, p_2, \Delta''_f \to \Omega_f \Psi_f \otimes (p \otimes split(\Omega'_f)) \Psi_2}_{A \multimap B}\ \mathcal{R}; \mathcal{C}'''', f', \mathcal{C}''; \Gamma; \Delta_c; \cdot$
  - $\mathcal{C} = \mathcal{C}', f, \mathcal{C}''$
  - $f = (\Delta_a; \Delta_{b_1}, p_2, \Delta_{b_2}; p; \Omega_f; \Psi_f)$ turns into $f' = (\Delta_a, \Delta_{b_1}, p_2; \Delta_{b_2}; p; \Omega_f; \Psi_f)$ $\mathsf{m}^{\Gamma}\ \Delta'_f \to \Omega'_f$ $\mathsf{m}^{\Gamma}\ \Delta'_f \to \Omega'_f$
- Match succeeds with backtracking to a persistent frame:
  - $s_2 = {}^{\mathsf{extend}(\Psi_f, \Psi_2)}\blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f, \Delta_{c_2} \to \Omega_f \Psi_f \otimes (!p \otimes split(\Omega'_f)) \Psi_2}_{A \multimap B}\ \mathcal{R}; \mathcal{C}'''', f', \mathcal{C}''; \Gamma; \Delta_{c_1}; \cdot$
  - $\mathcal{C} = \mathcal{C}', f, \mathcal{C}''$
  - $f = [\Gamma_1, p_2, \Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f; \Psi_f]$ turns into $f' = [\Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f; \Psi_f]$ $\mathsf{m}^{\Gamma}\ \Delta'_f \to \Omega'_f$ $\mathsf{m}^{\Gamma}\ \Delta'_f \to \Omega'_f$

If $s_1 = \triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma$ is well-formed in relation to T and $s_1 \quad \mapsto^* \quad s_2$ then either:

- Match fails:
  - $s_2 = \triangleleft_{A \multimap B}\ \mathcal{R}; \cdot; \Gamma$
- Match succeeds with backtracking to a linear frame:
  - $s_2 = {}^{\mathsf{extend}(\Psi_f, \Psi_2)}\blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f, p_2, \Delta''_f \to \Omega_f \Psi_f \otimes (p \otimes split(\Omega'_f)) \Psi_2}_{A \multimap B}\ \mathcal{R}; \mathcal{C}'''', f', \mathcal{C}''; \Gamma; \Delta_c; \cdot$
  - $\mathcal{C} = \mathcal{C}', f, \mathcal{C}''$
  - $f = (\Delta_a; \Delta_{b_1}, p_2, \Delta_{b_2}; p; \Omega_f; \Psi_f)$ turns into $f' = (\Delta_a, \Delta_{b_1}, p_2; \Delta_{b_2}; p; \Omega_f; \Psi_f)$ $\mathsf{m}^{\Gamma}\ \Delta'_f \to \Omega'_f$ $\mathsf{m}^{\Gamma}\ \Delta'_f \to \Omega'_f$
- Match succeeds with backtracking to a persistent frame:
  - $s_2 = {}^{\mathsf{extend}(\Psi_f, \Psi_2)}\blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f, \Delta_{c_2} \to \Omega_f \Psi_f \otimes (!p \otimes split(\Omega'_f)) \Psi_2}_{A \multimap B}\ \mathcal{R}; \mathcal{C}'''', f', \mathcal{C}''; \Gamma; \Delta_{c_1}; \cdot$
  - $\mathcal{C} = \mathcal{C}', f, \mathcal{C}''$
  - $f = [\Gamma_1, p_2, \Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f; \Psi_f]$ turns into $f' = [\Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f; \Psi_f]$ $\mathsf{m}^{\Gamma}\ \Delta'_f \to \Omega'_f$ $\mathsf{m}^{\Gamma}\ \Delta'_f \to \Omega'_f$

Proof. Proof by lexicographic induction on the state transitions. First on the size of Ω and then on the size of the second argument of the linear frame or on the first argument of the linear frame and then on the size of the stack $\mathcal{C}$. Sub-cases:

- match p ok
  Induction on the state with a new frame (Ω is smaller). Trivial if match fails, otherwise it succeeds by adding new frames (including the new frame) or by backtracking.
- match p fail
  State gets smaller (see next).
- match !p ok
  Use the strategy used for match p ok.
- match !p fail: state gets smaller.
- match $\mathbf{1}$: trivial because $split$ removes $\mathbf{1}$.
- match !: same.
- next p
  Frame gets smaller so we can use the induction hypothesis:
  - Match fails: trivial.
  - Match succeeds with no backtracking: the frame that was updated is the successful frame to backtrack to.
  - Match succeeds with backtracking: the frame $f \in \mathcal{C}$ is the frame we need.
- next frame: stack gets smaller.
- next !frame: stack gets smaller (see above).

□

For the induction hypothesis to be applicable in Lemma 4.4.1 there must be a relation between the machine states. We can define a lexicographic ordering $s_1 \prec s_2$, meaning that s_1 has a smaller number of remaining steps than state s_2. The specific ordering is as follows:

1. $\triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma \prec \triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}', \mathcal{C}; \Gamma$
   The continuation must use the top of the stack $\mathcal{C}'$ before using $\mathcal{C}$;
2. $\triangleleft_{A \multimap B}\ \mathcal{R}; (\Delta, \underset{\mathsf{m}^\Gamma\ \Delta_f \to \Omega_f}{\Delta_1; \Delta_2; p; \Omega}), \mathcal{C}; \Gamma \prec \triangleleft_{A \multimap B}\ \mathcal{R}; (\Delta; \underset{\mathsf{m}^\Gamma\ \Delta_f \to \Omega_f}{\Delta_1, \Delta_2; p; \Omega}), \mathcal{C}; \Gamma$
   A continuation frame with more candidates has more steps to do than a frame with less candidates;
3. $\triangleleft_{A \multimap B}\ \mathcal{R}; [\underset{\mathsf{m}^\Gamma\ \Delta_f \to \Omega_f}{\Gamma_1; \Delta; !p; \Omega}]; \Gamma \prec \triangleleft_{A \multimap B}\ \mathcal{R}; [\underset{\mathsf{m}^\Gamma\ \Delta_f \to \Omega_f}{\Gamma_1, \Gamma_2; \Delta; !p; \Omega}]; \Gamma$
   Same as the previous one;
4. $\triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma \prec \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to \Omega'}_{A \multimap B}\ \mathcal{R}; \mathcal{C}', \mathcal{C}; \Gamma; \Delta; \Omega$
5. $\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to \Omega'}_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma; \Delta; \Omega \prec \triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}', \mathcal{C}; \Gamma$
6. $\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'_1 \to \Omega'_1}_{A \multimap B}\ \mathcal{R}; \mathcal{C}', \mathcal{C}; \Gamma; \Delta_1; \Omega_1 \prec \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'_2 \to \Omega'_2}_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma; \Delta_2; \Omega_2$
   Adding continuation frames to the stack makes the proof smaller as long as Ω is also smaller;

7. $\blacktriangleright_{A \multimap B}^{\mathsf{m}^\Gamma\ \Delta_1' \to \Omega_1'}\ \mathcal{R}; \mathcal{C}', (\Delta_f, \underset{\mathsf{m}^\Gamma\ \Delta_f' \to \Omega_f'}{\Delta_{f_1}; \Delta_{f_2}; p; \Omega_f}), \mathcal{C}; \Gamma; \Delta_1; \Omega_1$
$\prec \blacktriangleright_{A \multimap B}^{\mathsf{m}^\Gamma\ \Delta_2' \to \Omega_2'}\ \mathcal{R}; \mathcal{C}'', (\Delta_f; \underset{\mathsf{m}^\Gamma\ \Delta_f' \to \Omega_f'}{\Delta_{f_1}, \Delta_{f_2}; p; \Omega_f}), \mathcal{C}; \Gamma; \Delta_2; \Omega_2$

8. $\blacktriangleright_{A \multimap B}^{\mathsf{m}^\Gamma\ \Delta_1' \to \Omega_1'}\ \mathcal{R}; \mathcal{C}', [\Gamma_1; \underset{\mathsf{m}^\Gamma\ \Delta_f \to \Omega_f}{\Delta_f; !p; \Omega_f}], \mathcal{C}; \Gamma; \Delta_1; \Omega_1$
$\prec \blacktriangleright_{A \multimap B}^{\mathsf{m}^\Gamma\ \Delta_2' \to \Omega_2'}\ \mathcal{R}; \mathcal{C}'', [\Gamma_1, \Gamma_2; \underset{\mathsf{m}^\Gamma\ \Delta_f' \to \Omega_f'}{\Delta_f; !p; \Omega_f}], \mathcal{C}; \Gamma; \Delta_2; \Omega_2$

4.4.2 Soundness Of Derivation

Proving that the derivation of the rule's RHS is sound is trivial except for comprehensions and aggregates. LLD deterministically computes the number of available aggregates to apply while HLD "guesses" the number of required derivations. In the next two sections, we show how to prove the soundness of aggregates. The strategy for proving both is identical due to their inherent similarities.

4.4.3 Aggregate Soundness

The proof that deriving an aggregate in LLD is sound in relation to HLD is built from four results: (1) proving that matching the aggregate's LHS is sound in relation to HLD; (2) proving that updating the continuation stacks makes them suitable for use in the next aggregate applications; (3) proving that deriving the aggregate's RHS is sound in relation to HLD; (4) proving that we can apply as many aggregates as the database allows.

Lemma 4.4.2 (Aggregate LHS match)
Consider an aggregate agg, where $\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))$, a triplet $T = A; \Gamma; \Delta_{\mathcal{I}}$ and a context $\Delta_{\mathcal{I}} = \Delta_1, \Delta_2, \Delta'$.
If $s_1 = \underset{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \blacktriangleright_{\mathsf{agg};\Sigma}^{\mathsf{m}^\Gamma\ \Delta' \to \Omega'}\ \mathcal{C}; \mathcal{P}; \Gamma; \Delta_1, \Delta_2; \Omega$ is well-formed in relation to T and $s_1 \mapsto^* s_2$ then either:

- Match succeeds with no backtracking to frames of stack $\mathcal{C}$ or $\mathcal{P}$ ($\mathcal{C} \neq \cdot$):
  - $s_2 = \underset{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \blacktriangleright_{\mathsf{agg};\Sigma}^{\mathsf{m}^\Gamma\ \Delta', \Delta_2 \to \Omega' \otimes split(\Omega)}\ \mathcal{C}', \mathcal{C}; \mathcal{P}', \mathcal{P}; \Gamma; \Delta_1; \cdot$
- Match fails:
  - $s_2 = \curvearrowright_{\Xi}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}\ \Gamma; \Delta_{\mathcal{I}}; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_N$
- Match succeeds with backtracking to a linear continuation frame in stack $\mathcal{C}$ ($\mathcal{C} \neq \cdot$):
  - $s_2 = \underset{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \blacktriangleright_{\mathsf{agg};\Sigma}^{\mathsf{m}^\Gamma\ \Delta_f', p_2, \Delta'' \to \Omega_f \otimes p \otimes split(\Omega_f')}\ \mathcal{C}'''', f', \mathcal{C}''; \mathcal{P}; \Gamma; \Delta_c; \cdot$
  - $\mathcal{C} = \mathcal{C}', f, \mathcal{C}''$
  - $f = (\Delta_a; \underset{\mathsf{m}^\Gamma\ \Delta_f' \to \Omega_f'}{\Delta_{b_1}, p_2, \Delta_{b_2}; p; \Omega_f})$ turns into $f' = (\Delta_a, \underset{\mathsf{m}^\Gamma\ \Delta_f' \to \Omega_f'}{\Delta_{b_1}, p_2; \Delta_{b_2}; p; \Omega_f})$

- Match succeeds with backtracking to a persistent continuation frame in stack $\mathcal{C}$ ($\mathcal{C} \neq \cdot$):
  - $s_2 = \overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f,\Delta_{c_2},\to\Omega_f\otimes!p\otimes split(\Omega'_f)}_{\mathsf{agg};\Sigma} \mathcal{C}''', f', \mathcal{C}''; \mathcal{P}; \Gamma; \Delta_{c_1}; \cdot$
  - $\mathcal{C} = \mathcal{C}', f, \mathcal{C}''$
  - $f = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{[\Gamma_1, p_2, \Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f]}$ turns into $f' = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{[\Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f]}$
- Match succeeds with backtracking to a persistent continuation frame in stack $\mathcal{P}$ ($\mathcal{C} = \cdot$):
  - $s_2 = \overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f,\Delta_{c_2},\to\Omega_f\otimes!p\otimes split(\Omega'_f)}_{\mathsf{agg};\Sigma} \mathcal{C}'; \mathcal{P}''', f', \mathcal{P}''; \Gamma; \Delta_{c_1}; \cdot$
  - $\mathcal{P} = \mathcal{P}', f, \mathcal{P}''$
  - $f = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{[\Gamma_1, p_2, \Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f]}$ turns into $f' = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{[\Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f]}$

If $\overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma$ is well-formed in relation to T then either:

- Match fails:
  - $s_2 = \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Xi} \Gamma; \Delta_{\mathcal{I}}; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_N$
- Match succeeds with backtracking to a linear continuation frame in stack $\mathcal{C}$ ($\mathcal{C} \neq \cdot$):
  - $s_2 = \overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f,p_2,\Delta''\to\Omega_f\otimes p\otimes split(\Omega'_f)}_{\mathsf{agg};\Sigma} \mathcal{C}''', f', \mathcal{C}''; \mathcal{P}; \Gamma; \Delta_c; \cdot$
  - $\mathcal{C} = \mathcal{C}', f, \mathcal{C}''$
  - $f = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{(\Delta_a; \Delta_{b_1}, p_2, \Delta_{b_2}; p; \Omega_f)}$ turns into $f' = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{(\Delta_a, \Delta_{b_1}, p_2; \Delta_{b_2}; p; \Omega_f)}$
- Match succeeds with backtracking to a persistent continuation frame in stack $\mathcal{C}$ ($\mathcal{C} \neq \cdot$):
  - $s_2 = \overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f,\Delta_{c_2},\to\Omega_f\otimes!p\otimes split(\Omega'_f)}_{\mathsf{agg};\Sigma} \mathcal{C}''', f', \mathcal{C}''; \mathcal{P}; \Gamma; \Delta_{c_1}; \cdot$
  - $\mathcal{C} = \mathcal{C}', f, \mathcal{C}''$
  - $f = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{[\Gamma_1, p_2, \Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f]}$ turns into $f' = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{[\Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f]}$
- Match succeeds with backtracking to a persistent continuation frame in stack $\mathcal{P}$ ($\mathcal{C} = \cdot$):
  - $s_2 = \overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f,\Delta_{c_2},\to\Omega_f\otimes!p\otimes split(\Omega'_f)}_{\mathsf{agg};\Sigma} \mathcal{C}'; \mathcal{P}''', f', \mathcal{P}''; \Gamma; \Delta_{c_1}; \cdot$
  - $\mathcal{P} = \mathcal{P}', f, \mathcal{P}''$
  - $f = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{[\Gamma_1, p_2, \Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f]}$ turns into $f' = \underset{\mathsf{m}^{\Gamma}\ \Delta'_f\to\Omega'_f}{[\Gamma_2; \Delta_{c_1}, \Delta_{c_2}; !p; \Omega_f]}$

We prove this particular lemma by reapplying the strategy used in Lemma 4.4.1[3]. Next, we need to prove that, when matching succeeds, the continuation stack is corrected for the next application of the aggregate. Note that the aggregate value is appended to Σ after the stack is corrected.

Theorem 4.4.2 (From update to derivation)
Consider an aggregate agg, where $\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap$

[3]We have omitted the Ψ context for brevity.

$B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))$, a triplet $T = A; \Gamma; \Delta_{\mathcal{I}}$ and that $\Delta_{\mathcal{I}} = \Delta, \Delta'$. A well-formed stack update $\underset{\Omega_{\mathcal{I}};\Delta;\Xi;\Delta'}{\overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\bowtie}}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma$ implies ${}_{\Omega_{\mathcal{I}};\Delta;\Xi,\Delta'} \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};V::\Sigma} f'; \mathcal{P}'; \Gamma; B\{\Psi(\widehat{x}), V/\widehat{x}, \sigma\}$, where:

- If $\mathcal{C} = \cdot$ then $\mathcal{C}' = \cdot$
- If $\mathcal{C} = \mathcal{C}'', (\underset{\mathsf{m}^{\Gamma}\ \cdot\to\Omega'}{\Delta_a; \Delta_b; p; \Omega})$ then $\mathcal{C}' = (\underset{\mathsf{m}^{\Gamma}\ \cdot\to\Omega'}{\Delta_a - \Delta'; \Delta_b - \Delta'; p; \Omega})$
- $\mathcal{P}'$ is the transformation of stack $\mathcal{P}$, where for every frame $f \in \mathcal{P}$ of the form $\underset{\mathsf{m}^{\Gamma}\ \cdot\to\Omega'}{[\Gamma'; \Delta_{\mathcal{I}}; !p; \Omega]}$ will turn into $f' = \underset{\mathsf{m}^{\Gamma}\ \cdot\to\Omega'}{[\Gamma'; \Delta_{\mathcal{I}} - \Delta'; !p; \Omega]}$

Proof. Use induction on the size of the stack $\mathcal{C}$. □

Corollary 4.4.1 (Match to derivation)
Consider an aggregate agg, where $\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))$, a triplet $T = A; \Gamma; \Delta_{\mathcal{I}}$ and that $\Delta_{\mathcal{I}} = \Delta, \Delta'$.
A well-formed $\underset{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}{\overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\blacktriangleright}}^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta; \cdot$ implies ${}_{\Omega_{\mathcal{I}};\Delta;\Xi,\Delta'} \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};V::\Sigma} f'; \mathcal{P}'; \Gamma; B\{\Psi(\widehat{x}), V/\widehat{x}, \sigma\}$, where:

- If $\mathcal{C} = \cdot$ then $\mathcal{C}' = \cdot$
- If $\mathcal{C} = \mathcal{C}'', (\underset{\mathsf{m}^{\Gamma}\ \cdot\to\Omega'}{\Delta_a; \Delta_b; p; \Omega})$ then $\mathcal{C}' = (\underset{\mathsf{m}^{\Gamma}\ \cdot\to\Omega'}{\Delta_a - \Delta'; \Delta_b - \Delta'; p; \Omega})$
- $\mathcal{P}'$ is the transformation of stack $\mathcal{P}$, where for every frame $f \in \mathcal{P}$ of the form $\underset{\mathsf{m}^{\Gamma}\ \cdot\to\Omega'}{[\Gamma'; \Delta_{\mathcal{I}}; !p; \Omega]}$ will turn into $f' = \underset{\mathsf{m}^{\Gamma}\ \cdot\to\Omega'}{[\Gamma'; \Delta_{\mathcal{I}} - \Delta'; !p; \Omega]}$

Proof. Invert the assumption and then apply Theorem 4.4.2. □

Aggregate derivation We have just seen that after a single aggregate application, we add a value V to the Σ context and that the continuation stacks are now valid. Now, we need to prove that deriving the aggregate's RHS is sound in relation to HLD by using the new stacks.

Theorem 4.4.3 (Aggregate derivation soundness)
If ${}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi_{\mathcal{I}}} \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Omega_1, \ldots, \Omega_n$ then:

- ${}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi_{\mathcal{I}}} \curvearrowright^{\Gamma_{\mathcal{N}},\Gamma_1,\ldots,\Gamma_n;\Delta_{\mathcal{N}},\Delta_1,\ldots,\Delta_n}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \cdot$
- If $\mathsf{der}^{\Gamma;\Pi}\Delta_{\mathcal{I}}; \Xi_{\mathcal{I}}; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n; \Omega_x \to \mathcal{O}$ then $\mathsf{der}^{\Gamma;\Pi}\Delta_{\mathcal{I}}; \Xi_{\mathcal{I}}; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}; \Omega_1, \ldots, \Omega_n, \Omega_x \to \mathcal{O}$

Proof. Straightforward induction on $\Omega_1, \ldots, \Omega_n$. □

Multiple aggregate derivation We now prove that it is possible to apply an aggregate several times in order to a get multiple values (one per application). In turn, we also conclude important results for the soundness of the aggregate computation mechanism.

Theorem 4.4.4 (Multiple aggregate derivation)
Consider an aggregate agg, where $\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))$, and a triplet $T = A; \Gamma; \Delta_{\mathcal{I}}$. Assume that there exists $n \geq 0$ applications of agg where the i_{th} application is related to the following information:
Δ_i : context of derived linear facts;
Γ_i : context of derived persistent facts;
Ξ_i : context of consumed linear facts;
V_i : a value representing the aggregate application;
Ψ_i : context representing new variable bindings for the aggregate.
Since each application consumes Ξ_i then the initial context $\Delta_{\mathcal{I}} = \Delta, \Xi_1, \ldots, \Xi_n$. We now define the two main implications of the theorem.

- Assume that $\Delta_{\mathcal{I}} = \Delta_a, \Delta_b$, $\Delta_b = \Delta'_b, p_1$ and there is a frame $f = \underset{\mathsf{m}^\Gamma \cdot \to \Omega'_f}{(\Delta_a, p_1; \Delta'_b; p; \Omega)}$

  If $s_1 = \underset{\Omega_{\mathcal{I}};\Delta,\Xi_1,\ldots,\Xi_n;\Xi}{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \blacktriangleright^{\mathsf{m}^\Gamma\ p_1 \to \Omega'_f \otimes p}_{\mathsf{agg};\Sigma} f; \mathcal{P}; \Gamma; \Delta_a, \Delta'_b; \Omega$ (well-formed in relation to T) and $s_1 \mapsto^* s_2$ then:
  - n values V_i ($\Sigma' = V_n :: \cdots :: V_1 :: \Sigma$)
  - n aggregate applications are derived:
    $s_2 = \curvearrowright^{\Gamma_{\mathcal{N}},\Gamma_1,\ldots,\Gamma_n;\Delta_{\mathcal{N}},\Delta_1,\ldots,\Delta_n}_{\Xi,\Xi_1,\ldots,\Xi_n} \Gamma; \Delta; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_{\mathcal{I}}$
  - n soundness proofs for the n aggregate matches:
    - $\mathsf{m}^\Gamma\ \Xi_1 \to A$
    - ...
    - $\mathsf{m}^\Gamma\ \Xi_n \to A$
  - n derivation implications for HLD:
    If $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_{i+1}, \ldots, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_i; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_i; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_i; \Omega_x \to \mathcal{O}$ then $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_{i+1}, \ldots, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_i; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{i-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{i-1}; B, \Omega_x \to \mathcal{O}$

- Assume that there is a frame $f = \underset{\mathsf{m}^\Gamma \cdot \to \Omega_f}{[\Gamma'; \Delta_{\mathcal{I}}; !p; \Omega]}$.

  If $s_1 = \underset{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \blacktriangleright^{\mathsf{m}^\Gamma \cdot \to !p \otimes \Omega_f}_{\mathsf{agg};\Sigma} \cdot; f, \mathcal{P}; \Gamma; \Delta_{\mathcal{I}}; \Omega$ (well-formed in relation to T) and $s_1 \mapsto^* s_2$ then:
  - n values V_i ($\Sigma' = V_n :: \cdots :: V_1 :: \Sigma$)
  - n aggregate applications are derived:
    $s_2 = \curvearrowright^{\Gamma_{\mathcal{N}},\Gamma_1,\ldots,\Gamma_n;\Delta_{\mathcal{N}},\Delta_1,\ldots,\Delta_n}_{\Xi,\Xi_1,\ldots,\Xi_n} \Gamma; \Delta; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_{\mathcal{I}}$
  - n soundness proofs for the n aggregate matches:
    - $\mathsf{m}^\Gamma\ \Xi_1 \to A$
    - ...

– $\mathsf{m}^{\Gamma}\ \Xi_n \to A$

- n derivation implications for HLD:
  If $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_{i+1}, \ldots, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_i; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_i; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_i; \Omega_x \to \mathcal{O}$ then
  $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_{i+1}, \ldots, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_i; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{i-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{i-1}; B, \Omega_x \to \mathcal{O}$

Proof. By mutual induction, first on either the size of Δ'_b (second argument of the linear continuation frame) or Γ' (second argument of the persistent frame in $\mathcal{P}$) and then on the size of $\mathcal{C}, \mathcal{P}$. We only show how to prove the first implication since the second implication is proven in a similar way.

$\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}$
$\Omega_{\mathcal{I}};\Delta,\Xi_1,\ldots,\Xi_n;\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ p_1 \to \Omega'_f \otimes p}_{\mathsf{agg};\Sigma} f; \mathcal{P}; \Gamma; \Delta_a, \Delta'_b; \Omega \mapsto^* s_2$ (1) assumption

By applying Lemma 4.4.2 to (1), we get:

- Failure:
  $s_2 = \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Xi} \Gamma; \Delta_{\mathcal{I}}; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_N$ (2) from lemma, thus $n = 0$

- Success with no backtracking to frames of stack $\mathcal{C}$ or $\mathcal{P}$:
  $s_2 = \Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}$
  $\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi \blacktriangleright^{\mathsf{m}^{\Gamma}\ p_1,\Xi'_1 \to \Omega'_f \otimes p \otimes split(\Omega)}_{\mathsf{agg};\Sigma} \mathcal{C}', f; \mathcal{P}; \Gamma; \Delta, \Xi_2, \ldots, \Xi_n; \cdot$ (2) from lemma
  $\Xi_1 = \Xi'_1, p_1$ (3) by definition

  $A \equiv^{\Psi} \Omega'_f \otimes p \otimes split(\Omega)$ (4) by well-formedness
  $\mathsf{m}^{\Gamma}\ \Xi_1 \to A$ (5) from match equivalence theorem and split equivalence on (4)

  $f = (\Delta_a, p_1; \Delta'_b; p; \Omega)$ (6) definition
  $\mathsf{m}^{\Gamma}\ \cdot \to \Omega'_f$

  Now, we apply Corollary 4.4.1 on (2)

  $f' = (\Delta_a, p_1 - \Xi_1; \Delta_b - \Xi_1; p; \Omega)$ (7) from the Corollary
  $\mathsf{m}^{\Gamma}\ \cdot \to \Omega'_f$

  $\Omega_{\mathcal{I}};\Delta,\Xi_2,\ldots,\Xi_n;\Xi,\Xi_1 \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};V_1::\Sigma} f'; \mathcal{P}'; \Gamma; B\{\Psi(\widehat{x}), V/\widehat{x}, \sigma\} \ldots$ (8) from the Corollary

  $\Omega_{\mathcal{I}};\Delta,\Xi_2,\ldots,\Xi_n;\Xi,\Xi_1 \curvearrowright^{\Gamma_{\mathcal{N}},\Gamma_1;\Delta_{\mathcal{N}},\Delta_1}_{\mathsf{agg};\Sigma} f'; \mathcal{P}'; \Gamma; \cdot$
  ... (9) applying Theorem 4.4.3 on (8)
  If $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_2, \ldots, \Xi_n; \Xi, \Xi_1; \Gamma_{\mathcal{N}}, \Gamma_1; \Delta_{\mathcal{N}}, \Delta_1; \Omega_x \to \mathcal{O}$ then
  $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_2, \ldots, \Xi_n; \Xi, \Xi_1; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}; B\{\Psi(\widehat{x}), V/\widehat{x}, \sigma\}, \Omega_x \to \mathcal{O}$ (10) from Theorem 4.4.3 on (8)

  $\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}$
  $\Omega_{\mathcal{I}};\Delta,\Xi_2,\ldots,\Xi_n;\Xi \blacktriangleleft_{\mathsf{agg};V_1::\Sigma} f'; \mathcal{P}'; \Gamma$ (11) next state of (9)

  By executing the next transition on (11) we either fail because there are no more candidates or no more frames and thus $n = 1$ or we have a new match from which we can apply the inductive

hypothesis (smaller number of candidates and/or frames) to get the remaining $n-1$ aggregate values.

- Success with backtracking to the linear continuation frame of stack $\mathcal{C}$:
  $s_2 = {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'_f,p_2,\Xi'_1 \to \Omega'_f \otimes p \otimes split(\Omega)}_{\mathsf{agg};\Sigma} \mathcal{C}', f'; \mathcal{P}; \Gamma; \Delta, \Xi_2, \ldots, \Xi_n; \cdot$ (2) from lemma

  $\Xi_1 = \Delta'_f, p_2, \Xi'_1$ (3) by definition

  $A \equiv^{\Psi} \Omega'_f \otimes p \otimes split(\Omega)$ (4) by well-formedness
  $\mathsf{m}^{\Gamma}\ \Xi_1 \to A$ (5) from match equivalence theorem and split equivalence on (4)

  $f = (\Delta_a, p_1; \underset{\mathsf{m}^{\Gamma}\ \cdot \to \Omega'_f}{\Delta'''_b}, p_2, \Delta''_b; p; \Omega)$ (6) frame to backtrack to
  turns into $f' = (\Delta_a, p_1, \underset{\mathsf{m}^{\Gamma}\ \cdot \to \Omega'_f}{\Delta'''_b}, p_2; \Delta''_b; p; \Omega)$ (4) resulting frame

  Use the same approach as the case with no backtracking.
- Success with backtracking to a persistent continuation frame of stack $\mathcal{P}$:
  Use the same approach as before.

□

This last theorem proves that from a certain initial continuation stack, we are able to apply the aggregate multiple times (until the stack is exhausted). The results of the theorem allows us to rebuild the proof tree in HLD since we get the HLD matching and derivation propositions. What remains to be done is to prove that we do the same for an empty continuation stack.

Lemma 4.4.3 (All aggregates)
Consider an aggregate agg, where $\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))$, and a triplet $T = A; \Gamma; \Delta_{\mathcal{I}}$. Assume that there exists $n \geq 0$ applications of agg where the i_{th} application is related to the following information:
Δ_i : context of derived linear facts;
Γ_i : context of derived persistent facts;
Ξ_i : context of consumed linear facts;
V_i : a value representing the aggregate application;
Ψ_i : context representing new variable bindings for the aggregate.
Since each application consumes Ξ_i then the initial context $\Delta_{\mathcal{I}} = \Delta, \Xi_1, \ldots, \Xi_n$.
If $s_1 = {}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\Omega_{\mathcal{I}};\Delta,\Xi_1,\ldots,\Xi_n;\Xi} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \cdot \to \mathbf{1}}_{\mathsf{agg};\cdot} \cdot; \cdot; \Gamma; \Delta, \Xi_1, \ldots, \Xi_n; A$ (well-formed in relation to T) and $s_1 \mapsto^* s_2$ then:

- n values V_i ($\Sigma' = V_n :: \cdots :: V_1 :: \Sigma$)
- n aggregate applications are derived:
  $s_2 = \curvearrowright^{\Gamma_{\mathcal{N}},\Gamma_1,\ldots,\Gamma_n;\Delta_{\mathcal{N}},\Delta_1,\ldots,\Delta_n}_{\Xi,\Xi_1,\ldots,\Xi_n} \Gamma; \Delta; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_{\mathcal{I}}$
- n soundness proofs for the n aggregate matches:

- $\mathsf{m}^{\Gamma}\ \Xi_1 \to A$
- $\ldots$
- $\mathsf{m}^{\Gamma}\ \Xi_n \to A$

- n derivation implications for HLD:
  If $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_{i+1}, \ldots, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_i; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_i; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_i; \Omega_x \to \mathcal{O}$ then
  $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_{i+1}, \ldots, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_i; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{i-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{i-1}; B, \Omega_x \to \mathcal{O}$

Proof. Apply Lemma 4.4.2 to get two sub-cases:

- Match fails:
  $s_2 = \curvearrowright_{\Xi}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \Gamma; \Delta_{\mathcal{I}}; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_N$ (2) from lemma, thus $n = 0$

- Match succeeds:
  $s_2 = \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Xi_1 \to \mathbf{1} \otimes split(A)}_{\mathsf{agg};\cdot} \mathcal{C}; \mathcal{P}; \Gamma; \Delta, \Xi_2, \ldots, \Xi_n; \cdot$ (2) from lemma
  $A \equiv^{\Psi} \mathbf{1} \otimes split(\Omega)$ (3) by well-formedness
  $\mathsf{m}^{\Gamma}\ \Xi_1 \to A$ (4) from match equivalence theorem and split equivalence on (4)

  Now, we apply Corollary 4.4.1 on (2)

  $\Omega_{\mathcal{I}};\Delta,\Xi_2,\ldots,\Xi_n;\Xi,\Xi_1 \curvearrowright_{\mathsf{agg};V_1::\cdot}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} f'; \mathcal{P}'; \Gamma; B\{\Psi(\widehat{x}), V/\widehat{x}, \sigma\} \ldots$ (5)

  $\Omega_{\mathcal{I}};\Delta,\Xi_2,\ldots,\Xi_n;\Xi,\Xi_1 \curvearrowright_{\mathsf{agg};V_1::\cdot}^{\Gamma_{\mathcal{N}},\Gamma_1;\Delta_{\mathcal{N}},\Delta_1} f'; \mathcal{P}'; \Gamma; \cdot$ (6) applying Theorem 4.4.3 on (5)
  If $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_2, \ldots, \Xi_n; \Xi, \Xi_1; \Gamma_{\mathcal{N}}, \Gamma_1; \Delta_{\mathcal{N}}, \Delta_1; \Omega_x \to \mathcal{O}$ then
  $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_2, \ldots, \Xi_n; \Xi, \Xi_1; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}; B\{\Psi(\widehat{x}), V/\widehat{x}, \sigma\}, \Omega_x \to \mathcal{O}$ (7) from Theorem 4.4.3 on (5)

  $\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta,\Xi_2,\ldots,\Xi_n;\Xi\end{smallmatrix}{}^{\triangleleft}_{\mathsf{agg};V_1::\cdot} f'; \mathcal{P}'; \Gamma$ (8) next state of (6)

  When executing the next transition of state (6) we either get $n = 1$ application of the aggregate or we apply Theorem 4.4.4 to get the remaining $n - 1$ applications.

□

4.4.4 Soundness Of Derivation

We are finally ready to prove that the derivation of the rule's RHS is sound in relation to HLD.

Lemma 4.4.4 (RHS derivation soundness)
If $s_1 = \curvearrowright_{\Xi}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \Gamma; \Delta_{\mathcal{I}}; \Omega$ then $s_1 \mapsto^* \circlearrowleft \Xi, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma^*; \Delta_{\mathcal{N}}, \Delta^*$ and
$\mathsf{der}^{\Gamma;\Pi}\Delta_{\mathcal{I}}; \Xi; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}; A \multimap B, \Omega \to \Xi, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma^*; \Delta_{\mathcal{N}}, \Delta^*$

Proof. Induction on Ω. Most of the sub-cases can be proven using the induction hypothesis or by straightforward rule inference. The sub-case for aggregates is more complicated and is proved below.

Aggregates Apply Lemma 4.4.3 on the assumption to get n applications of the aggregate. Assume that $\Delta_{\mathcal{I}} = \Delta, \Xi_1, \ldots, \Xi_n$, where Ξ_i are the facts consumed and Γ_i, Δ_i the facts derived by the i^{th} application. The lemma proves the following:

- n values $\Sigma = V_n :: \cdots :: V_1 :: \cdot$
- n applications are derived:
  $\curvearrowright_{\Xi,\Xi_1,\ldots,\Xi_n}^{\Gamma_{\mathcal{N}},\Gamma_1,\ldots,\Gamma_n;\Delta_{\mathcal{N}},\Delta_1,\ldots,\Delta_n} \Gamma; \Delta; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_N$ (final state) (1)
- n propositions $\mathsf{m}^{\Gamma}\ \Xi_i \to A$ (2)
- n implications
  If $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_{i+1}, \ldots, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_i; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_i; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_i; \Omega_x \to \mathcal{O}$ then
  $\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_{i+1}, \ldots, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_i; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{i-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{i-1}; B, \Omega_x \to \mathcal{O}$ (3)
- n contexts $\Psi_1, \ldots, \Psi_n$ for variable bindings (4)

From (1) we apply the inductive hypothesis since C is smaller than the original aggregate:
$\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi, \Xi_1, \ldots, \Xi_n; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega$
$\quad \to \Xi, \Xi_1, \ldots, \Xi_n, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n, \Gamma^*; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n, \Delta^*$
Since we are building the proof tree backwards, starting from the final derivation result, we first need to derive $\mathcal{R}_{agg}^{(\widehat{V},\Sigma)}$ by applying rules $\mathsf{der\ agg}_2$:
$\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi, \Xi_1, \ldots, \Xi_n; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n; \mathcal{R}_{agg}^{(\Phi(\widehat{v}),\Sigma)}, \Omega$
$\quad \to \Xi, \Xi_1, \ldots, \Xi_n, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n, \Gamma^*; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n, \Delta^*$

If $n = 0$ then this is all we need, otherwise we need to rebuild the matching and derivation tree of the n^{th} aggregate. Using the n^{th} implication (3) on (5):
$\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_{n-1}; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{n-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{n-1}; B, \mathcal{R}_{agg}^{(\Phi(\widehat{v}),\Sigma)}, \Omega$
$\quad \to \Xi, \Xi_1, \ldots, \Xi_n, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n, \Gamma^*; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n, \Delta^*$

Using $\mathsf{der}\ \multimap$ and the matching proposition (2) on (6), the $A \multimap B$ implication is reconstructed:
$\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_{n-1}; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{n-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{n-1}; A \multimap B, \mathcal{R}_{agg}^{(\Phi(\widehat{v}),\Sigma)}, \Omega$
$\quad \to \Xi, \Xi_1, \ldots, \Xi_n, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n, \Gamma^*; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n, \Delta^*$

Next, we package the implication and the aggregate using $\mathsf{der}\ \otimes$:
$\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_{n-1}; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{n-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{n-1}; (A \multimap B) \otimes \mathcal{R}_{agg}^{(\Phi(\widehat{v}),\Sigma)}, \Omega$
$\quad \to \Xi, \Xi_1, \ldots, \Xi_n, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n, \Gamma^*; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n, \Delta^*$

Now, we apply $\mathsf{der}\ \forall$ to include the whole term and deconstruct Σ into $\sigma :: V_{n-1} :: \cdots :: V_1 :: \cdot$ since V_n is the σ variable. The $\widehat{x}$ terms included in the aggregate's LHS are replaced using information constructed in Ψ_n:
$\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_{n-1}; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{n-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{n-1};$
$\quad \forall_{\widehat{x},\sigma}.((A' \multimap B') \otimes \mathcal{R}_{agg}^{(\Psi(\widehat{v}),\sigma::V_{n-1}::\cdots::V_1::\cdot)}), \Omega$

$$\rightarrow \Xi, \Xi_1, \ldots, \Xi_n, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n, \Gamma^*; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n, \Delta^*$$

This last expression can be folded into $\mathcal{R}_{agg}^{(\Psi(\widehat{v}),V_{n-1}::\cdots::V_1::\cdot)}$:
$\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_n; \Xi, \Xi_1, \ldots, \Xi_{n-1}; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_{n-1}; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_{n-1}; \mathcal{R}_{agg}^{(\Psi(\widehat{v}),V_{n-1}::\cdots::V_1::\cdot)}, \Omega$
$\rightarrow \Xi, \Xi_1, \ldots, \Xi_n, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n, \Gamma^*; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n, \Delta^*$

The last 5 steps are then applied $n-1$ times to get:
$\mathsf{der}^{\Gamma;\Pi}\Delta, \Xi_1, \ldots, \Xi_n; \Xi; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}; \mathcal{R}_{agg}^{(\Psi(\widehat{v}),\cdot)}, \Omega$
$\rightarrow \Xi, \Xi_1, \ldots, \Xi_n, \Xi^*; \Gamma_{\mathcal{N}}, \Gamma_1, \ldots, \Gamma_n, \Gamma^*; \Delta_{\mathcal{N}}, \Delta_1, \ldots, \Delta_n, \Delta^*$

This completes the sub-case for aggregates.

□

4.4.5 Wrapping-up

In order to bring everything together, we need to use the RHS derivation soundness lemma (Lemma 4.4.4) and the LHS match result lemma (Lemma 4.4.1). We first prove that if the LLD machine is able to reach the final state, then there exists one rule where matching was successful. Then, we prove that the application of such rule is sound in relation to HLD.

Theorem 4.4.5 (Soundness)
If $s_1 = \texttt{infer}\ \Delta_{\mathcal{I}}; \Phi; \Gamma$ then either $s_1 \mapsto^* \circlearrowleft \Xi; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}$ and $\exists_{R \in \Phi}.\mathsf{app}_{\Psi}^{\Gamma;\Pi}\ \Delta_{\mathcal{I}}; \Pi \rightarrow \Xi; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}$ or $s_1 \mapsto^* \mathsf{next}_{\Gamma;\Delta_{\mathcal{I}}}$.

Proof. If $\Phi = \cdot$ then the second conclusion applies immediately, otherwise use induction on the size of Φ.

Assume $\Phi = A \multimap B, \Phi'$ and $\mathcal{R} = (\Delta_{\mathcal{I}}; \Phi')$.

$\texttt{infer}\ \Delta_{\mathcal{I}}; \Phi; \Gamma$ (1) first state of the assumption

Applying Lemma 4.4.1 (LHS match result) to the state after two transitions of (1) leads to two sub-cases:

- Match fails:

  $\triangleleft_{A \multimap B}\ \mathcal{R}; \cdot; \Gamma$ (2)

  $\texttt{infer}\ \Delta_{\mathcal{I}}; \Phi'; \Gamma$ (3) state after (2)

  $\exists_{R' \in \Phi'}.\mathsf{app}_{\Psi}^{\Gamma;\Pi}\ \Delta_{\mathcal{I}}; \Pi \rightarrow \Xi; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}.$ (4) i.h. on (3) where $R' \in \Phi'$ if Φ' is not empty

  $\mathsf{next}_{\Gamma;\Delta_{\mathcal{I}}}$ (5) if $\Phi' = \cdot$

- Match succeeds:

$\Psi \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to split(A)\Psi}_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma; \Delta; \cdot$	(1) $\Delta_{\mathcal{I}} = \Delta', \Delta$
$split(A) \equiv^{\Psi} A$	(2) well-formedness of state (1)
$\mathsf{m}^\Gamma\ \Delta \to A\Psi$	(3) match equivalence for (2) and (1)
$\curvearrowright^{\vdots}_{\Delta'} \Gamma; \Delta; B$	(4) state after (1)
$\curvearrowright^{\vdots}_{\Delta'} \Gamma; \Delta; B \ \ \mapsto^* \ \circlearrowleft \Xi; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}$	(5) applying Lemma 4.4.4 to (4)
$\mathsf{der}^{\Gamma;\Pi}\Delta; \Xi; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}; B \to \Delta', \Xi^*; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}$	(5) from the same Lemma
$\mathsf{app}^{\Gamma;\Pi}_{\Psi}\ \Delta, \Delta'; \Pi \to \Delta', \Xi^*; \Gamma_{\mathcal{N}}; \Delta_{\mathcal{N}}$	(6) using app rule on (5) and (3)

□

4.5 Related Work

Linear logic has been used in the past as a basis for logic-based programming languages [Mil85], including bottom-up and top-down programming languages. Lolli, a programming language presented in [HM94], is based on a fragment of intuitionistic linear logic and proves goals by lazily managing the context of linear resources during top-down proof search. LolliMon [LPPW05a] is a concurrent linear logic programming language that integrates both bottom-up and top-down search, where top-down search is done sequentially but bottom-up computations, which are encapsulated inside a monad, can be performed concurrently. Programs start by performing top-down search but this can be suspended in order to perform bottom-up search. This concurrent bottom-up search stops until a fix-point is achieved, after which top-down search is resumed. LolliMon is derived from the concurrent logical framework called CLF [WCPW04, CPWW02, WCPW03].

4.6 Chapter Summary

In this chapter we presented the proof theoretic foundations of LM. First, we introduced the sequent calculus of the linear logic fragment that supports LM. We then presented HLD, the high level dynamic semantics that was created by interpreting the linear logic fragment using focusing and chaining. Next, we designed LLD, an abstract machine for the operational semantics that mimics the execution of rules in our virtual machine minus small details. Finally, we proved that LLD is sound in relation to HLD, thus showing a connection from LLD to the linear logic fragment.

Chapter 5

Local Computation: Data Structures and Compilation

In Chapter 3, we presented different programs that showcased the declarative core of the LM language. In this chapter, we focus on the implementation of LM by describing how local computation is achieved, with a focus on compilation and rule execution. Next, in Chapter 6, we describe the multi threaded implementation that allows LM programs to run on multi core architectures. The coordination aspects of LM, that we introduce later in this dissertation along with the implementation details, are discussed in depth in Chapters 7 and 8.

This chapter is organized as follows. We start a brief overview of the LM's implementation, including its compiler and virtual machine. Next, we present the node data structure, including a detailed description of the data structures used to efficiently manipulate logical facts. We also describe how rules are scheduled locally from newly derived facts stored in the database. Finally, we give a description of the compilation algorithm that turns logical rules into efficient compiled code.

5.1 Implementation Overview

The implementation of the LM language includes a compiler and a virtual machine (VM). Figure 5.1 presents an overview of the compilation process for LM programs. The two main boxes represent the two major components of the system, namely, the compiler and virtual machine.

The virtual machine contains supporting data structures for managing the database of facts and to schedule the execution of rules. The parallel engine described in Chapter 6 is also a major part of the virtual machine and is responsible for managing multi threaded execution by launching threads, managing communication and scheduling concurrent execution.

The compiler transforms LM files into C++ code that uses the virtual machine facilities to implement the program logic. The compiled code implements the inference rules of the program and uses the API of the virtual machine to derive and retract facts and to schedule the execution of rules in the file named `file.cpp`. Since LM does not have traditional input/output facilities, the compiler also creates a separate file, named `file.data`, that contains the program's initial facts and graph structure and is loaded by the runtime when the program is executed.

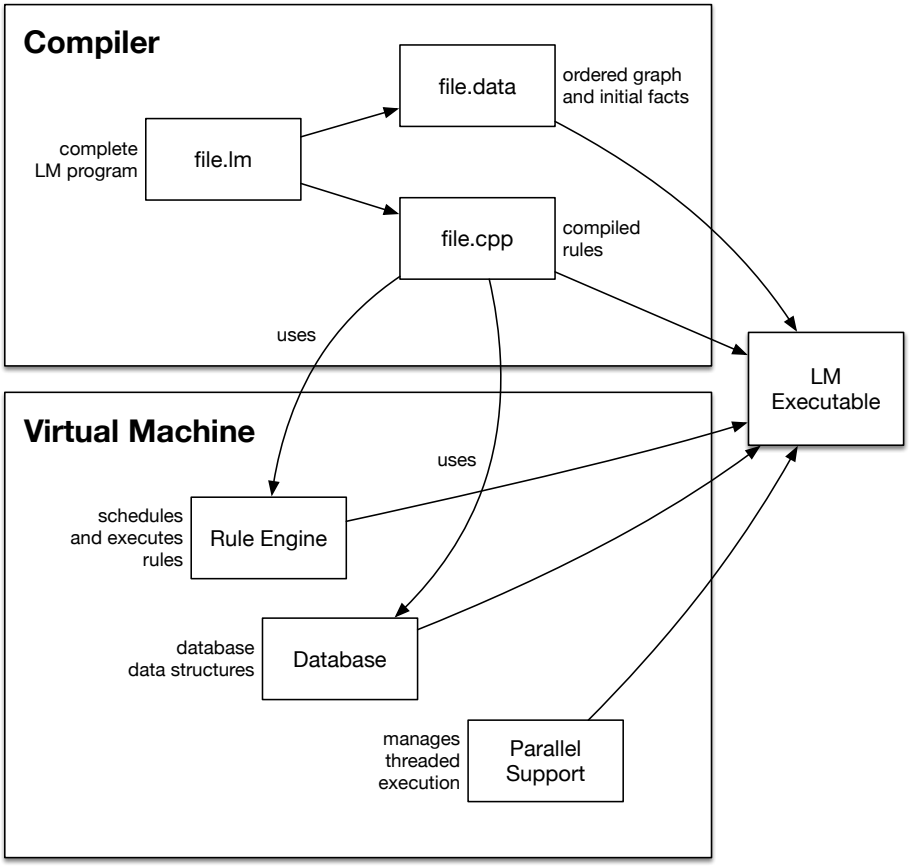


Figure 5.1: *Compilation of a LM program into an executable. The compiler transforms a LM program into a C++ file,* `file.cpp`*, with compiled rules, and into a data file with the graph structure and initial facts,* `file.data`*. The virtual machine which includes code for managing multi threaded execution and the database of facts is then linked with both files to create a stand alone executable that can be run in parallel.*

To complete the compilation process, we use a C++ compiler to compile the virtual machine files and `file.cpp` into object files that are then linked along with `file.data`. At the end, we have a stand alone executable that allows the user to input the number of threads to use, program arguments, facilities to measure run time and also database printing facilities.

Alternatively, the programmer may also decide to compile a more general version of the virtual machine that is able to run byte-code files generated by the compiler. This allows faster development since the programmer only needs to recompile the LM program and not the whole runtime stack. However, LM programs will run slower since the byte-code must be interpreted by the virtual machine. This imposes a significant penalty on programs with many mathematical operations, especially floating point computations.

5.2 Node Data Structure

The main characteristic of LM rules is that they are constrained by the first argument. Rule derivation uses only facts from the same node, therefore there is no need to synchronize with other nodes to derive facts. However, when nodes derive non-local facts (owned by other nodes), then the implementation must synchronize and *send* the facts to the target node. From the point of view of the receiving node, these are called *incoming facts*. Note that this is related to the parallel aspects of the virtual machine and more details are given in the next chapter.

During the lifetime of a program, each node goes through different states as specified by the state machine. Figure 5.2 presents the state machine with the valid state transitions. In the **running** state, the node is applying rules. In the **inactive** state, the node has no new facts to be considered and all candidate rules have been tried. In the **active** state, the node has new facts to be considered but is waiting to be executed by a thread. Finally, in the **stealing** state, the node is currently being stolen by some thread.

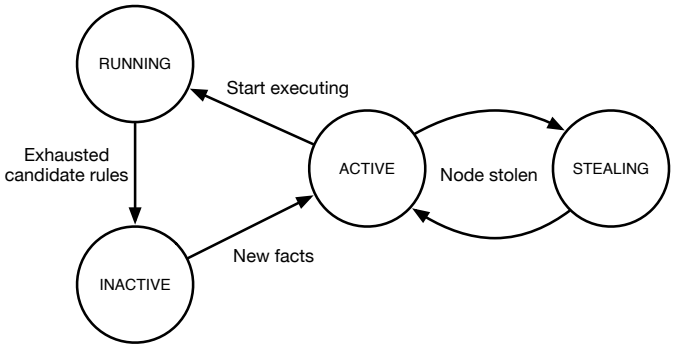


Figure 5.2: *The node state machine.*

As shown in Fig. 5.3, each node contains the following attributes:

- *State*: the node state flag.
- *State Lock*: a lock that protects the following node attributes: *State*, *Owner* and *Incoming Fact Buffer*.
- *DB Lock*: a lock that protects *Linear DB*, *Persistent DB* and *Rule Engine*. The lock is held when the node is applying rules or when incoming facts from other nodes need to be added to the database data structures.
- *Rule Engine*: a data structure that detects which rules are candidates and should be derived. The data structure is fully explained in Section 5.3.
- *Owner*: a pointer to the thread responsible for executing this node.
- *Incoming Fact Buffer*: a data structure that holds incoming facts that could not be added to the database data structures since the node is currently applying rules.
- *Linear DB*: the database of linear facts as a set of structures for storing linear facts for each linear predicate.
- *Persistent DB*: the database of persistent facts.

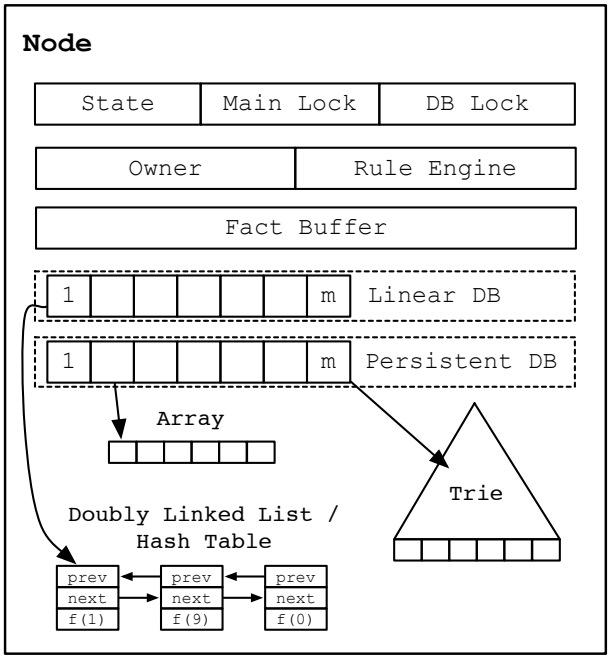


Figure 5.3: *Layout of the node data structure.*

The database of facts must be implemented efficiently because during matching of rules we need to restrict the facts using *join constraints*, which fix arguments of predicates to instantiated values. A database fact is made up of 2 pointers (`prev` and `next`) and a variable number of arguments. The first (node) argument of each fact is not stored in memory since facts are already indexed by node. Additionally, facts are indexed by predicate and use one of the following data structures:

- *Trie Data Structures* are used to store persistent facts. Tries are trees where facts are indexed by common prefix arguments. The `prev` and `next` pointers are used to navigate through all the facts stored in the trie.
- *Array Data Structures* are used to store persistent facts that are used in the LHS of rules that do not require matching (e.g., no join constraints) and exist only as initial facts. Facts stored in this data structure do not have the `prev` and `next` pointers because they are already chained by being part of a contiguous memory area. The compiler performs static analysis of the program's rules in order to choose between trie and array data structures for a particular persistent predicate.
- *Doubly Linked List Data Structures* are used to store linear facts. We use a doubly linked list because adding and removing facts are efficient operations. The `prev` and `next` pointers are used to chain the facts of the linked list.
- *Hash Tree Data Structures* are used to improve lookup when linked lists are too long and when we need to do search filtered by a fixed argument. The virtual machine decides which arguments are best to be indexed (see Section 5.2.1) and then uses a hash table indexed by the appropriate argument. If we need to go through all the facts, we just iterate through all

the facts in the table. Facts are hashed by using the indexed argument and each item in the table corresponds to a value that contains a list of facts with such argument. When a hash bucket has too many values, then the bucket is expanded from a linked list into another hash table, creating a hash tree [Bag01].

Figure 5.4 shows an example for a hash table data structure for a `p(node, int, int)` predicate with 5 linear facts indexed by the third argument. Note that for each bucket, facts are indexed by either a list or another hash table, recursively.

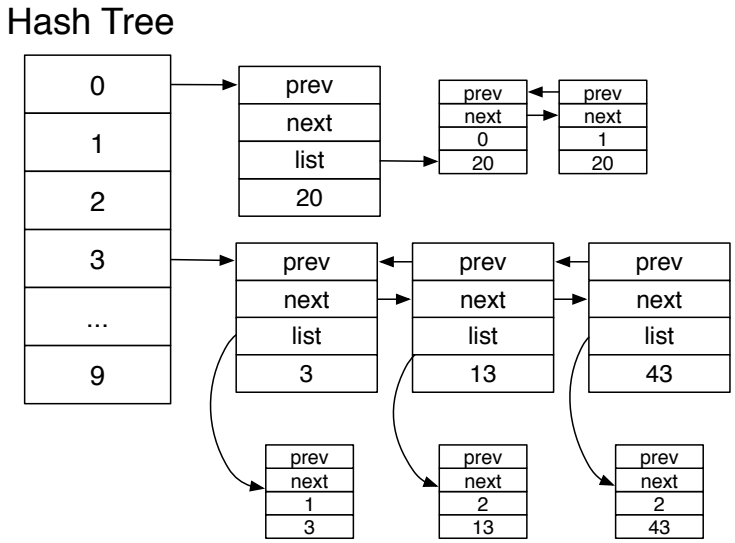


Figure 5.4: *Hash tree data structure for a* `p(node, int, int)` *predicate containing the following facts:* `p(@1, 0, 20)`, `p(@1, 1, 20)`, `p(@1, 1, 3)`, `p(@1, 2, 13)`, *and* `p(@1, 2, 43)`. *Facts are indexed by computing* $arg_2 \mod 10$ *where* arg_2 *is the third argument of the fact. The node argument is not stored.*

5.2.1 Indexing Engine

During rule derivation, facts to be searched and filtered to match the rule constraints. One of the most common constraints is to ensure that some arguments of facts are equal to an arbitrary value. In order to avoid iterating over all the facts in such cases, the VM employs a dynamic mechanism that decides, heuristically, which argument may be optimal to index. When a predicate is indexed, it will use a hash tree data structure instead of doubly linked list for faster lookup. The indexing algorithm is executed along with normal computation and empirically tries to assess the argument of each predicate that more equally spreads the database across the values of the argument.

The indexing algorithm is performed in three main steps and executes only in one thread of the VM, along with normal computation. First, the algorithm gathers lookup statistics by keeping a counter for each predicate's argument. Every time a fact search is performed where arguments are fixed to a value, the counter of such arguments is incremented. This phase is performed during rule execution for the first 100 nodes (or 1% of the nodes of the graph, when there are more than 100000 nodes) that are executed by the thread running the indexing algorithm.

The second step of the algorithm selects the candidate arguments of each predicate. If a search for a predicate required no fixed arguments, then the predicate will be not indexed. If only one argument was fixed, then that argument immediately becomes the indexing argument. Otherwise, the top 2 arguments are selected for the third phase, where *entropy statistics* are collected dynamically.

During the third phase, each candidate argument has an entropy score. Before a node transitions to the **running** state, the facts of the predicate are used in the following formula applied for the two arguments:

$$Entropy(F, A) = -\sum_{v \in values(F,A)} \frac{count(F, A = v)}{total(F)} \log_2 \frac{count(F, A = v)}{total(F)}$$

where A is the target argument, F is the set of linear facts for the target predicate, $values(F, A)$ is set of values of the argument A, $count(F, A = v)$ counts the number of linear facts where argument A is equal to v and $total(F)$ counts the number of linear facts in F. The entropy value is a good metric because it tells us how much information is needed to describe an argument. If more information is needed, then that must be the best argument to index.

For each argument, the $Entropy(A, F)$ value is then multiplied by the number of times it has been used for lookup. The argument with the best score is selected and then a global variable called `indexing_epoch` is updated to **completed**. When nodes are executed and the global `indexing_epoch` variable has changed to **completed**, the node data structures are rebuilt by transforming doubly linked lists into hash tree data structures. Note that the transformation is only done if the linked list has a large number of facts (at least eight).

The VM dynamically augments hash trees if necessary. When a hash tree bucket has too many elements, a child hash tree node is created. On the other hand, if the number of elements in the hash node gets too small, then the hash node is removed and replaced with a doubly linked list.

We have seen good results with this scheme. The overhead of dynamic indexing is negligible since the whole algorithm is performed only once. The programmer can also add indexes statically, if needed, using the directive `index pred/arg`, where `pred` is the argument name and `arg` is the argument number to index.

5.3 Rule Engine

The rule engine decides which rules may need to be executed while respecting the rule priority semantics presented in Section 3.3. The rule engine is composed of 3 data structures:

- *Rule Queue* is a bitmap representing the rules currently scheduled to run. When bit i in the *Rule Queue* is set, it means that rule i is scheduled to be run. A rule i is set to run when all the facts that satisfy the rule's LHS are available. Rules are executed by fetching the least significant bit of the bitmap, unsetting that bit and then executing the corresponding rule. This operation is accomplished by using the *bit scan forward (bsf)* assembly instruction available on x86/x86-64 machines;

```
a, e(1) -o b.  // Rule 1.
a -o c.        // Rule 2.
b -o d.        // Rule 3.
e(0) -o f.     // Rule 4.
c -o e(1).     // Rule 5.

a.
e(0).
```

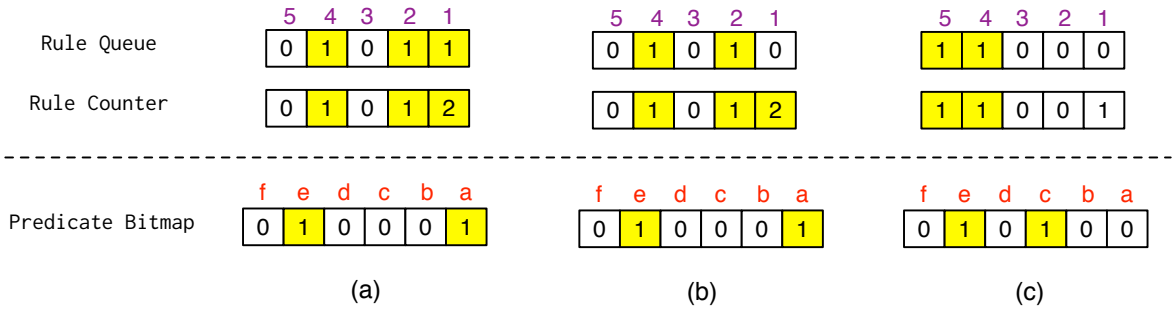


Figure 5.5: *Example program and corresponding rule engine data structures. The initial state is represented in (a), where the rules scheduled to run are 1, 2 and 4. After attempting rule 1, bit 0 is unset from the* Rule Queue, *resulting in (b). Figure (c) is the result of applying rule 2,* `a -o c`, *which marks rule 5 in the* Rule Queue *since the rule is now* available *in the* Rule Counter.

- *Rule Counter* counts the number of predicates that exist in the database and that are needed by a rule. For instance, the rule `a, e(1) -o b` needs predicates `a` and `e`, which means that when the count in the rule counter goes up to two, then the rule is scheduled to run. Likewise, if the counter is decremented to one then the rule is removed from the *Rule Queue*.
- *Predicate Bitmap* is a bitmap representing the existence of facts for a given predicate in the database. When a predicate becomes available then the *Rule Counter* is updated to take into account the existence of new facts.

To better understand how our rule engine works, Fig. 5.5 shows an example program and the corresponding rule engine data structures. When the program starts, the facts for predicates `a` and `e` exist, therefore the *Rule Counter* starts with rules 1, 2 and 4 where the values are 2, 1, and 1, respectively. Since these rules have the required counter to be applied, the *Rule Queue* bitmap starts with the same three rules (Fig. 5.5(a)). In order to pick rules for execution, we take the rule corresponding to the least significant bit from the *Rule Queue* bitmap, initially the first rule `a, e(1) -o b`. However, since we don't have fact `e(1)`, this rule fails and its bit in *Rule Queue* must be set to 0. Figure 5.5(b) shows the rule engine data structures at that point.

The next rule in *Rule Queue* is the second rule `a -o c`. Because this rule succeeds, fact `a` is consumed and fact `c` is derived. We thus update `Predicates Bitmap` accordingly, and decrease the counters for the first and second rules in *Rule Counter* since such rules are no longer applicable (`a` was consumed), and increase the counter for the fifth rule since `c` was derived. Finally, to update the *Rule Queue*, we must schedule the fifth rule since its counter has been

increased to the required number (we have all predicates). Figure 5.5(b) shows the rule engine data structures at that point. In the continuation, the rule engine will schedule the fourth and fifth rules to run.

Note that every node in the program requires the set of data structures presented in Fig. 5.5. We use 64 bit integers to implement the 2 bitmaps and an array of 16 bits integers for the *Rule Counter*.

For persistent facts, we do a small optimization to reduce the number of derivations. We divide the program rules into two sets: *persistent rules* and *non persistent rules*. Persistent rules are rules where only persistent facts are involved. We compile such rules incrementally, i.e., we attempt to fire all rules when a new persistent fact is derived. This is called the *pipelined semi-naive* evaluation and it originated in the P2 system [LCG+06]. This evaluation method avoids excessive re-derivations of the same fact. The order of derivation does not matter for those rules, since only persistent facts are used.

5.4 Compilation

As an intermediate step, our compiler first transforms rules into high level instructions that are then transformed into C++. In Appendix E we present an overview of the high level instructions that can be used as a reference to the operations that are required by the compiler. In this section, we present the main algorithm of the compiler and its key optimizations. To make our presentation more readable, we present our examples using pseudo-code instead of C++ code.

5.4.1 Ordinary Rules

After a rule is compiled, it must respect the *fact constraints* (facts must exist in the database) and the *join constraints* that can be represented by variable constraints and/or boolean expressions. For instance, consider the second rule of the bipartiteness checking program presented in Fig. 3.10:

```
visit(A, P),
colored(A, P)
   -o colored(A, P).
```

The fact constraint include the facts required to trigger the rule, namely `visit(A, P)` and `colored(A, P)`, and the join constraints include the implicit constraint that the second argument of `visit` must be equal to the second argument of `colored` because the variable `P` is used for both arguments. However, rules may also have explicit constraints, such as:

```
visit(A, P1),
colored(A, P2),
P1 <> P2
   -o fail(A).
```

The data structures presented in Section 5.2 support iteration over facts and for linear facts, they also support deletion. Iteration is provided through *iterators*. For linked lists, the iterator points to the linear fact in the list and uses the `next` field to traverse the list. For hash tables, it is possible to iterate through the whole table or iterate through a single bucket. For tries, while

iteration goes through every fact, the operation can be customized with join constraints in order to prune search. To illustrate how rules are compiled, consider again the rule:

```
visit(A, P),
colored(A, P)
   -o colored(A, P).
```

The compiler transforms the rule into two nested *while* loops, as follows:

```
1  colored_list <- linked_list("colored")
2  visit_list <- linked_list("visit")
3  visit <- visit_list.head()
4  while(visit is valid)
5  {
6     colored <- colored_list.head() // Retrieve first element of the list.
7     while(colored is valid)
8     {
9        if(visit.get_int(1) == colored.get_int(1)) { // Equal arguments?
10          // New fact is derived.
11          new_colored <- create_fact("colored") // New fact for predicate colored.
12          new_colored.set_int(1, visit.get_int(1)) // Set arguments.
13
14          colored_list.add(new_colored) // Add new fact to the linked list.
15
16          // Deleting used up facts.
17          colored <- colored_list.delete_and_next(colored)
18          visit <- visit_list.delete_and_next(visit)
19          goto next
20       }
21       colored <- colored.next()
22    }
23    visit <- visit.next()
24 next:
25    continue
26 }
```

The compilation algorithm iterates through the atomic propositions of the rule's LHS and creates nested loops to try all the possible combinations of facts. For this rule, all the pairs of facts `colored` and `visit` must be searched until the implicit constraint is true. First, the `visit` linked list is iterated over by first retrieving the head fact and then using the `next` pointer of each fact. Inside this outer loop, a nested `while` loop is created for predicate `colored`. This inner loop includes a check for the constraint.

If the constraint expression is true then the rule matches and a new `colored` fact is derived and two used linear facts are consumed by deleting them from the linked lists. After the rule is derived, the `visit` and `colored` facts are deleted from the linked list and the pointers adjusted to the next elements of each list. Afterwards, the `goto next` statement is used to jump to the outer loop, which will use the new fact pointers. This forces the procedure to try all the combinations of the rule from the database. Furthermore, we must jump to the first linear loop because we cannot use the next fact from the deepest loop since we may have constraints between the first linear loop and the deepest loop that were validated using deleted facts.

If the implicit constraint failed, another `colored` fact would be checked by assigning `colored` to `colored.next()`. Likewise, if the initial `visit` fact fails for all `colored` facts, then the next

`visit` fact in the list is tried by following the `next` pointer. Note that, for this particular example, we ignore some obvious optimizations such as avoiding the deletion and then re-derivation of `colored` facts.

In order to understand how rule priorities affect compilation, consider a slightly different bipartiteness checking program where the second and third rules are swapped:

```
1  visit(A, P),
2  uncolored(A)
3     -o {B | !edge(A, B) -o visit(B, next(P))},
4        colored(A, P).
5
6  visit(A, P1),
7  colored(A, P2),
8  P1 <> P2
9     -o fail(A).
10
11 visit(A, P),
12 colored(A, P)
13    -o colored(A, P).
14
15 visit(A, P),
16 fail(A)
17    -o fail(A).
```

The procedure for the rule being compiled cannot be the same since the derived `colored` fact is used in the LHS of a higher priority rule, namely, the rule in line 9. Once the `colored` fact is derived, the procedure must return to schedule the higher priority rule, as follows:

```
1  colored_list <- linked_list("colored")
2  visit_list <- linked_list("visit")
3  colored <- colored_list.head()
4  while(colored is valid)
5  {
6     visit <- visit_list.head() // Retrieve first element of the list.
7     while(visit is valid)
8     {
9        if(visit.get_int(1) == colored.get_int(1)) { // Equal arguments?
10          new_colored <- create_fact("colored") // New fact for predicate colored.
11          new_colored.set_int(1, visit.get_int(1)) // Set arguments.
12
13          // New fact is derived.
14          colored_list.add(new_colored) // Add new fact to the linked list.
15
16          // Deleting facts.
17          colored_list.delete_and_next(colored)
18          visit_list.delete_and_next(visit)
19          return
20       }
21       visit <- visit.next()
22    }
23    colored <- colored.next()
24 }
```

This enforces the priority semantics of the language described in Section 3.3. When a rule derives facts that were used as input to a higher priority rule, the initial rule is only fired once because the newly derived facts may trigger the application of a higher priority rule.

Figure 5.6 presents the algorithm for compiling rules into C++ code. First, we split the rule's RHS into atomic propositions and constraints. Fact expressions map directly to iterators while fact constraints map to *if* expressions. A possible compilation strategy is to first compile all the atomic propositions and then compile the constraints. However, this may require unneeded database lookups since some constraints may fail early. Therefore, our compiler introduces constraints as soon as all the variables in the constraint are all included in the already compiled atomic propositions. The order in which fact propositions are selected for compilation does not interfere with the correctness of the compiled code, thus our compiler selects the atomic proposition ($RemoveBestProposition$) that needs the highest number of join constraints, in order to prune search and avoid undesirable database lookups. If two atomic propositions have the same number of new constraints, then the compiler always picks the persistent proposition since persistent facts are not deleted.

Derivation of new facts belonging to the local node requires only that the new facts are added to the local node data structure. Facts that belong to other nodes are sent using an appropriate runtime API.

5.4.2 Persistence Checking

Not all linear facts need to be deleted. For instance, in the compiled rule above, the fact `colored(A, P)` is re-derived in the rule's RHS. Our compiler is able to turn linear loops into persistent loops for linear facts that are consumed and then asserted. The rule is then compiled as follows:

```
1  colored_list <- linked_list("colored")
2  visit_list <- linked_list("visit")
3  colored <- colored_list.head()
4  while(colored is valid)
5  {
6     visit <- visit_list.head() // Retrieve first element of the list.
7     while(visit is valid)
8     {
9        if(visit.get_int(1) == colored.get_int(1)) { // Equal arguments?
10          // Delete visit.
11          visit <- visit_list.delete_and_next(visit) // Get next visit fact.
12          goto next
13       }
14       visit <- visit.next()
15 next:
16       continue
17    }
18    colored <- colored.next()
19 }
```

In this new version of the code, only the `visit` facts are deleted, while the `colored` facts remain untouched. In the bipartiteness checking program, each node has one `colored` fact and

Data: Rule R1, Rules
Result: Compiled Code
$LHSProps \longleftarrow LHSAtomicPropositionsFromRule(R1)$;
$Constraints \longleftarrow ConstraintsFromRule(R1)$;
$Code \longleftarrow CreateFunctionForRule()$;
$FactIterators \longleftarrow []$;
$CompiledProps = []$;
while $LHSProps$ *not empty* **do**
    $Prop \longleftarrow RemoveBestProposition(LHSProps)$;
    $CompiledProps.push(Prop)$;
    $Iterator \longleftarrow Code.InsertIteration(Prop)$;
    $FactIterators.push(Iterator)$;
    `// Select constraints that are covered by CompiledFacts.`
    $NextConstraints \longleftarrow RemoveConstraints(Constraints, CompiledProps)$;
    $Code.InsertConstraints(NextConstraints)$;
end
$RHSProps = RHSAtomicPropositionsFromRule(R1)$;
while $RHSProps$ *not empty* **do**
    $Prop \longleftarrow RemoveFact(RHSProps)$;
    $Code.InsertDerivation(Prop)$;
end
for $Iterator \in FactIterators$ **do**
    if $IsLinear(Iterator)$ **then**
        $Code.InsertRemove(Iterator)$;
    end
end
`// Enforce rule priorities.`
if $FactsDerivedUsedBefore(Rules, R1)$ **then**
    $Code.InsertReturn()$;
else
    $Code.InsertGoto(FirstLinear(FactIterators))$;
end
return $Code$

Figure 5.6: *Compiling LM rules into C++ code.*

this compiled code simply filters out the `visit` facts with the same color. Please note that the `colored` facts are now iterated in the outer loop in order to make the `goto` statement jump to the inner loop. This is now possible because the `colored` fact is not deleted during rule derivation.

5.4.3 Updating Facts

Many inference rules consume and then derive the same predicate but with different arguments. The compiler recognizes those cases and, instead of consuming the fact from its linked list or hash table, it updates the fact in-place. As an example, consider the following rule:

```
new-neighbor-pagerank(A, B, New),
neighbor-pagerank(A, B, Old)
   -o neighbor-pagerank(A, B, New).
```

Assuming that `neighbor-pagerank` is stored in a hash table and indexed by the second argument, the code for the rule above is as follows:

```
1  new_neighbor_pagerank_list <- linked_list("new-neighbor-pagerank")
2  neighbor_pagerank_table <- hash_table("neighbor-pagerank")
3  new_neighbor_pagerank <- new_neighbor_pagerank.head()
4  while(new_neighbor_pagerank is valid)
5  {
6     // Hash table for neighbor-pagerank is indexed by the second argument,
7     // therefore we search for the bucket using the second argument
8     // of new-neighbor-pagerank.
9     neighbor_pagerank <- neighbor_pagerank_table.lookup(new_neighbor_pagerank.get_node(1))
10    while(neighbor_pagerank is valid)
11    {
12       if(new_neighbor_pagerank.get_node(1) == neighbor_pagerank.get_node(1))
13       {
14          // Update fact argument.
15          neighbor_pagerank.set_float(2, new_neighbor_pagerank.get_float(2))
16          new_neighbor_pagerank <- new_neighbor_pagerank_list.
17                   delete_and_next(new_neighbor_pagerank)
18          goto next
19       }
20       neighbor_pagerank <- neighbor_pagerank.next()
21    }
22    new_neighbor_pagerank <- new_neighbor_pagerank.next()
23 next:
24    continue
25 }
```

Note that `neighbor-pagerank` fact is updated using `set_float`. The rule also does not return since it is the highest priority rule. If there was a higher priority rule using `neighbor-pagerank`, then the code would have to return since the act of updating a fact is equivalent to deriving a new one.

5.4.4 Enforcing Linearity

We have already introduced the `goto` statement as a mechanism to avoid reusing deleted linear facts. However, this is not enough in order to enforce linearity of facts. Consider the following

inference rule:

```
add(A, N1),
add(A, N2)
   -o add(A, N1 + N2).
```

Using the standard compilation algorithm, two nested loops are created, one for each add fact. However, notice that there is an implicit constraint (the two add linear facts must be different) when creating the iterator for `add(A, N2)` since this fact cannot be the same as the first one. That would invalidate linearity since a single linear fact would be used to prove two linear facts. This is easily solved by adding a constraint in the inner loop by checking if the second fact is the same as the first one.

```
1  add_list <- linked_list("add")
2  add1 <- add_list.head()
3  while(add1 is valid)
4  {
5     add2 <- add_list.head()
6     while(add2 is valid)
7     {
8        if(add1 != add2)
9        {
10          add1.set_int(1, add1.get_int(1) + add2.get_int(1))
11          add2 <- add_list.delete_and_next(add2)
12          goto next
13       }
14       add2 <- add2.next()
15    }
16    add1 <- add1.next()
17 next:
18    continue
19 }
```

Figure 5.7 presents the steps for executing this rule when the database contains three facts. The fact variables never point to the same fact.

5.4.5 Comprehensions

Consider the comprehension used in the first rule of the bipartiteness checking program in Fig. 3.10:

```
visit(A, P),
uncolored(A)
   -o {B | !edge(A, B) -o visit(B, next(P))},
      colored(A, P).
```

The attentive reader will remember that comprehensions are sub-rules, therefore they should be compiled like normal rules. Comprehensions must also derive all possible combinations. However, the rule itself must return if the comprehension's RHS derives a fact that is used by a higher priority rule. The example rule does not need to return since it has the highest priority and the `visit` facts derived in the comprehension are contained in other nodes. The code for the rule is shown below:

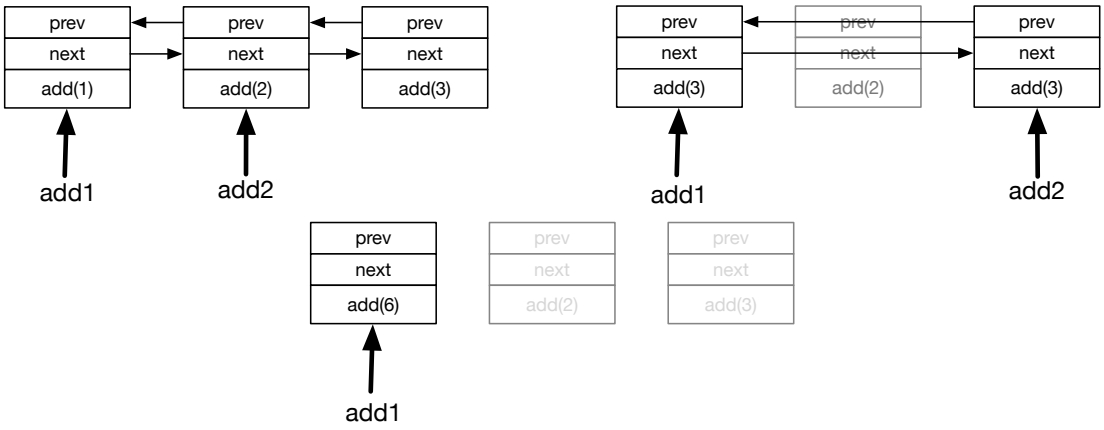


Figure 5.7: *Executing the add rule. First, the two iterators point to the first and second facts and the former is updated while the latter is consumed. The second iterator then moves to the next fact and the first fact is updated again, now to the value* `6`*, the expected result.*

```
colored_list <- linked_list("colored")
visit_list <- linked_list("visit")
uncolored_list <- linked_list("uncolored")
visit <- visit_list.head()
while(visit is valid)
{
   uncolored <- uncolored_list.head()
   while(uncolored is valid)
   {
      // Comprehension code.
      edge_trie <- trie("edge")
      edge <- edge_trie.first()
      while(edge is valid)
      {
         new_visit <- create_fact("visit") // New visit fact.
         new_visit.set_int(1, next(visit.get_int(1)))
         // Send fact to B.
         send_fact(new_visit, edge.get_node(1))
         edge <- edge.next()
      }
      new_colored <- create_fact("colored")
      new_colored.set_int(1, visit.get_int(1))
      colored_list.add(new_colored)
      visit <- visit_list.delete_and_next(visit)
      uncolored <- uncolored_list.delete_and_next(uncolored)
      goto next
   }
   uncolored <- uncolored.next()
next:
   continue
}
```

Special care must be taken when the comprehension's sub-rule uses the same predicates that are derived by the main rule. Rule inference must be atomic in the sense that, after a rule matches, the comprehensions in the rule's RHS can use the facts that were present before the rule was matched. Consider a rule with n comprehensions or aggregates, where each $CLHS_i$ and $CRHS_i$ are the LHS and RHS of the comprehension/aggregate i, respectively, and $RHSP$ represents the atomic propositions found in rule's RHS. The formula used by the compiler to detect conflicts between predicates is the following:

$$\bigcup_{i}^{n}[CLHS_i \cap RHSP] \cup \bigcup_{i}^{n}[CLHS_i \cap \bigcup_{j}^{n}[CRHS_j]]$$

If the result of the formula is not empty, then the compiler disables optimizations for the conflicting predicates and derives the corresponding facts into a temporary data structure before being added into the database data structures. As an example, consider the following rule:

```
update(A),
!edge(A, B)
   -o {P | points(A, B, P) -o update(B)},
      points(A, B, 1).
```

We have $n = 1$ comprehensions, where $CLHS_0 =$ `points(A, B, P)`, $CRHS_0 =$ `update(B)`, and $RHSP =$ `points(A, B, 1)`. There is a conflict because $CLHS_0 \cap RHSP = \{$`points`$\}$, which requires `points(A, B, 1)` to be derived after the comprehension or be stored in a temporary data structure to avoid conflicts with the comprehensions, which could consume the newly derived fact. Fortunately, most rules in LM programs do not show these kinds of conflicts and thus can be fully optimized.

5.4.6 Aggregates

Aggregates are similar to comprehensions. They are also sub-rules but a value is accumulated for each combination of the sub-rule. After all the combinations are derived, a final RHS term is derived with the accumulated value. Consider the following rule that computes a PageRank value by aggregating all neighbors PageRank values:

```
update(A),
pagerank(A, OldRank)
      -o [sum => V; B | neighbor-pagerank(A, B, V) -o neighbor-pagerank(A, B, V)
            -> pagerank(A, damping/P + (1.0 - damping) * V)].
```

The variable `V` is initialized to `0.0` and sums all the PageRank values of the neighbors as seen in the code below. The aggregate value is then used to update the second argument of the initial `pagerank` fact.

```
1  pagerank_list <- linked_list("pagerank")
2  update_list <- linked_list("update")
3  neighbor_pagerank_list <- linked_list("neighbor_pagerank_list")
4  pagerank <- pagerank_list.head()
5  while(pagerank is valid)
6  {
7     update <- update_list.head()
8     while(update is valid)
```

```
9     {
10       V <- 0.0
11       neighbor_pagerank <- neighbor_pagerank_list.head()
12       while(neighbor_pagerank is valid)
13       {
14          V <- V + neighbor_pagerank.get_float(2)
15          neighbor_pagerank <- neighbor_pagerank.next()
16       }
17       // RHS of the aggregate
18       pagerank.set_float(1, damping / P + (1.0 - damping) * V)
19       update <- update_list.delete_and_next(update)
20       goto next
21    }
22    pagerank <- pagerank.next()
23 next:
24    continue
25 }
```

5.5 Chapter Summary

In this chapter, we gave an overview of LM's implementation, including the compiler and the virtual machine. We focused on the sequential aspects of the implementation, namely, the data structures used for manipulating facts and how the compiler turns logical rules into efficient code. In the next chapter, we focus on the parallel aspects of the virtual machine and present an evaluation of the performance and scalability of the implementation.

Chapter 6

Multi Core Implementation

This chapter describes the multi core aspects of LM's virtual machine and provides a detailed experimental evaluation of the performance and scalability of the system. First, we describe how local node computation and parallelism are integrated, with a focus on locking and memory allocation, and then we evaluate the performance, memory usage and scalability of our implementation. The mechanisms related to coordination and their implementation will be described in Chapter 7.

6.1 Parallelism

A key goal of our parallel design is to keep the threads as busy as possible and to reduce the overhead of inter-thread communication. Initially, the VM partitions the application graph of N nodes into T subgraphs (the number of threads) and then each thread works on their own subgraph. During execution, threads can steal nodes of other threads to keep themselves busy. The load balancing aspect of the system is performed by our work scheduler that is based on a simple work stealing algorithm.

Since LM uses an asynchronous and graph-based model of computation where nodes of the graph can compute independently of other nodes, the granularity problem, common in other declarative languages, can be solved by dynamically grouping nodes of the graph into subgraphs that are processed by a single executing thread. This avoids the creation of too many threads of control and allows the dynamic assignment of nodes to threads through the use of work stealing.

In our VM, each VM's thread uses a *Work Queue* that contains **active** nodes, i.e., nodes that have new facts to process and are part of the thread's subgraph. The work queue is implemented as a singly linked list and is embedded in the node data structures. Initially, the work queue is filled with the nodes in the thread's subgraph in order to derive the initial facts. A thread may go through different states during its lifetime. The state is kept track in the *State* flag which may have one of the following values:

- **active**: The thread has a non-empty *Work Queue* or is currently executing rules for a node (which is not in the *Work Queue*).
- **stealing**: The thread has no active nodes in the *Work Queue* and is attempting to steal nodes from other threads.

- **idle**: The thread is trying to synchronize with other threads to terminate the program.
- **terminated**: The thread (and all the other threads) have terminated and the program is finished.

Figure 6.1 presents the valid transitions for the thread state flag. The dashed line from **idle** to **stealing** indicates that the transition is made in a non-deterministic fashion.

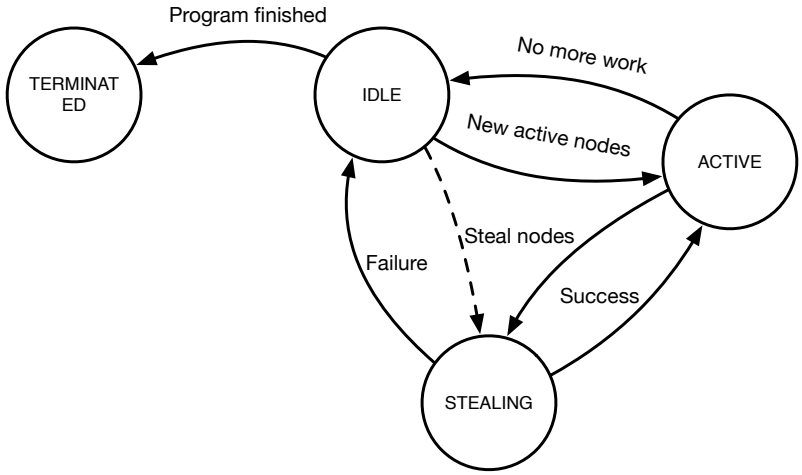


Figure 6.1: *The thread state machine as represented by the* State *flag. During the lifetime of a program, each thread goes through different states as specified by the state machine.*

The pseudo-code for the main thread loop is shown in Fig. 6.2. After a loop iteration, a thread inspects its *Work Queue* and while there are active nodes, procedure `process_node()` (complete pseudo-code in Fig. 6.4) will perform local computation on active nodes. When a thread's *Work Queue* is empty, it attempts to steal half of the nodes from another thread. Starting from a random thread, it cycles through all the threads to find one active thread from whom it will try to steal half of its nodes. If the thread fails to steal work, it will go **idle** and periodically attempt to steal work from another active thread. Eventually, all threads will fail to steal work since there is no more work to do and they will go idle. There is a global atomic counter that is used to detect termination. Once a thread goes idle, it decrements the global counter and changes its flag to idle. Since every thread will be busy-waiting and checking the global counter, they will detect a zero value and stop executing, transitioning to the **terminated** state.

Figure 6.3 ties everything together and presents the layout of our virtual machine for a program with six nodes and two running threads. In the figure, we represent the thread's *Work Queue* and the thread's logical space where the thread's subgraph is located. We also show the internals of node `@1` and the thread operations that force threads to interact with the node data structures.

In order to understand how threads interact with each other, we now review the node data structure that was presented in Section 5.2. The node lock *DB Lock* protects the data structures of the database, including the array, trie, linked list and hash table data structures and the *Rule Engine*. The *State Lock* structure protects everything else, especially the *State* flag and the temporary set of facts represented by the *Incoming Fact Buffer*. The *Incoming Fact Buffer* is used to hold logical facts that can not be added immediately to the database data structures. The *Owner*

```
Data: Thread TH
while true do
    TH.work_queue.lock();
    node ⟵ TH.work_queue.pop_node() ;
    TH.work_queue.unlock();
    if node then
        process_node(node);
    else
        // Attempt to steal some nodes.
        if !TH.steal_nodes() then
            TH.become_idle();
            while len(TH.work_queue) == 0 do
                // Try to terminate
                if TH.synchronize_termination() then
                    terminate;
                end
                if TH.steal_nodes() then
                    // Thread is still in the stealing state
                    break;
                end
            end
            // There's new nodes in the queue.
            TH.become_active();
        end
    end
end
```

Figure 6.2: *Thread work loop: threads process active nodes from the work queue until no more active nodes are available. Node stealing using a* steal half *strategy is employed when the thread has no more active nodes.*

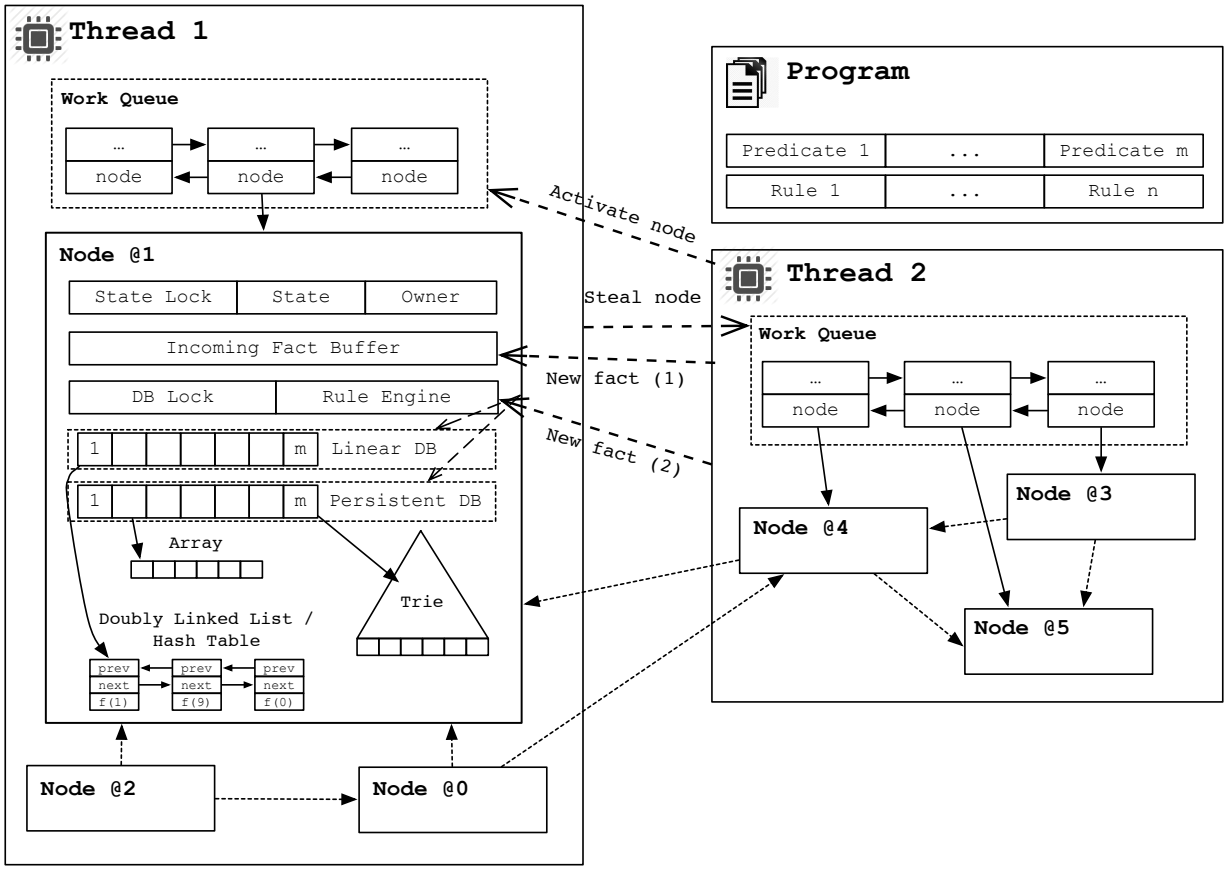


Figure 6.3: *Layout of the virtual machine. Each thread has a work queue that contains active nodes (nodes with facts to process) that are processed one by one by the thread. Communication between threads happens when nodes send facts to nodes located in other threads.*

field points to the thread responsible for processing the node and the *Rule Engine* schedules local computation.

Whenever a new fact is derived through rule derivation, we need to update the data structures for the corresponding node. If the node is currently being executed by the thread (local send), then the fact is added to the node data structures since the *DB Lock* is being held while the node is being executed. If that is not the case, then we have to synchronize since multiple threads might be updating the same data structures. For example, in Fig. 6.3, when thread 2 derives a fact to node `@1` (owned by thread 1), it first locks node `@1` using the *State Lock* and then it attempts to lock *DB Lock*, which gives thread 2 full access to the node. In this case, thread 2 adds the new fact to the database (*New fact (2)* in Fig. 6.3) and to the *Rule Engine*. However, if the *DB Lock* could not be acquired because node `@1` is currently being processed, then the new fact is added to *Incoming Fact Buffer* (*New fact (1)* in Fig. 6.3). The facts stored in *Fact Buffer* will then be processed whenever the corresponding node is processed again.

When a thread interacts with another thread to send a fact, it also needs to make sure that the target node is made **active** (see Fig. 5.2) and that it is also placed in the target thread's *Work Queue* (*Activate node* in Fig. 6.3). To handle concurrency issues, we have a (per thread) per *Work*

```
Data: Node N
N.state_lock.lock();
N.db_lock.lock();
// Add facts from the Incoming Fact Buffer into the database
N.DB.merge(N.incoming_fact_buffer);
N.state_lock.unlock();
N.rule_engine.run_rules();
N.db_lock.unlock();
// Check if node N is done for now
N.state_lock.lock();
if N.rule_engine.has_candidate_rules() or
            N.incoming_fact_buffer.has_facts() then
    N.state_lock.unlock();
    // Start from the beginning
    goto start_of_procedure;
end
N.state ⟵ inactive;
N.state_lock.unlock();
```

Figure 6.4: *Pseudo-code for the* `process_node` *procedure.*

Queue lock called the *Queue Lock* that is held when the *Work Queue* is being operated on. As an example, consider again the situation in which thread 2 sends a new fact to node `@1`. If node `@1` is not active, then thread 2 also needs to activate node `@1` by pushing it to the *Work Queue* of thread 1. After this synchronization point, the target thread is ensured to be active and with a new node to process.

We now summarize the synchronization hot-spots of the VM by describing how the locks are used in order to implement different operations:

- **Deriving New Facts**: We use the node's *State Lock* and then attempt to lock the *DB Lock*. If the *DB Lock* cannot be used, then the new facts are added to the *Incoming Fact Buffer*, otherwise the node database structures are updated with new facts. If the target node is not currently in any work queue, we lock the destination work queue and then add the node and change the state of the node to **active**. Finally, if the target thread that owns the target node is **idle**, we activate it by updating its state flag to **active**. The complete pseudo-code for this operation is shown in Fig. 6.5.
- **Node Stealing**: For node stealing, we acquire the lock of the target thread's queue and then copy the stolen node pointers to a temporary buffer. For each node, we use the *State Lock* to update its *Owner* attribute and then add it to the thread's work queue. Figure 6.6 presents the pseudo-code for this synchronization mechanism. Note that when using $pop\_half(stealing)$, the *Work Queue* will (unsafely) change the state of the nodes to **stealing**. When adding coordination to the language as presented in Chapter 7, the node stealing loop in the pseudo-code must have an extra check for cases when the node's owner is updated using coordination facts.

```
Data: Target Node N, Fact F
if N.db_lock.try_lock() then
    // New Fact (1)
    N.DB.add_fact(F);
    N.rule_engine.new_fact(F);
    N.db_lock.unlock();
    // Still need to check if node is active
    N.state_lock.lock();
else
    // New Fact (2)
    N.state_lock.lock();
    N.FactBuffer.add_fact(F);
end
// Activate node
TTH ⟵ N.Owner;
if N.state == inactive then
    TTH.work_queue.lock();
    TTH.work_queue.push(N);
    TTH.work_queue.unlock();
    N.state ⟵ active;
    if TTH.State == idle then
        TTH.become_active();
    end
end
N.state_lock.unlock();
```

Figure 6.5: *Synchronization code for sending a fact to another node.*

```
Data: Source Thread TH, Target Thread TTH
TTH.work_queue.lock();
nodes ⟵ TTH.work_queue.pop_half(stealing);
TTH.work_queue.unlock();
for node in nodes do
    node.state_lock.lock();
    if node.state! = stealing then
        // Node was put back into the queue, therefore we give up
        node.state_lock.unlock();
        continue
    end
    node.owner ⟵ TH;
    node.state ⟵ active;
    node.state_lock.unlock();
end
TH.work_queue.push(nodes);
```

Figure 6.6: *Synchronization code for node stealing (source thread steals nodes from target thread).*

- **Node Computation**: The *DB Lock* is acquired before any rule is executed on the node since node computation manipulates the database. The lock is released when all candidate rules are executed. The initial pseudo-code for the `process_node` procedure in Fig. 6.4 shows this synchronization protocol.
- **Node Completion**: Once node computation is completed and all candidate rules have been executed, the *State Lock* is acquired in order to change the state flag. Note that if newer facts have been derived by other nodes, computation is resumed on the current node instead of using another node from the *Work Queue*. The final section of `process_node` in Fig. 6.4 shows this synchronization protocol.

All the locks in the VM are implemented using Mellor-Crummey and Scott (MCS) spin-locks [MCS91], which are fair and contention-free spin-locks that use a FIFO queue to implement spin-lock operations.

In order to manipulate the *State* flag of each thread (see Section 6.1) we do not use locks but instead manipulate the state flag using lock-free *compare-and-set* operations to implement the thread state machine as specified in Fig. 6.1.

6.2 Runtime Data Structures And Garbage Collection

LM supports recursive types such as lists, arrays and structures. These compound data structures are immutable and shared between multiple facts. Such structures are stored in the heap of the VM and are managed through reference counting. For instance, each list is a *cons cell* with 3 fields: `tail`, the pointer to the next element of the list; `head`, the element stored by this element

of the list; and `refs`, which counts the number of pointers to this list element in the VM. The list is deleted from the heap whenever `refs` is decremented to zero.

Nodes are also subject to garbage collection. If the database of a node becomes empty and there are no references to the node from other logical facts, then the node is deleted from the program. We keep around a small number of freed nodes that can be reused immediately if another node is created. We avoid garbage collection schemes based on tracing since objects are created and discarded at specific points of the virtual machine. A reference counting mechanism is thus more appropriate than a parallel tracing garbage collector which would entail pausing the execution of the program to garbage collect the unused objects.

6.3 Memory Allocation

Memory allocation in the VM is extremely important because there is a need to repeatedly allocate and deallocate logical facts. Logical facts tend to be small memory objects since they are composed of two pointers (16 B) plus a variable number of arguments (8 B each), which may lead to fragmentation if allocation is not done carefully. Moreover, since the VM is multi threaded, allocation also needs to be scalable, therefore using the standard `malloc` facility provided by the POSIX standard is not preferred since each operating system uses a different implementation that may or may not scale well in multi threaded environments.

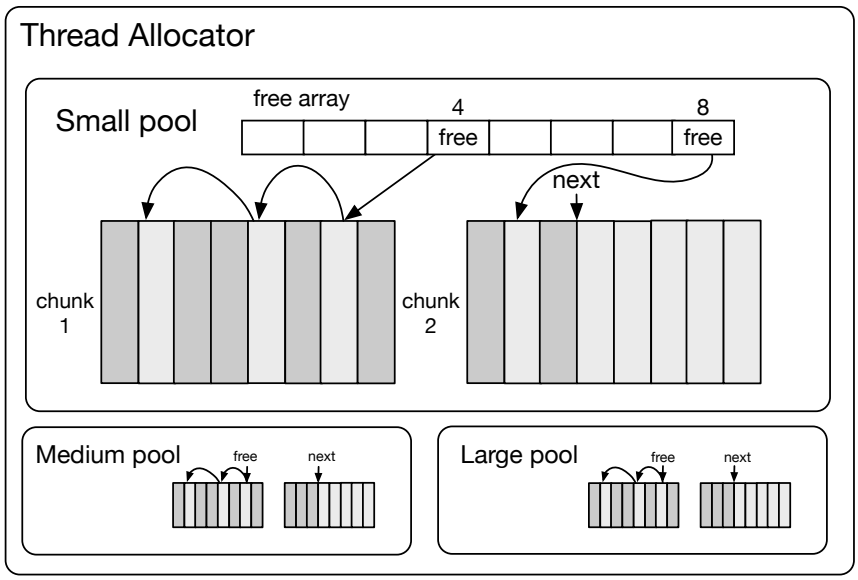


Figure 6.7: *Threaded allocator: each thread has 3 memory pools for objects and each pool contains a set of memory chunks. Each pool also contains an ordered array for storing deallocated objects of a particular size. In this example, the pool for small objects has an ordered array for sizes 4 and 8, where size 4 has 3 free objects and size 8 has just 1.*

In order to solve these two issues, we decided to implement a memory allocator inspired on the SLAB allocator [Bon94]. SLAB allocation is a memory management technique created in

the Solaris 5.4 kernel used to efficiently allocate kernel objects. Its advantages include reduced fragmentation and improved reuse of deallocated data structures since these are reused in newer allocations.

We pre-allocate three pools of memory chunks: one pool for small objects (at most 128 B), one pool for medium objects (at least 1 KB) and another pool for large objects (more than 1 KB). A memory pool is composed of multiple chunks (or *arenas*) which are large contiguous areas of memory. Each pool has a `next` pointer that points to a position in the current chunk, i.e., that which has objects that have not been allocated yet. The third and final element of the memory pool is the `free array`, which is an array which is ordered by size and each position points to a free object of a given size. When an object is freed, a binary search is performed on the sorted and the position for that object's size is updated to point to the deleted object. As expected, freed objects are reused whenever an object of the target size is requested by the system.

When allocating a particular object, we compute the object size and use the appropriate pool. First, we check if there are any free objects in the free list for that particular size and if there isn't any, we use the `next` pointer to rapidly allocate space on the current chunk of the memory pool. Whenever the `next` pointer reaches the end of the latest chunk, a bigger chunk is allocated and `next` is reset to the first position of the new chunk. The objects in `free` list are chained together so that it is possible to mark multiple objects as being deallocated. An example is presented in Fig. 6.7.

In order to reduce thread contention in the allocator, each thread uses a separate instance of the threaded allocator. When a thread wants to allocate an object, it asks its own allocator for a new object. When deallocating, a thread may deallocate an object that is part of another thread's chunk. This is not an issue since chunks are not garbage collected from the system, therefore both allocation and deallocation require no synchronization between threads.

Our threaded allocator does not attempt to place facts of the same node physically together. When firing rules, this may become an issue since the thread may need to read multiple chunks which were allocated by different threads, increasing the number of cache line misses. In Section 6.4.6, we explore an alternative allocator design and compare it against the allocator presented in this section.

6.4 Experimental Evaluation

In this section, we evaluate the performance and scalability of the VM. The main goals of our evaluation are:

- Compare the performance of LM programs against hand-written imperative C++ programs;
- Evaluate the scalability of LM programs when using up to 32 cores in parallel;
- Understand the impact and effectiveness of our dynamic indexing algorithm and its indexing data structures used for logical facts (namely, hash tables);
- Understand the impact of the memory allocator on scalability and multi core performance and compare it against alternative designs;

For our experimental setup, we used a computer with a 32 Core AMD Opteron(tm) Processor 6274 HE @ 1400 MHz with 32 GBytes of RAM memory running the Linux Kernel 3.18.6-100.fc20.x86_64. The C++ compiler used is the GCC 4.8.3 (g++) with the following `CXXFLAGS` flags: `-O3 -std=c++11 -march=x86-64`. All experiments were executed 3 times and the running times were averaged.

6.4.1 Sequential Performance

To understand the absolute performance of LM programs, we compared their execution time using a multi threaded capable virtual machine running on a single thread against hand-written sequential C++ programs[1]. All C++ programs were compiled with the same compilation flags used for LM for fairness. Arguably, compiled C/C++ programs are a good standard for comparing the performance of new programming languages since they tend to offer the best performance on several popular benchmarks [lan]. Python is a *scripting* programming language that is much slower than compiled C/C++ programs and thus can be seen as a good lower-bound in terms of performance.

The goal of the evaluation is to understand the effectiveness of our compilation strategy and the effectiveness of our dynamic indexing algorithms, including the data structures (hash trees) used to index logical facts. We used the following programs in our experiments[2]:

- Belief Propagation: a machine learning algorithm to de-noise images. Program is presented in Section 8.3.3.
- Heat Transfer: an asynchronous program that solves the heat transfer equation using an implicit method for transferring heat between nodes. Program is presented in Section 7.8.3.
- Multiple Single Shortest Distance (MSSD): a program that computes the shortest distance from a subset of nodes of the graph to all the nodes in the graph. A modified version is later presented in Section 7.1.
- MiniMax: the AI algorithm for selecting the best player move in a Tic-Tac-Toe game. The initial board was augmented in order to provide a longer running benchmark. Program is presented in Section 7.8.1.
- N-Queens: the classic puzzle for placing queens on a chess board so that no two queens threaten each other. Program is presented in Section 7.8.2.

Table 6.1 presents the comparison between LM and sequential C++ programs. Comparisons to other systems are shown under the **Other** column, which presents the speedup ratio of the C++ program over the target system (numbers greater than 1 indicate C++ is faster). Note that the systems used are executed with a single thread. Since we also want to assess the VM's scalability, we use different dataset sizes for each program.

The Belief Propagation experiment is the program where LM performs the best when compared to the C++ version. We found out that the mathematical operations required to update the nodes belief values are expensive and make up a huge part of the total computation time. This

[1] All C++ programs available at `http://github.com/flavioc/misc`.
[2] All LM programs available at `http://github.com/flavioc/meld/benchs/progs`.

Program	Size	C++ Time (s)	LM	Other
Belief Propagation	50x50	3.16	1.27	1.08 (GraphLab)
	200x200	49.36	1.36	1.25 (GraphLab)
	300x300	135.56	1.35	1.25 (GraphLab)
	400x400	169.99	1.35	1.27 (GraphLab)
Heat Transfer	80x80	4.62	7.28	-
	120x120	20.29	7.07	-
MSSD	US 500 Airports	0.69	2.76	13.44 (Python) 0.25 (Ligra)
	OCLinks	7.00	7.35	16.10 (Python) 0.34 (Ligra)
	EU Email	13.47	2.08	9.80 (Python) 0.32 (Ligra)
	Twitter	27.22	8.58	8.28 (Python) 0.27 (Ligra)
	US Power Grid	55.33	5.64	10.99 (Python) 0.30 (Ligra)
	Live Journal	221.91	4.38	5.75 (Python) 0.20 (Ligra)
	Orkut	281.59	1.66	4.23 (Python) 0.23 (Ligra)
MiniMax	Small	2.89	7.58	27.43 (Python)
	Big	21.47	8.81	-
N-Queens	11	0.28	1.81	20.36 (Python)
	12	1.42	3.09	24.30 (Python)
	13	7.90	4.99	27.85 (Python)
	14	47.90	6.42	31.52 (Python)

Table 6.1: *Experimental results comparing different programs against hand-written versions in C++. For the C++ programs, we show the execution time in seconds (**C++ Time (s)**). In columns **LM** and **Other**, we show the speedup ratio of C++ by dividing the run time of the target system by the run time of the C++ program. Numbers greater than 1 mean that the C++ program is faster.*

is clearly shown by the low overhead numbers. We also compared our performance against the GraphLab and LM is only slightly slower, which is also a point in LM's favor.

The Heat Transfer program behaves somewhat like Belief Propagation but LM is almost an order of magnitude slower than the C++ version. We think this is because the heat transfer computation is small which tends to show the overhead of the VM.

For the MSSD program, we used seven different datasets:

- US 500 Airports [CPSV07, Ops15], a graph of the 500 top airports in the US with around 5000 connections. The shortest distance is calculated for all nodes;
- OCLinks [Ops15, OP09], a Facebook-like social network with around 2000 nodes and 20000 edges. The shortest distance is calculated for all nodes;
- US Power Grid [Ops15, WS98], the power grid for western US with around 5000 nodes and 13000 edges. The shortest distance is calculated for all nodes;
- Twitter [LK14, LM12], a graph with 81306 nodes and 1768149 edges. The shortest distance is calculated for 40 nodes;
- EU Email [LK14, LKF07] a graph with 265000 nodes and 420000 edges. The shortest

distance is calculated for 100 nodes;

- Orkut [LK14, YL12], a large graph representing data from the Orkut social network. The graph contains 3072441 nodes and 117185083 edges.
- Live Journal [LK14, Bac06], a large graph representing data from the Live Journal social network. The graph contains 4847571 nodes and 68993773 edges.

The C++ MSSD version applies the Dijkstra algorithm for each node we want to compute the shortest distance from. While the Dijkstra algorithm has a better complexity than the algorithm used in LM, LM's algorithm is able to process distances from multiple sources at the same time. Our experiments show that the C++ program effectively beats LM's version by a large margin, but that gap is reduced when using larger graphs such as EU Email. The Python version also uses the Dijkstra algorithm and is one order of magnitude slower than the C++ version and usually slower than LM. We also wrote the MSSD program using the Ligra system [SB13][3], an efficient and scalable framework for writing graph-based programs. Our experiments show that Ligra is, on average, three times as fast as our C++ program, which means that LM is around 15 times slower than Ligra. We also experimented with Valgrind's cachegrind tool [NS07], and measured the number of cache misses for LM, C++, and Ligra. For instance, in the OCLinks dataset, Ligra's MSSD program has only a 5% cache miss rate (for 700 million memory operations) while the C++ version has a 9.4% miss rate (for 2 billion memory operations), which may explain the differences in performance between C++ and Ligra. LM shows a 6% cache miss rate but also a much higher number of memory accesses (seven times more than the C++ program). Note that we compiled Ligra without parallel support.

For the MiniMax program, we used two different starting states, Small and Big, where the tree generated by Big is ten times larger than the one generated by Small. The C++ version of the MiniMax program uses a single recursive function that updates a single state as it is recursively called to generate the best score and corresponding move. The LM version is seven to eight times slower due to the memory requirements and costs of creating new nodes using the exists construct. In Chapter 7, we will show how to improve the space complexity of the MiniMax program and the corresponding run time.

The LM's N-Queens programs shows some scalability issues since the overhead ratio increases as the size of the problem increases. However, the same behavior is seen in the Python program. The C++ version uses a backtracking strategy to try out all the possibilities and uses a vector to store board states. Since there is only at most N (size of the board) vectors at the same time, it shows better behavior than all the other programs. However, we should note that a 3 to 6-fold slowdown is a good trade-off for a higher-level program that will execute faster when using more threads as we are going to observe next.

From these results, it is possible to conclude that LM's virtual machine offers a decent performance when compared to hand-written C++ programs. We think these results originate from four main aspects of our system: efficient indexing, good locality with array data structures for persistent facts, an efficient memory allocator, and a good compilation scheme. As noted in [MIM15], scalability should not be the sole focus of a parallel/distributed system such as LM. This is even more important in declarative languages which are known to suffer from performance issues

[3] Available at http://github.com/flavioc/ligra.

when compared to programming languages such as C++.

6.4.2 Memory Usage

To complete our comparison against the C++ programs, we have measured and compared the average memory used by both systems. Table 6.2 presents the memory statistics for LM programs while Table 6.3 presents statistics of the C++ programs and a comparison to LM's memory requirements. In this thesis, the memory usage is measured by the memory occupied by all the live objects in the virtual machine. The **Final** column shows the total memory usage after the program finishes firing the last rule and the **Average** column indicates the average memory usage of the program throughout its lifetime [4].

In the Belief Propagation program, LM uses 2 to 3 times more memory than the corresponding C++ version. This is because the nodes in the LM program keep a copy of the belief values of neighbor nodes, while the C++ program uses shared memory and the stack to read and compute the neighbors belief values. The LM version of Heat Transfer has a slightly larger gap when compared to C++, since heat values are represented as integers while the belief values in Belief Propagation are represented as arrays, which is a more compact representation when compared to the memory required to store linear facts.

When the MiniMax program completes, there are only two facts on the database that indicate the best player move. The MiniMax program is also the only program in this experiment that dynamically generates a (tree) graph, which is destroyed once the best move is found. The VM's garbage collector that collects empty nodes is able to delete the tree nodes (except the root) and the **Final** memory usage reflects that since it is much smaller than the **Average** statistic. However, because the garbage collector retains a small number of freed nodes that may be reused later, the average size per fact is 15KB, which also includes those freed nodes. Note that the memory usage of the C++ program is much smaller because it uses function calls to represent the tree structure of the MiniMax algorithm.

The MSSD program shows that the LM VM requires about 2 to 5 times more memory than the corresponding C++ program. The ratio is larger when the graph and computed distances are smaller and this is due to the extra data structures required by the VM (i.e., the node data structure). For the Live Journal and Orkut datasets, LM's memory usage is about the same or smaller than the C++ program. We think this is because the VM uses hash tree data structures, which use less memory than the hash table data structures in the C++ program. In terms of average memory per fact, we see that the MSSD requires on average 100B, where a big part of it are the hash table data structures used for indexing. For the Orkut dataset, where 100 million facts are derived, the average memory per fact is exactly 30B, which is about the 32B required to store a particular linear fact for representing one shortest distance.

For the N-Queens program, the results show why there are scalability issues when using a larger N since the memory usage increases significantly. On the positive side, the average memory usage per fact remains the same for all data sets. In respect to the C++ program, it is expected that it should consume almost no memory because it uses the stack to compute the

[4]The **Average** values were computed by taking memory usage measurements every 100 milliseconds, from which an average was computed.

Program	Size	Average	Final	# Facts	Each
Belief Propagation	50x50	3.7MB	3.6MB	34.4K	0.11KB
	200x200	58.5MB	57.6MB	557.6K	0.11KB
	300x300	131.1MB	129.6MB	1256.4K	0.11KB
	400x400	233.3MB	230.5MB	2.2M	0.11KB
Heat Transfer	80x80	8.7MB	8.2MB	63.3K	0.13KB
	120x120	19.6MB	18.4MB	143.4K	0.13KB
MSSD	US 500 Airports	27.8MB	19.3MB	164.3K	0.12KB
	OCLinks	502.1MB	231.9MB	2.2M	0.11KB
	EU Email	485.4MB	401.8MB	3.8M	0.13KB
	Twitter	1125.7MB	362.9MB	4.5M	0.08KB
	US Power Grid	2.7GB	2.6GB	24.4M	0.11KB
	Live Journal	5.5GB	3.3GB	71.5M	0.05KB
	Orkut	3.6GB	2.9GB	100.6M	0.03KB
MiniMax	Small	1667.9MB	30KB	2	15.00KB
	Big	13.5GB	30KB	2	15.00KB
N-Queens	11	2.8MB	667KB	3.2K	0.20KB
	12	13.4MB	2.9MB	14.9K	0.20KB
	13	71MB	14.5MB	74.5K	0.20KB
	14	403.7MB	73.1MB	366.5K	0.20KB

Table 6.2: *Memory statistics for LM programs. The meaning of each column is as follows: column* ***Average*** *represents the average memory use of the program over time;* ***Final*** *represents the memory usage after the program completes;* ***# Facts*** *represents the number of facts in the database after the program completes;* ***Each*** *is the result of dividing* ***Final*** *by* ***# Facts*** *and represents the average memory required per fact.*

solutions.

We conclude that the LM VM has high memory requirements due to the relatively large size of the node data structures (including the rule engine and indexing data structures). The high memory usage tends to degrade the performance of programs when compared to more efficient languages and systems.

6.4.3 Dynamic Indexing

In the previous experiments, we used the full functionality of the virtual machine, including dynamic indexing and indexing data structures. We now evaluate the impact of the dynamic indexing algorithm by running the previous experiments without dynamic indexing. Table 6.4 shows the comparison between the versions with and without indexing. Column **Run Time** shows that the MSSD program benefits from indexing because the version without indexing is around 2 to 100 times slower than the version with indexing enabled. Since MSSD computes the shortest distance to multiple nodes, its rules require searching for the shortest distance facts of arbitrary nodes. All the other programs do not require significant indexing but also do not

Program	Size	Average	C / LM	Final	C / LM
Belief Propagation	50x50	2.7MB	73%	2.7MB	45%
	200x200	45.1MB	77%	45.1MB	35%
	300x300	99.7MB	76%	99.8MB	38%
	400x400	181.3MB	77%	181.4MB	35%
Heat Transfer	80x80	2.3MB	26%	2.3MB	20%
	120x120	5.4MB	27%	5.4MB	23%
MSSD	US 500 Airports	7.8MB	28%	15.7MB	23%
	OCLinks	76.6MB	15%	151.9MB	14%
	EU Email	267.3MB	55%	298.2MB	44%
	Twitter	245.3MB	21%	333.10MB	15%
	US Power Grid	744.5MB	35%	1491.7MB	32%
	Live Journal	5.4GB	99%	5.4GB	64%
	Orkut	7.5GB	206%	7.6GB	109%
MiniMax	Small	1KB	0%	0KB	0%
	Big	1KB	0%	0KB	0%
N-Queens	11	1KB	0%	528KB	8%
	12	1KB	0%	2.5MB	6%
	13	1KB	0%	16.9MB	13%
	14	1KB	0%	75.9MB	9%

Table 6.3: *Average and final memory usage of each C++ program. The columns **C / LM** compare the memory usage of C programs against the memory usage of LM programs for the average and final memory usage, respectively (higher numbers are better).*

display any significant slowdown from using dynamic indexing. In terms of memory usage, the version with indexing uses slightly more memory, especially for the MSSD program, requiring, on average, 50% more memory due to the existence of hash trees used to support indexing.

6.4.4 Scalability

In this section, we analyze the scalability of several LM programs using up to 32 threads. Note that we are studying declarative LM programs that do not make use of coordination. We also compare the run times against the run time of the hand-written C++ programs when such run times are available. We used the programs of the previous section with a few additions:

- PageRank: an asynchronous version of the PageRank program without synchronization between iterations. Every time a node sends a new PageRank value to its neighbors and the change is significant, then the neighbors are scheduled to recompute their PageRanks.
- Greedy Graph Coloring (GGC): an algorithm that colors nodes in a graph so that no two adjacent nodes have the same color. We start with a small number of colors and then we expand the number of colors when we cannot color the graph.

Program	Size	Run Time	Average Memory
Belief Propagation	50x50	0.88	0.03
	200x200	1.00	0.02
	300x300	1.08	0.02
	400x400	1.01	0.03
Heat Transfer	80x80	1.01	1.00
	120x120	1.12	1.00
MSSD	US 500 Airports	34.57	1.65
	OCLinks	108.93	1.36
	EU Email	3.86	1.45
	Twitter	1.97	1.19
MiniMax	Small	0.98	1.00
	Big	0.95	1.00
N-Queens	11	0.98	1.00
	12	1.00	1.00
	13	1.00	1.00
	14	1.00	1.00

Table 6.4: *Measuring the impact of dynamic indexing and related data structures. Column **Run Time** shows the slow down ratio of the unoptimized version (numbers greater than 1 show indexing improvements). Column **Average Memory** is the result of dividing the average memory of the optimized version by the unoptimized version (large numbers indicate that more memory is needed when using indexing data structures).*

These two programs were not used in the previous section because we did not implement the corresponding C++ program.

We executed all the programs using 17 configurations. The base configuration uses 1 thread and the other 16 configurations use an even numbers of threads for a maximum of 32 threads. Figure 6.8 presents the overall scalability computed as the weighted harmonic mean of all the benchmarks presented in this section. The weight used in the harmonic mean formula is the run time of each program when executed with one thread. The one threaded version executes with all the synchronization mechanisms enabled, just like the multi-threaded executions. The experimental results show that, on average, our benchmarks have a 15-fold speedup when using 32 threads.

We now analyze each program separately. For the plots presented next, the x axis represents the number of threads used, the left y axis represents the run time (in milliseconds) for that particular configuration and uses a logarithmic scale and the right axis y represents the speedup computed as T_1/T_i, where T_i represents the run time of the program using i threads. In some plots, there is also an horizontal black line that represents the run time of the C++ program and is used to understand how many threads are needed in order for an LM program to run faster than the C++ program. Note that all run times are the average of three runs.

The scalability results for the Belief Propagation program are presented in Fig. 6.9. This program has the best scalability with a 60-fold speedup for 32 threads. This is because the de-

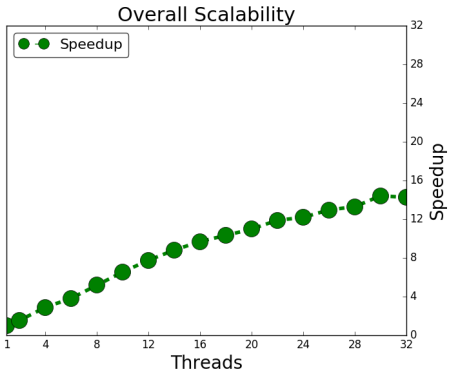


Figure 6.8: *Overall scalability for the benchmarks presented in this section. The average speedup is the weighted harmonic mean using the single threaded run time as the weight of each benchmark.*

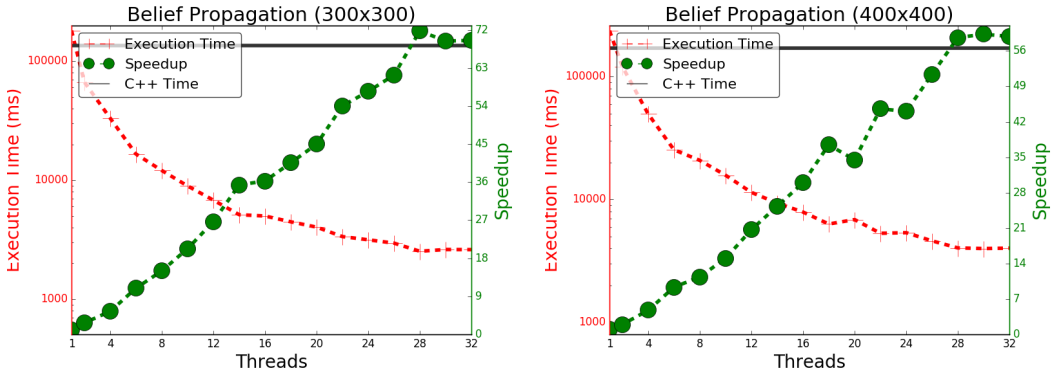


Figure 6.9: *Scalability results of the Belief Propagation program.*

fault ordering used for 1 thread is not optimal, while the same ordering works better for multiple threads. Additionally, the existence of multiple threads increase the amount of up-to-date information from neighbor nodes, resulting in super-linear speedups. Note that we also used the same computation ordering in the C++ program, which is also the one used in the GraphLab framework.

The results for the Heat Transfer program are shown in Fig. 6.10. We note that LM requires around 15 threads to reach the run time of the C++ program. This results from the poor absolute performance of the LM program that was mentioned in the previous section. Although we use a grid dataset, the work available in the graph is not equally distributed due to different initial heat values, therefore an almost linear speedup should not be expected. When comparing the two datasets used, the 80x80 configuration has less scalability than the 120x120 dataset due to its smaller size. For the 120x120 dataset, LM has a 12-fold speedup for 32 threads, which makes the LM version run almost twice as fast as the C++ version.

Figure 6.11 presents the results for the Greedy Graph Coloring (GGC) program. In GGC, we use two datasets: Google Plus [LK14], a graph representing social circles in Google Plus with

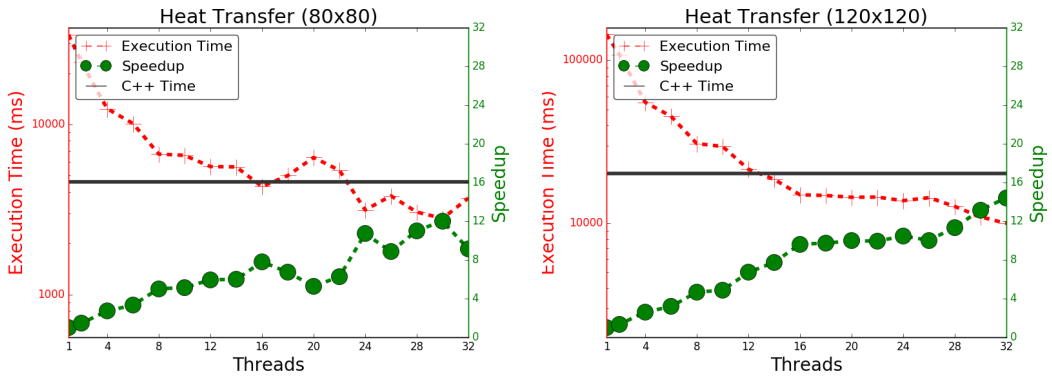


Figure 6.10: *Scalability results for the Heat Transfer program. Because the Heat Transfer program performs poorly when compared to the C++ program, the LM version needs at least 15 threads to reach the run time of the C++ program (for the 120x120 dataset).*

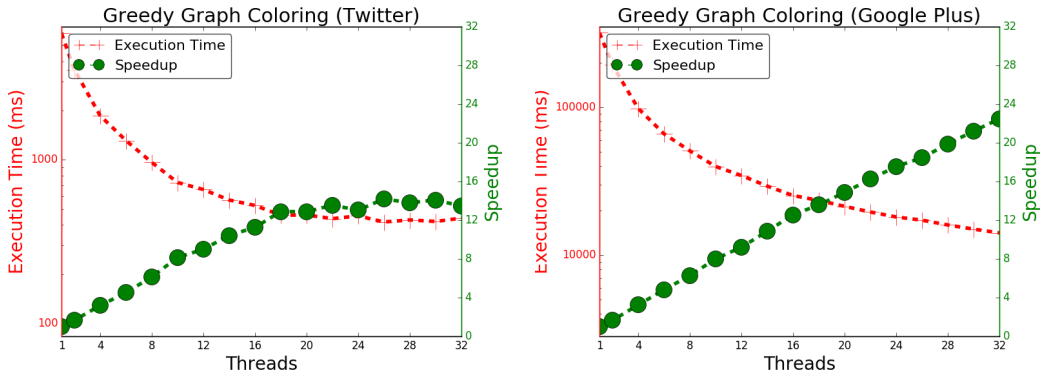


Figure 6.11: *Scalability of the Greedy Graph Coloring program.*

107614 nodes and 13673453 edges, and Twitter, a graph representing Twitter networks with 81306 nodes and 1768149 edges (note that Twitter was also used before in the MSSD program). The Twitter and Google Plus datasets have almost the same number of nodes but Google Plus has ten times more edges, which makes coloring more time consuming. Since Twitter is a much smaller dataset, its speedup is much smaller than Google Plus, as expected. However, while the speedup of Twitter starts to drop after 16 threads, the LM system is still able to make it faster by adding more threads. For Google Plus, a 20-fold speedup is achieved for 32 threads, which is a reasonable scalability considering that GGC is not a program where work is equally distributed.

The results for the N-Queens program are shown in Fig. 6.12. We decided to just use the 13 and 14 configurations since those take longer to run. The 13-Queens configuration has 169 (13x13) nodes while the 14-Queens configuration has 196 (14x14) nodes. The LM program considers the chess board as a graph and the bottom rows have more work to perform since the valid queens placements are built from the top to the bottom row. For a more in-depth explanation of the N-Queens program please see Section 7.8.2.

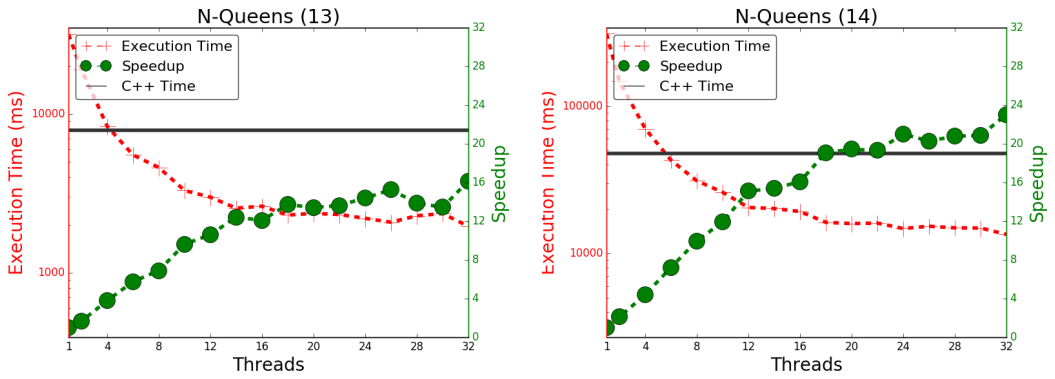


Figure 6.12: *Scalability results for the N-Queens program. The 13 configuration has 73712 solutions, while the 14 configuration has 365596 solutions. In terms of graph size, the 13 configuration has* 13×13 *nodes while the 14 configuration has* 14×14 *nodes.*

Due to the small number of nodes in the N-Queens program and the fact that more work is performed at the bottom rows, the N-Queens program is hard to parallelize. Furthermore, the solutions for the puzzle are built by constructing lists which are shared between threads. This introduces some potential false sharing because the reference count of each list data structure needs to be updated when new solutions are constructed. The 14-Queens configuration has good efficiency using 16 to 20 threads, while the 13-Queens configuration efficiency starts to drop after 14 threads. One can argue that the best number of threads is correlated to the size of the board, because the second half of execution is better balanced and partitioned when the number of threads matches the size of the board.

The speedup results for the MiniMax program is shown in Fig. 6.13. This is an interesting program because the graph is constructed during run time. Furthermore, due to the LM default scheduling order, the tree is explored in a breadth-first fashion, requiring the program to hold the complete MiniMax tree in memory. The Big configuration, for instance, uses, on average, 14GB of memory and a total of 30GB of memory before collecting the tree. Since the machine used for the experiments has only 32GB, the Small configuration scales better due to the smaller memory requirements. In Section 7.8.1, we will show how this program can be improved by changing the default scheduling order of LM and improving the memory usage of the program.

The scalability results for the MSSD program are shown in Fig. 6.14. The amount of work required to compute the results of each dataset depends on the size of the graph and the number of nodes from which the distance must be computed. The first conclusion is that more work implies more scalability, and the US Power Grid dataset has a 20-fold speedup for 32 threads, the best from the 6 datasets used. The second conclusion is that all datasets are able to, at least, reach the execution speed of the corresponding C++ program. However, some datasets such as EU Email or Orkut, are notoriously faster than C++ when using more than 5 threads, which is a impressive result. We think this is because the number of source nodes is small. Furthermore, LM as a slight advantage over the C++ program because, in LM, distances to different nodes are propagated at once, while in the C++ version, the Dijkstra algorithm runs in iterations - one for

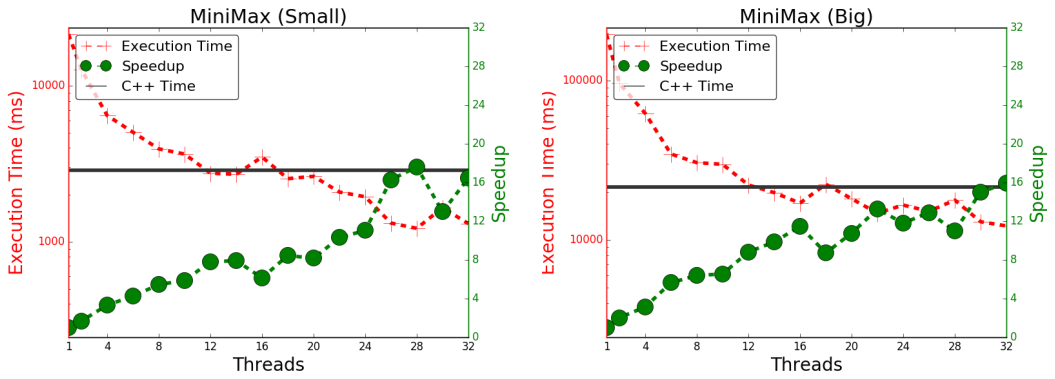
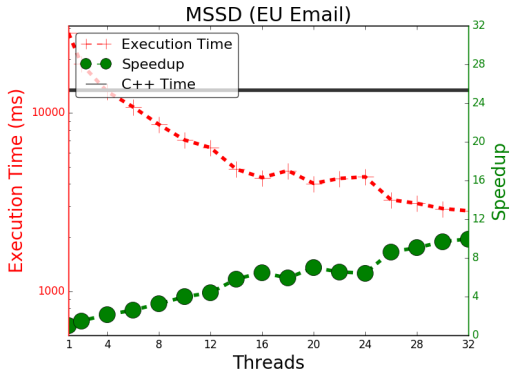


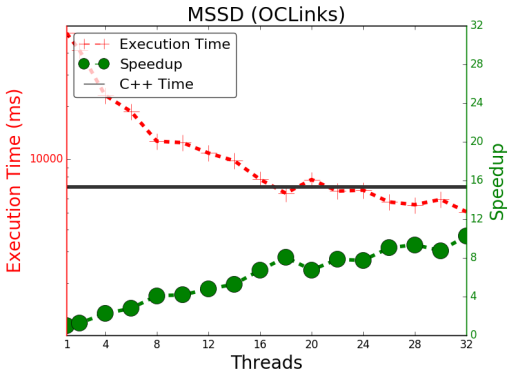
Figure 6.13: *Scalability for the MiniMax program. Although the Big configuration has more work available and could in principle scale better than Small, the high memory requirements of Big makes it scale poorly.*

each node from which we want to calculate the distance from.

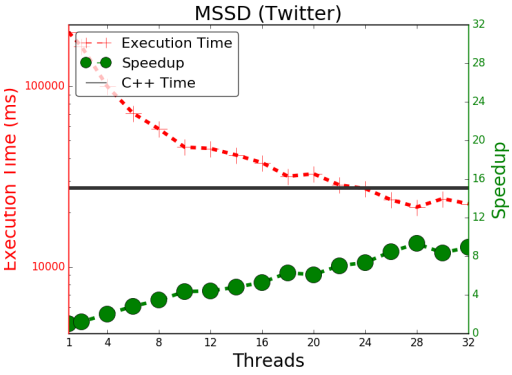
The final scalability results are shown for PageRank in Fig. 6.15. We used two datasets, Google Plus and Pokec. Google Plus [LK14] has 107614 nodes and 13673453 edges and is based on a Google Plus social network, while Pokec [LK14] has 1632803 nodes and 30622564 edges and represents data from a popular social network website in Slovakia. Even though Pokec is the larger graph, Google Plus has a denser graph, where the average number of edges per node is 127 compared to the average of 18 for Pokec. This may explain why Google Plus is more scalable with a 14-fold speedup for 32 threads.



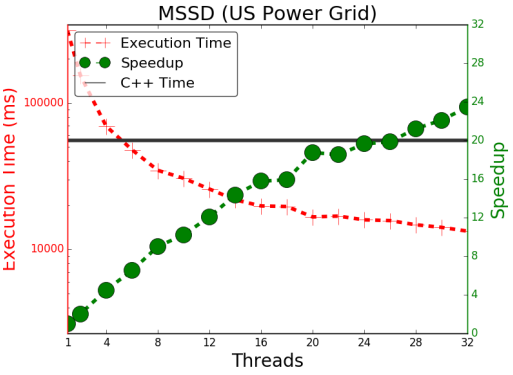
(a) *Graph with 265000 nodes and 420000 edges. The shortest distance is calculated for 100 nodes.*



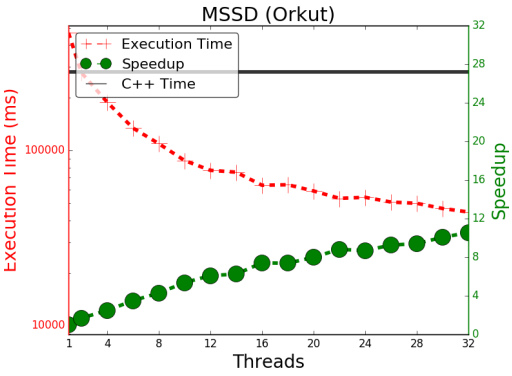
(b) *Graph with around 2000 nodes and 20000 edges. The shortest distance is calculated for all nodes.*



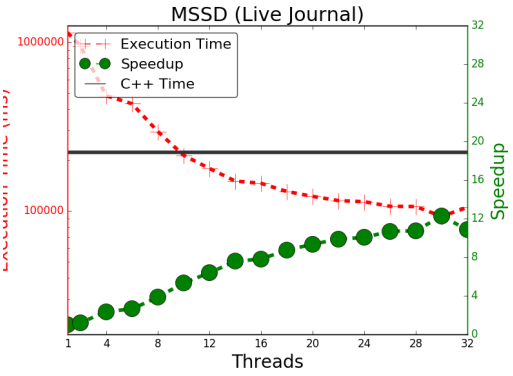
(c) *Graph with 81306 nodes and 1768149 edges. The shortest distance is calculated for 40 nodes.*



(d) *Graph with around 5000 nodes and 13000 edges. The shortest distance is calculated for all nodes.*

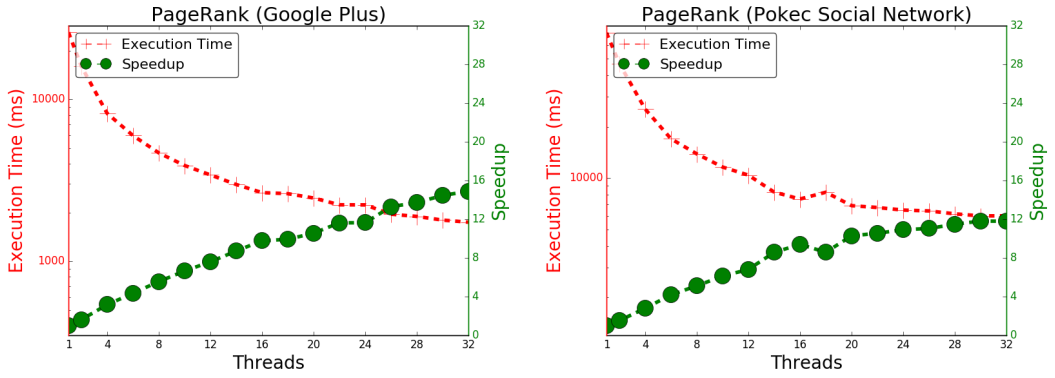


(e) *Graph with 3072441 nodes and 117185083 edges. The shortest distance is calculated for two nodes.*



(f) *Graph with around 4847571 nodes and 68993773 edges. The shortest distance is calculated for two nodes.*

Figure 6.14: *Scalability for the MSSD program. The LM system scales better when there is more work to do per node, with the US Power Grid dataset showing the most scalability.*



(a) *The Google Plus graph has a high average of edges per node of 127.*

(b) *The Pokec graph has a lower average of edges per node of 18.*

Figure 6.15: *Scalability results for the asynchronous PageRank program. The superior scalability of the Google Plus dataset may be explained by its higher density of edges.*

6.4.5 Threaded allocator versus `malloc`

In order to understand the role that our threaded allocator plays in the performance of programs, we evaluate and compare it against the performance of the LM system by using the `malloc` function provided by the operating system.

Figure 6.16 presents several programs comparing `malloc` with the threaded allocator. The programs Belief Propagation, MiniMax, and N-Queens have poor scalability when using the `malloc` operator due to high thread contention when calling `malloc` since those programs need to allocate many small objects. The PageRank program shows good scalability with `malloc` but it is only because the configuration with 1 thread runs slowly when compared to the other configurations. For all the remaining programs, there is less overall performance and scalability when compared to the threaded allocator.

As these results show, the performance of the memory allocator is crucial for good performance and scalability. Linear logic programs spend a significant amount of time asserting and retracting linear facts which require that memory allocation must be done efficiently and without starving other threads. Reuse of deallocated linear facts also makes more likely that threads will hit hot cache lines and thus improve speed significantly.

6.4.6 Threaded allocator versus node-local allocator

While our default threaded allocator performs and scales well, we also explored some alternative designs. In this section, we explore a design where we move the allocation decisions from the thread to the node itself, in order to store the facts of the same node close together and thus increase locality. However, this design requires locking since multiple threads may allocate facts from the same node. Our expectation is that such costs will be offset by the increased locality and reduced cache misses when deriving facts.

The node-local allocator allocates pages of memory from the threaded allocator which are then used to allocate facts for that particular node. When a thread needs to allocate or deallocate a fact, it acquires the allocator lock of the target node's allocator and performs the allocation operation. There is a doubly-linked list of memory pages and each page contains facts of different sizes (predicates).

Figure 6.17 presents an example state of a node-local allocator. The node has 3 memory pages, all connected using the `next` and `prev` pointers. Each page also has a reference count (`refcount`) of the objects allocated in the page. If the reference count ever drops to zero, then the memory page is deallocated. Deallocated facts are kept on an ordered array for different sizes using the mechanism we implemented for the threaded allocator. We have decided to reference count objects in the node-local allocator because it is more difficult to share objects between nodes and maintaining a reference count helps reduce memory usage.

Figure 6.18 presents the comparison between the threaded and node-local allocator for the most relevant datasets (the performance was similar for the other datasets).

The first major observation from the comparison is that the threaded allocator has better sequential performance than the node-local allocator for most programs. This is probably the result of better locality for the threaded allocator because it does not need to create pages for each node and instead uses the thread pages to maintain all facts, reducing cache line misses. Overall,

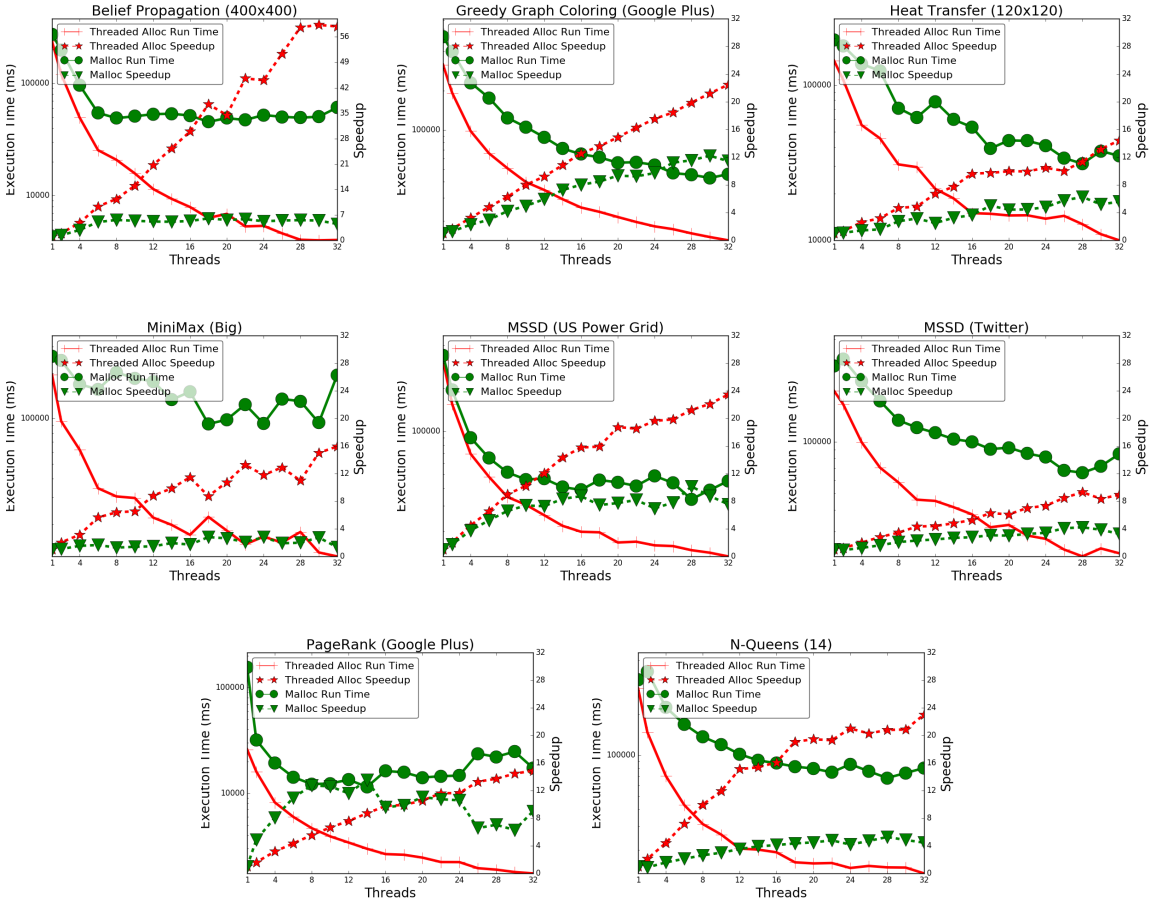


Figure 6.16: *Comparing the threaded allocator described in Section 6.3 against the* `malloc` *allocator provided by the C library.*

the performance of the threaded allocator is better or similar than the node-local allocator, for both single and multi threaded threaded execution (see GGC for a clear example).

The second observation is in the N-Queens program, where the threaded allocator beats the node-local allocator by a high margin in terms of scalability and performance. Note that the N-Queens program allocates many lists, which are not allocated on a per node basis but on a thread basis, therefore the extra work required to maintain each node-local allocator is not offset by the increased locality because the threads need to traverse lists to find valid board states.

To make our comparison more interesting, we also deactivated the reference counting mechanisms of the node-local allocator and compared it against the threaded allocator. The goal is to understand how reference counting affects the performance of the node-local allocator. The complete comparison are shown in Fig. 6.19. In a nutshell, it performs almost the same as the full featured node-local allocator, except for programs which require more memory such as MiniMax and Heat Transfer. In the case of MiniMax, we were unable to completely execute the Big dataset since the VM required more memory than the available memory, resulting in long execution times. This is the result of not collecting unused memory pages, which results in high

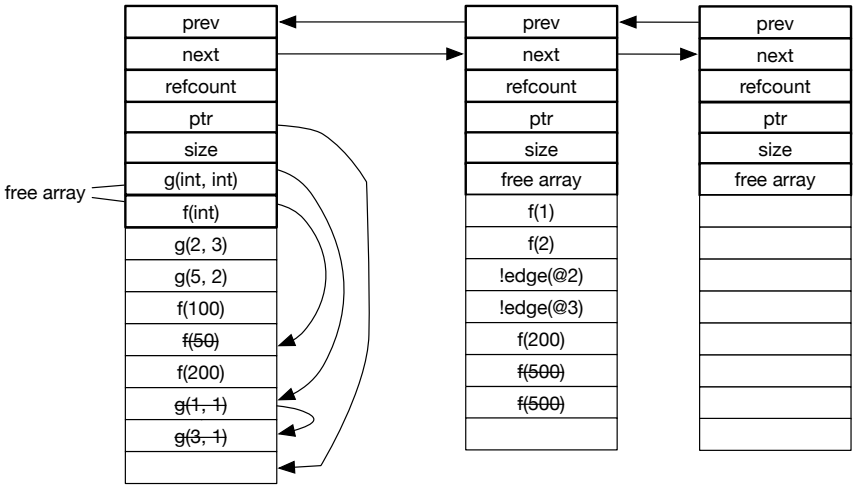


Figure 6.17: *Fact allocator: each node has a pool of memory pages for allocating logical facts. Each page contains: (i) several linked lists of free facts of the same size (*`free_array`*); (ii) a reference count of used facts (*`refcount`*); (iii) a* `ptr` *pointer that points to unallocated space in the page. In this figure, predicates* `f` *and* `g` *have several deallocated facts that are ready to be used when a new fact needs to be acquired.*

memory usage.

Overall, the performance of both allocators is similar, therefore we have decided to use the threaded allocator by default since it is simpler and offers good performance across the board. Furthermore, the threaded allocator requires less allocated memory because it enables more sharing between nodes.

6.5 Related Work

6.5.1 Virtual Machines

Virtual machines are a popular technique for implementing interpreters for high level programming languages. Due to the increased availability of parallel and distributed architectures, several machines have been developed with parallelism in mind [KDGN97]. One good example is the Parallel Virtual Machine (PVM) [Sun90], which servers as an abstraction to program heterogeneous computers as a single machine. Another important machine is the Threaded Abstract Machine (TAM) [CGSvE93, Gol94], which defines a self-scheduled machine language of parallel threads where a program is represented using conventional control flow.

Prolog, the most prominent logic programming language, has a rich history of virtual machine research centered extending/adapting the Warren Abstract Machine (WAM) [War83]. Because Prolog programs tend to be naturally parallel, much research has been done to either parallelize the WAM or create new parallel machines. Two types of parallelism are possible in Prolog: *OR-parallelism*, where several clauses for the same goal can be executed, and *AND-parallelism*, where goals in the same clause are tried in parallel. For OR-parallelism, we have several mod-

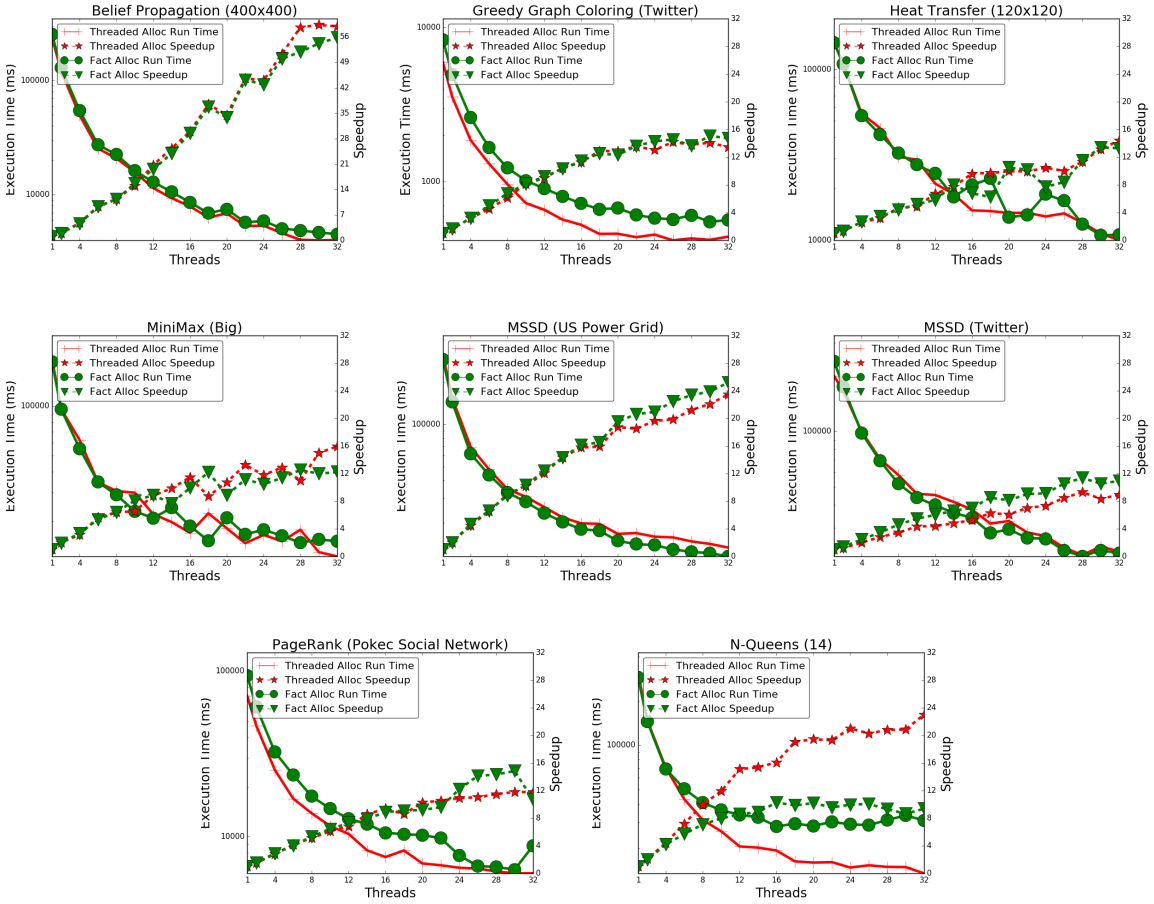


Figure 6.18: *Comparing the threaded allocator described in Section 6.3 against the node-local allocator.*

els such as: the SRI model [War87], the Argonne model [BDL+88], the MUSE model [AK90] and the BC machine [Ali88]. For AND-parallelism, different implementations were built on top of the WAM [Her86, LK88], however more general models such as the Andorra model [HJ90] were developed which allows both AND and OR-parallelism. In contrast to LM, where parallelism arises from computing on different nodes with a partitioned database of facts, AND and OR-parallelism arises from proving different facts independently.

6.5.2 Datalog Execution

The Datalog language is a forward-chaining logic programming and therefore requires different evaluation strategies than those used by Prolog, which is a backward-chaining logic programming language. Arguably, the most well-known strategy for Datalog programs with recursive rules is the *Semi-Naïve Fixpoint* algorithm [BR87], where the computation is split into iterations and the facts generated in the previous generation are used as inputs to derive the facts in the next iteration. The advantage of this mechanism is that no redundant computations are per-

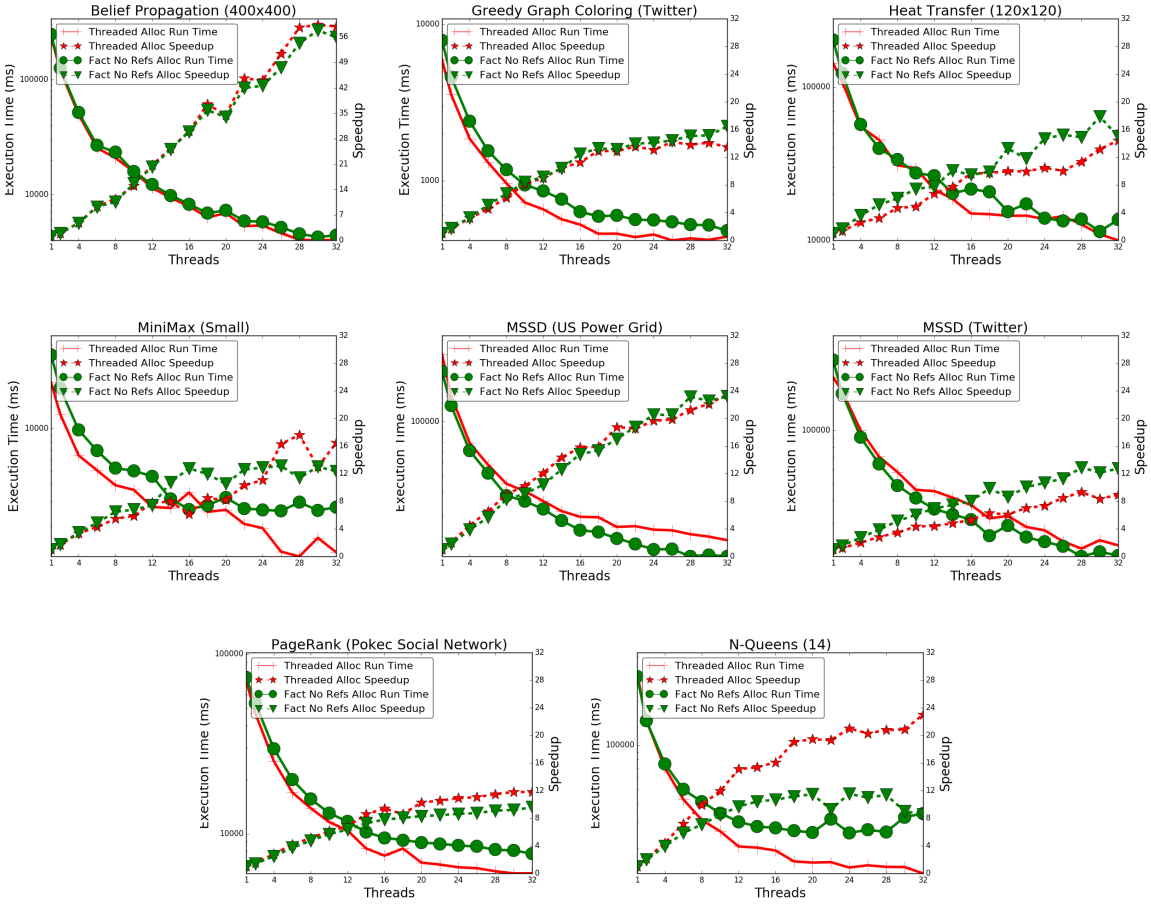


Figure 6.19: *Comparing the threaded allocator described in Section 6.3 against the fact allocator without reference counting.*

formed, which could happen in the case of recursive rules but is avoided when the next iteration only uses new facts derived by the previous iteration.

In the context of the P2 system [LCG+06], which was created for developing declarative networking programs, the previous strategy is not suitable since it is centralized, therefore a new strategy, called *Pipelined Semi-Naïve* (PSN) evaluation, was developed for distributed computation. PSN evaluation evaluates one fact at a time by firing any rule that is derivable using the new fact and older facts. If the new fact is already in the database, then the fact is not used for derivation since it would derive redundant facts.

LM's evaluation strategy is similar to PSN, however, it considers the whole database of facts when firing rules due to the existence of rule priorities. For instance, when a node fires a rule, new facts added for the local node are considered as a whole in order to select the next inference rule. In the case of PSN, each new fact would be considered separately. Still, PSN and LM are both asynchronous strategies which take advantage of the nature of distributed and parallel architectures, respectively. Finally, an important distinction that should be made between PSN and LM is that PSN is for logic programs with only persistent facts, which results in deterministic

results, however because LM uses linear facts, it follows a *don't care* or *committed choice* non-determinism, which may result in different results depending on the order of computations.

6.5.3 CHR implementations

Many basic optimizations used in the LM compiler such as join optimizations and the use of different data structures for indexing facts were inspired in work done on CHR [HdlBSD04]. Wuille et al. [WSD07] have described a CHR to C compiler that follows some of the ideas presented in this chapter and De Koninck et al. [DKSD08] showed how to compile CHR programs with dynamic priorities into Prolog. The novelty of our work focuses on supporting a combination of comprehensions, aggregates and rule priorities.

6.6 Chapter Summary

This chapter provided a full description of the multi core implementation of LM, with a focus on thread management, work stealing and memory allocation. We explained how the virtual machine is organized to provide scalable multi threaded execution and provided experiments on a wide range of problems to demonstrate its applicability and scalability. We also studied the importance of good memory allocators for improved scalability and execution.

Chapter 7

Coordination Facts

In Chapter 6, we saw that an LM program operates on a graph data structure $G = (V, E)$ with nodes V and edges E. Concurrent computation is performed non-deterministically by a set of execution threads T that work on subsets of G. Our implementation allows threads to steal nodes from other threads in order to improve load balancing. However, there are many scheduling details that are left for our implementation to decide and which the programmer has no control over. How should a thread schedule the computation of its sub-graph? Is node stealing beneficial to all programs? What is the best sub-graph partitioning for a given LM program? In order to allow custom node scheduling and partitioning, we introduce *coordination facts*, a mechanism that is indistinguishable from regular computation and that allows the programmer to control how the runtime system executes programs. This is an important first step in allowing the programmer to declaratively control execution. In Chapter 8, we further extend the language with thread-based facts to allow reasoning over the state of the execution threads.

7.1 Motivation

To motivate the need for coordination, we use the Single Source Shortest Path (SSSP) program, a concise program that can take advantage of custom scheduling policies to improve its performance. Figure 7.1 and Figure 7.2 show a possible implementation in LM, and Fig. 7.3 shows an instance of the program on a particular dataset. Note that in Section 6.4.1, we presented the MSSD program which is essentially the SSSP program modified to compute multiple shortest distances.

The SSSP program starts with the declaration of the predicates (lines 1-3). As usual, the `edge` predicate is a persistent predicate that describes the edges of the graph, where the third argument is the weight of each edge. Predicate `shortest` represents a valid distance and path from node `@1` to the node in the first argument. Predicate `relax` also represents a valid path, and, is used to improve the distance in `shortest`. The program computes the shortest distance from node `@1` by improving the distance in the `shortest` facts, until the shortest distance is reached in all nodes. In Fig. 7.2, we declare an example set of initial facts for the program: `edge` facts describe the graph; `shortest(A, +00, [])` is the initial shortest distance (infinity) for all nodes; and `relax(@1, 0, [@1])` starts the algorithm by setting the distance from `@1` to `@1` to be 0.

```
1  type edge(node, node, int).                                    // Predicate declaration
2  type linear shortest(node, int, list int).
3  type linear relax(node, int, list int).
4
5  shortest(A, D1, P1), D1 > D2, relax(A, D2, P2)          // Rule 1: newly improved path
6     -o shortest(A, D2, P2),
7        {B, W | !edge(A, B, W) -o relax(B, D2 + W, P2 ++ [B])}.
8
9  shortest(A, D1, P1), D1 <= D2, relax(A, D2, P2)                // Rule 2: longer path
10    -o shortest(A, D1, P1).
```

Figure 7.1: *Single Source Shortest Path program code.*

```
1  !edge(@1, @2, 3). !edge(@1, @3, 1).
2  !edge(@3, @2, 1). !edge(@3, @4, 5).
3  !edge(@2, @4, 1).
4  shortest(A, +00, []).
5  relax(@1, 0, [@1]).
```

Figure 7.2: *Initial facts for the SSSP program.*

The first rule of the program (lines 5-7) reads as follows: if the current `shortest` path `P1` with distance `D1` is larger than a new path `relax` with distance `D2`, then replace the current shortest path with `D2`, delete the new `relax` path and propagate new paths to the neighbors (comprehension at line 7). The comprehension iterates over the edges of node `A` and derives a new `relax` fact for each node `B` with the distance `D2 + W`, where `W` is the weight of the edge. For example, in Fig. 7.3(a) we apply rule 1 in node `@1` and two new `relax` facts are derived at node `@2` and `@3`. Fig. 7.3(b) is the result after applying the same rule but at node `@2`.

The second rule of the program (lines 9-10) retracts a `relax` fact that has a longer distance than the current shortest distance stored in `shortest`. There are many opportunities for custom scheduling in the SSSP program. For instance, after applying rule 1 in Fig. 7.3(a), it is possible to either apply rules in either node `@2` or node `@3`. This decision depends largely on implementation factors such as node partitioning and number of threads in the system. Still, it is easy to prove that, independent of the particular scheduling used, the final result presented in Fig. 7.3(c) is always achieved.

The SSSP program is concise and declarative but its performance depends on the order in which nodes are executed. If nodes with greater distances are prioritized over other nodes, the program will generate more `relax` facts and it will take longer to reach the shortest distances. From Fig. 7.3, it is clear that the best scheduling is the following: `@1`, `@3`, `@2` and then `@4`, where only 4 `relax` facts are generated. If we had decided to process nodes in order `@1`, `@2`, `@4`, `@3`, `@4`, `@2`, then 6 `relax` facts would have been generated. The optimal solution for SSSP is to schedule the node with the shortest distance, which is essentially the Dijkstra shortest path algorithm [Dij59]. Note how it is possible to change the complexity of the algorithm by simply changing the order of node computation, but still retain the same declarative nature of the program.

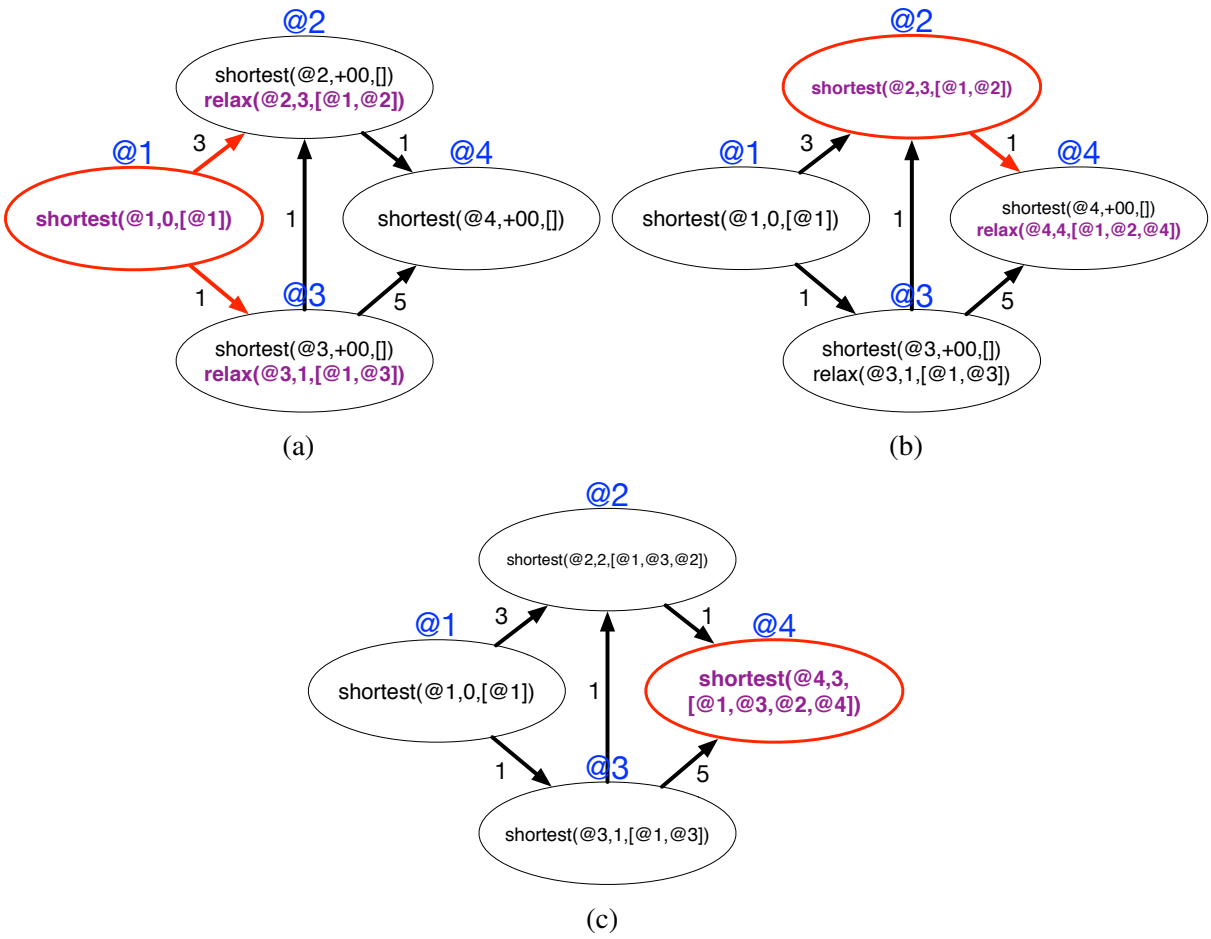


Figure 7.3: *Graphical representation of an example dataset for the SSSP program: (a) represents the program's state after propagating the initial distance at node* `@1`*, followed by (b) where the first rule is applied in node* `@2`*, and (c) represents the final state of the program, where all the shortest paths have been computed.*

7.2 Types of Facts

LM introduces the concept of coordination that allows the programmer to write code that changes how the runtime system schedules and partitions the set of available nodes across the threads of execution. Beyond the distinction between linear and persistent facts, LM further classifies facts into 3 categories: *computation facts*, *structural facts* and *coordination facts*. Predicates are also classified accordingly.

Computation facts are regular facts used to represent the program state. In Fig. 7.1, `relax` and `shortest` are both computation facts.

Structural facts describe information about the connections between the nodes in the graph. In Fig. 7.1, `edge` facts are structural since the corresponding `edge` is used for communication between nodes. Note that structural facts can also be seen as computation facts since they are heavily used in the program's logic.

Coordination facts are classified into *scheduling facts* and *partitioning facts* and allow the programmer to change how the runtime schedules nodes and how it partitions the nodes among threads of execution, respectively. Coordination facts can be used either in the LHS, RHS or both. This allows scheduling and partition decisions to be made based on the state of the program and on the state of the underlying machine. In this fashion, we keep the language declarative because we reason logically about the state of execution, without the need to introduce extra-logical operators into the language which would generate significant difficulties when proving properties about the programs.

Both scheduling and partitioning facts can be further classified into two kinds of facts: *sensing facts* and *action facts*. Sensing facts are used to sense information about the underlying runtime system, such as the placement of nodes in the CPU and scheduling information.[1]

Action facts are used to apply coordination operations on the runtime system. Action facts are linear facts which are consumed when the corresponding action is performed.[2] We use them to change the order in which nodes are evaluated in the runtime system and to make partitioning decisions (assign nodes to threads). It is possible to give hints to the virtual machine in order to prioritize the computation of some nodes.

With sensing and action facts, we can write rules that sense the state of the runtime system and then apply decisions in order to improve execution speed or change partitioning information. In most situations, this set of rules can be added to the program without modifying the meaning of the original rules. When using sensing facts in rules, the meaning of the programs will change and the programmer needs to be aware of the semantics of coordination and how the virtual machine transitions from one state to another using built-in linear logic rules (see Section 8.4 for a detailed discussion).

[1] In the original Meld [ARLG+09], sensing facts were used to get information about the outside world, like temperature, touch data, neighborhood status, etc.

[2] Like sensing facts, action facts were also introduced in the original Meld and were used to make the robots perform actions in the outside world (e.g., moving, changing speed).

7.3 Scheduling Facts

In order to allow different scheduling strategies, we introduce the concept of *node priority* by assigning a priority value to every node in the program and by introducing coordination facts that manipulate the priority values. By default, nodes have the same priority value and thus can be picked in any order. In our implementation, we use a FIFO approach because older nodes tend to have a higher number of unexamined facts, from which they can derive subsequent new facts.

We have two kinds of priorities: a *temporary priority* and a *default priority*. When a temporary priority exists, it supersedes the default priority. Once a node is processed and becomes **inactive**, the temporary priority is removed and the default priority is used. In a nutshell, the *priority* of a node is equal to the temporary priority if there is one or it is the default priority otherwise. Initially, all nodes have a default priority of $-\infty$.

The following list presents the action facts available to manipulate the scheduling decisions of the system:

- `update-priority(node A, float F)`: Changes the temporary priority of node `A` to `F`.
- `set-priority(node A, float F)`: Sets the temporary priority of node `A` to `F` if `F` is *better* than the current priority (either default or temporary). The programmer can decide (see 7.5) if priorities are to be ordered in ascending or descending order, thus if node `A` has priority `G`, we only change the temporary priority to `F` if `F > G` (ascending order) or `F < G` (descending order).
- `add-priority(node A, float F)`: Changes the temporary priority of node `A` to `F + G`, where `G` is the priority of the node. A negative priority is also allowed.
- `remove-priority(node A)`: Removes the temporary priority from node `A`.
- `schedule-next(node A)`: Changes the temporary priority of node `A` to be $+\infty$ if priorities are descending, or $-\infty$ if priorities are ascending.
- `set-default-priority(node A, float F)`: Sets the default priority of node `A` to `F`.
- `stop-program()`: Immediately stops the execution of the whole program.

LM also provides two sensing facts:

- `priority(node A, float P)`: Represents the priority of node `A`. Value `P` is equal to the temporary priority or equal to the default priority if `A` has no temporary priority assigned to it.
- `default-priority(node A, float P)`: Value `P` represents the default priority of node `A`.

Sensing facts can only be used in the LHS of rules and are exempt from the constraint that forces every fact used in the body to have the same first argument. Note that when sensing facts are used to prove new facts, they must be re-derived so that the coordination model remains consistent. Furthermore, when the programmer uses action facts such as `set-priority` and `set-default-priority`, the runtime system implicitly updates and re-derives the associated sensing facts, without any programmer interaction.

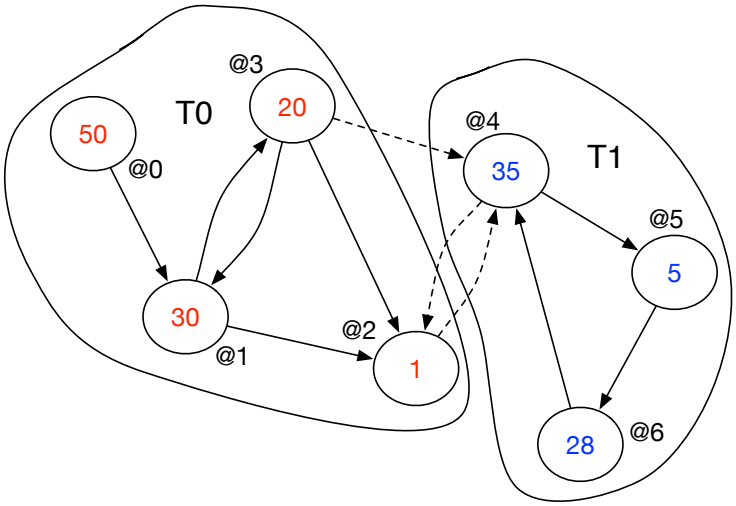


Figure 7.4: *Priorities with sub-graph partitioning. Priorities are used on a per-thread basis therefore thread* `T0` *schedules* `@0` *to execute, while* `T1` *schedules node* `@4`.

The priorities assigned to nodes follow a partial ordering since each thread selects the highest priority node from its sub-graph and not from the whole graph. Figure 7.4 shows an example of a graph being processed by two threads, `T0` and `T1`. The order for `T0` will be `@0`, `@1`, `@3`, `@2` and for thread `T1` it will be `@4`, `@6`, `@5`. Note that priorities of nodes can be set from any node in the graph, even if those nodes live on different threads. Of course, this requires communication between threads.

7.4 Partitioning Facts

We provide several coordination facts for dealing with node partitioning among the running threads. Since each node is placed in some thread, the partitioning facts revolve around thread placement. In terms of action facts, we have the following:

- `set-thread(node A, thread T)`: Places node `A` in thread `T` until the program terminates or a `set-moving(A)` fact is derived.
- `set-affinity(node A, node B)`: Places node `B` in the thread of node `A` until the program terminates or a `set-moving(B)` fact is derived.
- `set-moving(node A)`: Allows node `A` to move freely between threads.

As an example of `set-thread`, consider again the graph in Fig. 7.4. If a coordination fact `set-thread(@2, T1)` is derived, then node `@2` will become part of the sub-graph of thread `T1`. The result is shown in Fig. 7.5.

Sensing facts retrieve information about node placement and are specified as follows:

- `thread-id(node A, thread T)`: Linear fact that maps node `A` to thread `T` which `A` belongs to. Fact `set-thread` implicitly updates fact `thread-id`.
- `is-static(node A)`: Fact available at node `A` if `A` is not allowed to move between threads.

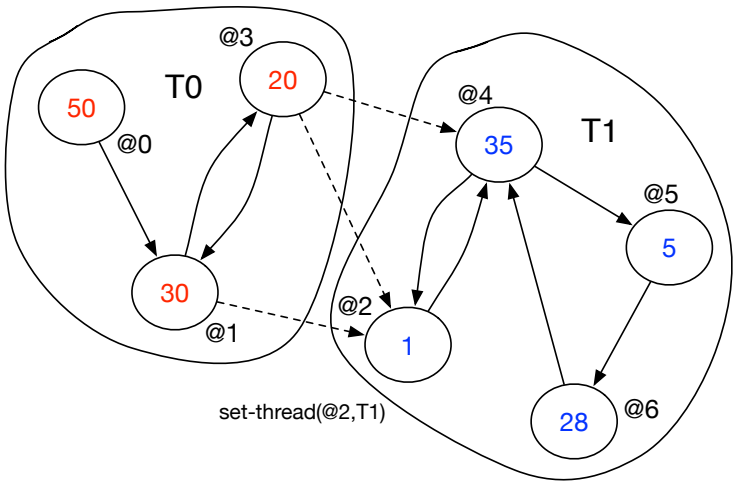


Figure 7.5: *Moving node* `@2` *to thread* `T1` *using* `set-thread(@2, T1)`.

- `is-moving(node A)`: Fact available at node `A` if `A` is allowed to move between threads.
- `just-moved(node A)`: Linear fact derived by the `set-thread` action if, at that moment, the node `A` is running on the target thread.

7.5 Global Directives

We also provide a few global coordination statements that cannot be specified as sensing or action facts but are still important:

- `priority @order ORDER` (`ORDER` can be either `asc` or `desc`): Defines if priorities are to be selected by the smallest or the greatest value, respectively.
- `priority @default P`: Informs the runtime system that all nodes must start with default priority `P`. Alternatively, the programmer can define a `set-default-priority(A, P)` initial fact.
- `priority @base P`: Informs the runtime system that the *base priority* should be `P`. The base priority is, by default, 0 when the priority ordering is `desc` and $+\infty$ when the ordering is `asc` and represents the smallest priority value possible.
- `priority @initial P`: Informs the runtime system that all nodes must start with temporary priority `P`. Alternatively, the programmer can define a `set-priority(A, P)` fact. By default, the initial priority value is equal to $+\infty$ (for `desc`) or 0 (for `asc`).

7.6 Implementation

In order to support priorities and node partitioning, we transformed the singly linked list work queue of each thread, as presented in Fig. 6.3, into two pairs of queues: two *standard queues* implemented as doubly linked lists and two *priority queues* implemented as min/max heaps.

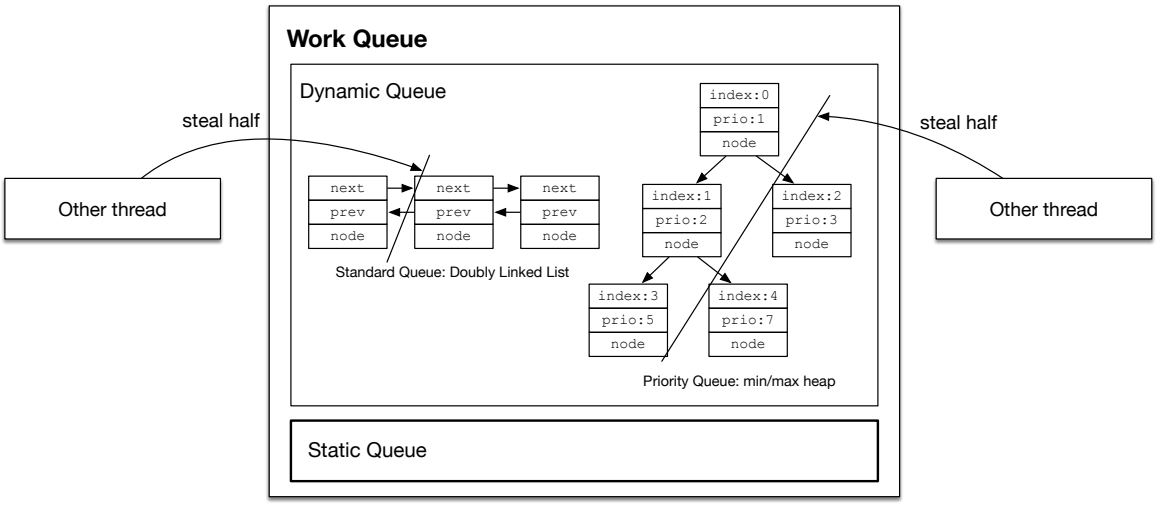


Figure 7.6: *Thread's work queue and its interaction with other threads: the dynamic queue contains nodes that can be stolen while the static queue contains nodes that cannot be stolen. Both queues are implemented with one standard queue and one priority queue.*

These four queues are decomposed into the *static queue* and the *dynamic queue*, where each queue has a standard and a priority queue. The static queue contains nodes that cannot be stolen and the *dynamic queue* contains nodes which can be stolen by other threads.

The standard queue contains nodes without priorities (i.e., nodes with base priorities) and supports push into tail, remove node from the head, remove arbitrary node, and remove first half of nodes. The priority queue orders contains nodes with priorities (ordered by the `priority` sensing fact) and is implemented as a binary heap array. It supports the following operations: push into the heap, remove the *min* node, remove an arbitrary node, remove half of the nodes (vertical split), and priority update. Operations for removing half of the queue are implemented in order to support node stealing, while operations to remove arbitrary nodes or update priority allows threads to change the priority of nodes. In the regular queue, the first $n/2$ elements are removed from the front of the list and, in the priority queue, the binary heap is split vertically for improved distribution of priorities.

Figure 7.6 presents an overview of the dynamic queue and how the remove half operations are implemented in the sub-queues in order to support node stealing. Note that the static queue is not represented but it also includes the same two sub-queues. Table 7.1 shows the complexity of queue operations and compares the standard queue against the priority queue. Except for the remove half operation, priority queue operations are more expensive.

The `next` and `prev` pointers of the standard queue are part of the node structure in order to save space. These pointers are also used in the priority queue but for storing the current priority and the node's index in the binary heap array.

Note that, in order to minimize inter-thread communication, node priorities are implemented at the thread level, i.e., when a thread picks the highest priority node from the priority queue, it picks the highest priority node from its subgraph of nodes and not the highest priority node from the whole graph.

Operation	Standard queue	Priority Queue
Push	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$
Pop	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$
Remove	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$
Remove Half	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$
Priority Update	-	$\mathcal{O}(\log N)$

Table 7.1: *Complexity of queue operations for both the standard queue and the priority queue.*

7.6.1 Node State Machine

To accommodate the new coordination facilities, we added a new node state to the state machine presented previously in Fig. 5.2. Figure 7.7 reviews the new set of states of the state machine.

- **running**: The node is deriving facts.
- **inactive**: The node is inactive, i.e., it has no new facts and it is not in any queue for processing.
- **active**: The node has new facts and it is in some queue waiting to be processed.
- **stealing**: The node has just been stolen and it is in the process of being moved to another thread.
- **coordinating**: The node moves from one queue to another or inside the priority queue when changing its priority.

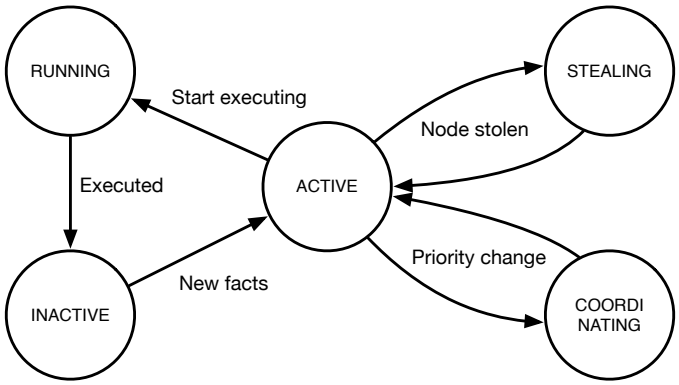


Figure 7.7: *Node state machine extended with the new **coordinating** state.*

7.6.2 Coordination Instructions

Coordination facts are implemented as API calls to the virtual machine which implement the appropriate operations. Sensing facts read information about the state of the virtual machine while action facts lock and update the appropriate underlying data structures such as the node data structure or the thread's queues.

When a thread T_1 performs coordination operations to a node owned by a thread T_2, it needs to synchronize with T_2 and, for that, the *State Lock* in the target node needs to be acquired. Optionally, we may need to also acquire the locks of T_2's queues if the priority is being updated. Note that both the standard queue and priority queue have separate locks in order to allow concurrent manipulation of the two data structures.

For the coordination facts `set-priority` and `add-priority`, when multiple operations are directed to the same node during a single node execution, we coalesce the multiple operations into a single operation. As an example, if a node derives two `set-priority` facts that change the priority of node `@1` to `1` and `2`, the virtual machine coalesces the two instructions into a single `set-priority` that is applied with value `2` (the higher priority) after all the candidate rules of the node are executed. The reason for this optimization is that nodes may run for some number of steps during the lifetime of the program, therefore immediately applying each and every coordination instruction is not cost effective. We aim to reduce the amount of locking and movement of nodes inside the queues due to priority updates.

7.7 Coordinating SSSP

Now that we have presented the coordination facts for LM, we present an example of their usage in Fig. 7.8 by extending the SSSP program presented before. The coordinated version of SSSP uses a global program directive to order priorities in ascending order (line 5) and the coordination fact `set-priority` (line 11). The proof of correctness for this program is presented in Appendix G.1.

```
1  type edge(node, node, int).                                  // Predicate declaration
2  type linear shortest(node, int, list int).
3  type linear relax(node, int, list int).
4
5  priority @order asc.
6
7  shortest(A, D1, P1), D1 > D2, relax(A, D2, P2)          // Rule 1: newly improved path
8     -o shortest(A, D2, P2),
9        {B, W | !edge(A, B, W) -o
10          relax(B, D2 + W, P2 ++ [B]),
11          set-priority(B, float(D2 + W))}.
12
13 shortest(A, D1, P1), D1 <= D2, relax(A, D2, P2)               // Rule 2: longer path
14    -o shortest(A, D1, P1).
15
16 shortest(A, +00, []).                                              // Initial facts
17 relax(@1, 0, [@1]).
```

Figure 7.8: *Shortest Path Program taking advantage of the* `set-priority` *coordination fact.*

Figure 7.9 presents the 4 steps of computation for the new SSSP program when executed with 1 thread. In every step, a new shortest path is computed at a different node, starting from the shorter paths up to the longer paths. This is exactly the same behavior as the Dijkstra's

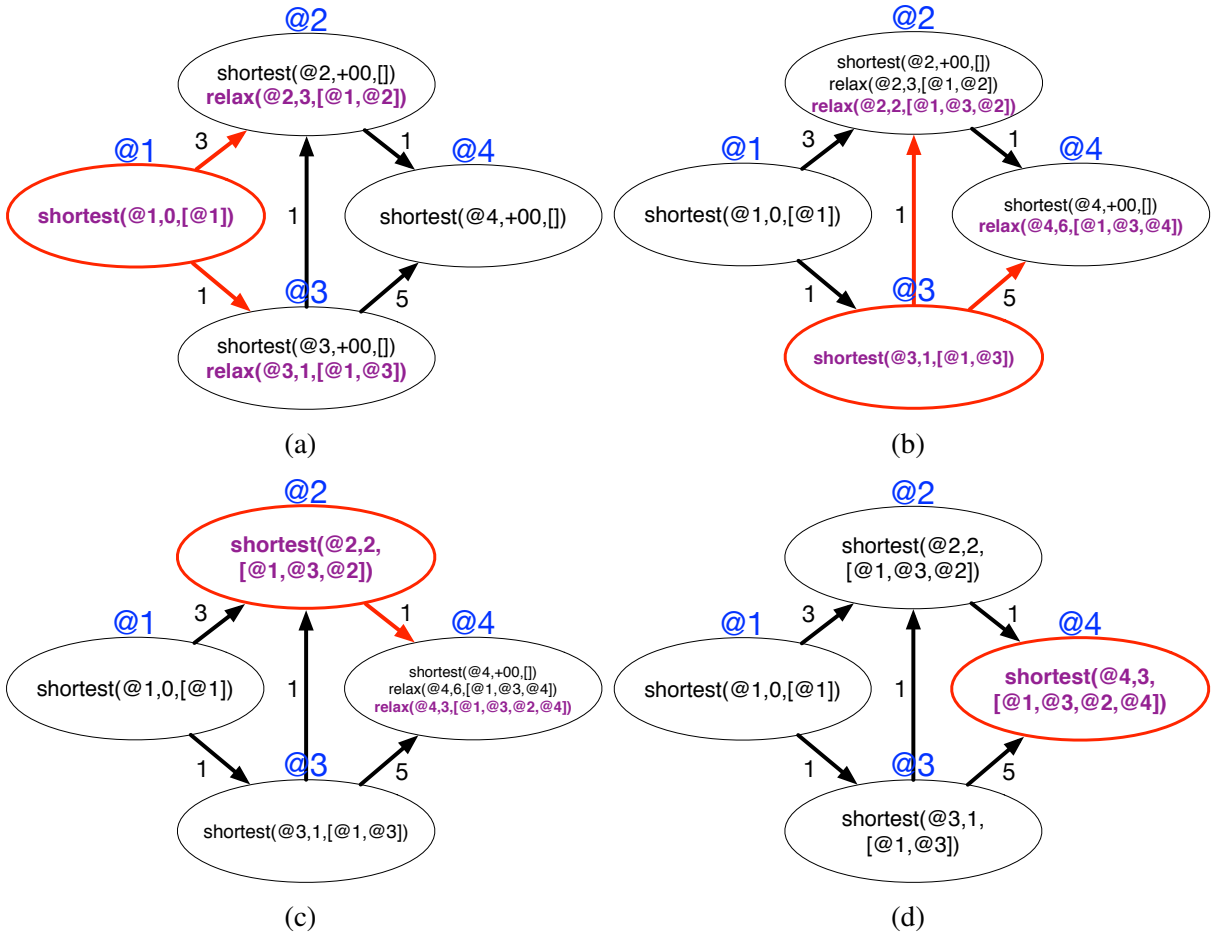


Figure 7.9: *Graphical representation of the new SSSP program in Fig. 7.8. (a) represents the program after propagating initial distance at node* `@1`*, followed by (b) where the first rule is applied in node* `@3`*. (c) represents the result after retracting all the* `relax` *facts at node* `@2` *and (d) is the final state of the program where all the shortest paths have been computed.*

algorithm [Dij59]. For multiple threads, the scheduling may not be optimal since threads pick the node with the shortest distance from their subset of nodes. However, we have increased parallelism (no global synchronization) and threads are still able to locally avoid unnecessary work.

In order to experiment with the coordinated SSSP program, we extend it to support the computation of shortest distances from multiple sources. This modified version, named MSSD, was already used in Section 6.4.1 for measuring the performance of LM's virtual machine. MSSD is not only more interesting than SSSP but also allows us to better understand the advantages of using coordination.

Table 7.2 presents fact statistics of the regular and coordinated versions of MSSD when executed using 1 thread. We gathered the number of facts derived, number of facts deleted, number of facts sent (to other nodes) and the final number of facts. Note that the number of facts sent is a subset of the number of facts derived, while the final number of facts is the number of

facts derived minus the number of facts deleted plus the number of initial facts. As expected, there is a reduction in the number of facts derived in all datasets. There is also a clear correlation between the reduction in facts processed and the run time reduction achieved using coordination.

Dataset	# Derived		# Sent		# Deleted		# Final
US 500 Airports	2.9M	83%	2.9M	83%	2.7M	82%	164.3K
OCLinks	64.9M	81%	64.9M	81%	62.8M	80%	2.2M
EU Email	35.4M	67%	34.9M	67%	32.7M	61%	3.8M
Twitter	357.5M	20%	357.4M	20%	354.7M	19%	4.5M
US Power Grid	207.6M	65%	207.6M	65%	183.2M	60%	24.4M
Live Journal	776.5M	21%	765.9M	20%	768.6M	20%	71.5M
Orkut	251.3M	40%	247.9M	39%	248.4M	39%	100.6M

Table 7.2: *Fact statistics for the MSSD program when run on 1 thread. For each dataset, we gathered the number of facts derived, number of facts deleted, number of facts sent to other nodes and total number of facts in the database when the program terminates. Columns* ***# Derived***, ***# Sent*** *and* ***# Deleted*** *show the number of facts for the regular version (first column) and then the ratio of facts for the coordinated version over the regular version. Percentage values less than 100% mean that the coordinated version produces fewer facts.*

Figure 7.10 shows the comparison between the regular version (without coordination) and the coordinated version of the MSSD program. We use the datasets already described in Section 6.4.1. In each plot we present the following lines: **Regular**, which represents the run time of the regular version; **Coordinated**, which represents the run time of the coordinated version; **Coordinated(1)/Regular(t)**, which represents the speedup of the regular version using the single threaded coordinated version as the baseline; **Coordinated(1)/Coordinated(t)**, which represents the speedup of the coordinated version against the run time of the coordination version using 1 thread as the baseline; **C++**, which represents the run time of the MSSD C++ program already presented in Section 6.4.1; **Ligra**, which represents the run time of the MSSD program written in the Ligra framework [SB13]. Ligra's program computes the multiple shortest distance computations using the BellmanFord program provided by the source distribution[3], but modified to compute the distance from a single source node separately. Note that the run time (vertical axis) axis uses a logarithmic scale.

Overall, there is a clear improvement in almost every dataset used. The datasets that see the best run time reductions are Twitter, Orkut and LiveJournal, which are the datasets where the distance is calculated for fewer nodes. When calculating the distance from multiple sources, the `set-priority` fact is less effective because, although it selects the node with the shortest distance to some source node, other distance computations (from other source nodes) are also propagated to neighbor nodes, which may not be optimal. However, a potential solution to this problem is to add new coordination facts and then change the underlying coordination machinery to support fact-based priorities (instead of node based). This shows the beauty of our coordination model since this would not require changes to the semantics of the language. Furthermore,

[3]Modified version of the program is available at `http://github.com/flavioc/ligra`

the shortest distance algorithm would still remain declarative and correct due to the invariants that are always held when applying the program's rules.

There is another interesting trend in our results which relates the improvement offered by the coordination version over the regular version and the number of threads used. Even if the number of threads goes up, the coordinated version is still able to keep about the same run time reduction ratio over the regular version. This means that even if higher priority nodes are picked from the each thread's subgraph, the coordinated version is still able to reduce run time. In other words, optimal (global) node scheduling is not required to see good results.

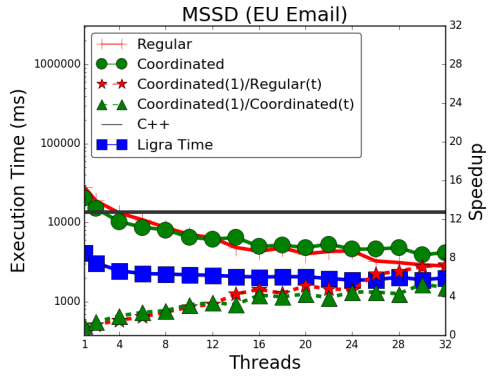
Ligra's version of the MSSD program performs badly when using multiple source nodes (datasets EU Email, OCLinks and US Power Grid). We found Ligra primitives (`edgeMap` and `vertexMap`) unsuitable for simultaneous computation of shortest distances. It would have been better to drop Ligra and use a task-parallel approach, where tasks, represented as source nodes, are assigned to threads and each task would use a sequential shortest distance algorithm. LM shines in such cases since it is a declarative language that can be used to solve a wider range of problems. Ligra beats LM in the Twitter, Orkut and LiveJournal datasets, where Ligra is, on average, 9, 3, and 10 times faster than LM, respectively. In terms of scalability, LM has similar scalability in the Twitter and Orkut datasets, but worse scalability in the LiveJournal dataset. It should be also noted that the datasets EU Email and OCLinks are too small for Ligra, therefore it is hard to make a scalability comparison for those datasets.

In Section 7.6.2, we described the coalescing optimization where priority action facts are coalesced into single operations. The MSSD program computes the shortest distances to multiple nodes and thus it can take advantage of this optimization. Consider a MSSD program that is computing the shortest distances from node `@1` and `@2`. If a node `@3` needs to propagate the shortest distance d_1 to another node `@4` (distance from `@1`) and a distance d_2 to node `@4` (distance from `@2`), then it would need to change the priority twice, first to d_1, and then to d_2. With the coalescing optimization, it only changes the priority to the best of both. It is then advantageous to only apply a single coordination operation in order to reduce queue operations and overall locking.

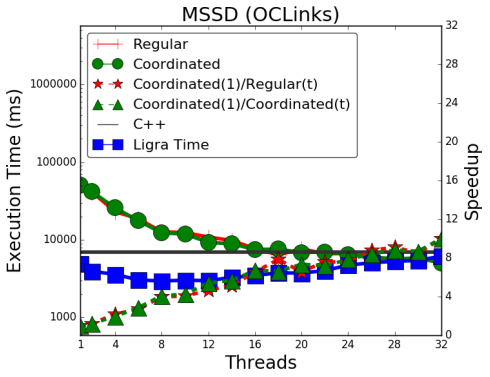
In order to observe the effects of the coalescing optimization, we disabled its support from the VM and then we ran the same experiments where priority updates are done immediately. The results are presented in Fig. 7.11 and are represented using the **UnOptimized** label. From the results, we can see that some datasets, such as EU Email and US Power Grid, maintain their overall scalability, especially the speedup (**Coordinated(1)/Coordinated(t)**) for 32 threads (**t = 32**). For the OCLinks, Twitter and Live Journal datasets, there is some performance degradation since their overall scalability drops without the optimization. This shows that coordination coalescing is an effective optimization for improving the run time of coordinated programs.

7.8 Applications

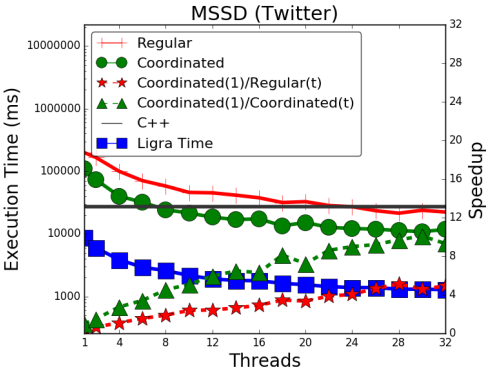
To further understand how coordination facts work, we present other programs that take advantage of them.



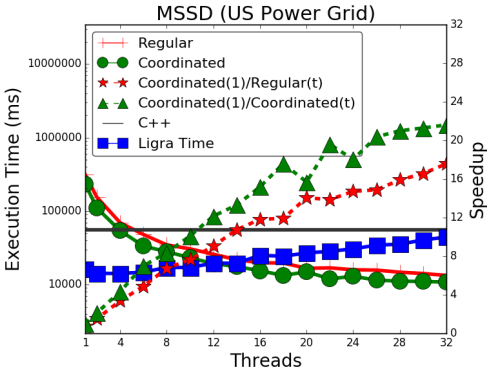
(a) *Graph with 265000 nodes and 420000 edges. The shortest distance is calculated for 100 nodes.*



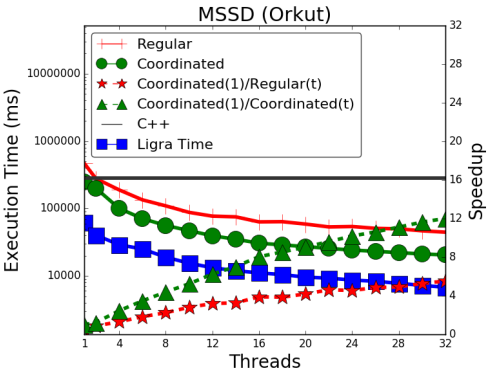
(b) *Graph with around 2000 nodes and 20000 edges. The shortest distance is calculated for all nodes.*



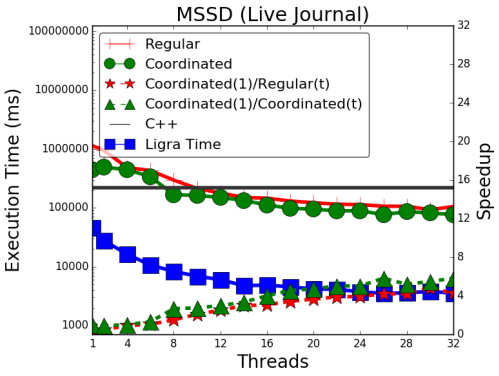
(c) *Graph with 81306 nodes and 1768149 edges. The shortest distance is calculated for 40 nodes.*



(d) *Graph with around 5000 nodes and 13000 edges. The shortest distance is calculated for all nodes.*

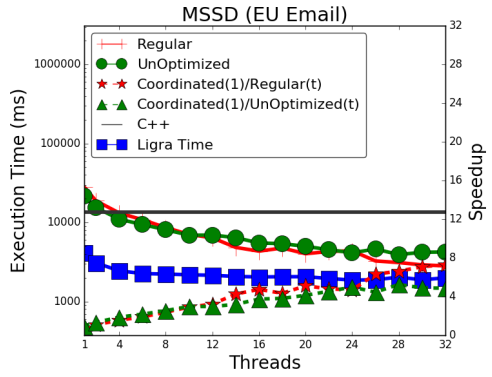


(e) *Graph with 3072441 nodes and 117185083 edges. The shortest distance is calculated for two nodes.*

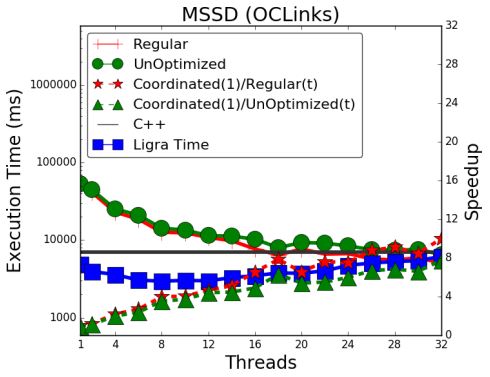


(f) *Graph with around 4847571 nodes and 68993773 edges. The shortest distance is calculated for two nodes.*

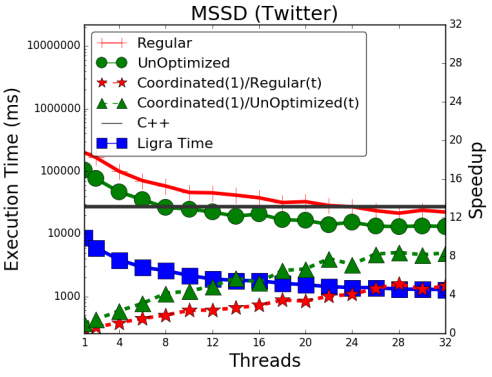
Figure 7.10: *Scalability comparison for the MSSD program when enabling coordination facts.*



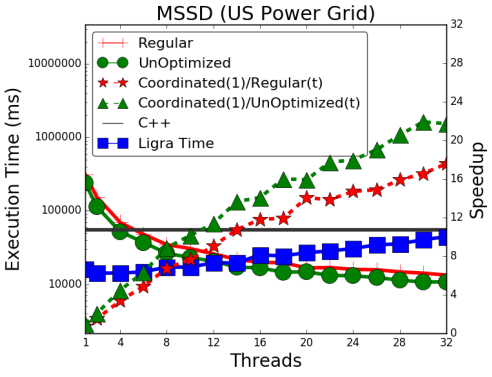
(a) *Graph with 265000 nodes and 420000 edges. The shortest distance is calculated for 100 nodes.*



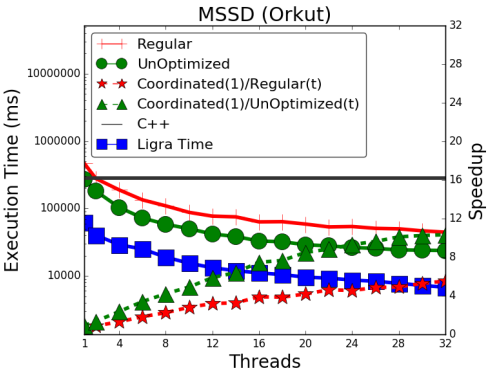
(b) *Graph with around 2000 nodes and 20000 edges. The shortest distance is calculated for all nodes.*



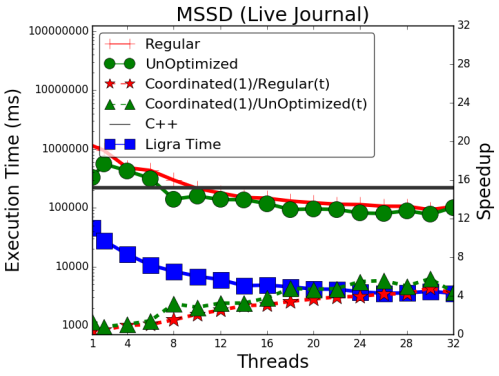
(c) *Graph with 81306 nodes and 1768149 edges. The shortest distance is calculated for 40 nodes.*



(d) *Graph with around 5000 nodes and 13000 edges. The shortest distance is calculated for all nodes.*



(e) *Graph with 3072441 nodes and 117185083 edges. The shortest distance is calculated for two nodes.*



(f) *Graph with around 4847571 nodes and 68993773 edges. The shortest distance is calculated for two nodes.*

Figure 7.11: *Scalability for the MSSD program when not coalescing coordination directives.*

7.8.1 MiniMax

The MiniMax algorithm is a decision rule algorithm for minimizing the possible loss for a worst case (maximum loss) scenario in a zero sum game for 2 (or more) players who play in turns.

The algorithm builds a game tree, where each tree node represents a game state and the children represent the possible game moves that can be made by either player 1 or player 2. An evaluation function is used to compute the score of the game for each leaf of the tree. A node is a leaf when the game state can no longer be expanded. Finally, the algorithm recursively minimizes or maximizes the scores of each node. To select the best move for player 1, the algorithm picks the move maximized at the root node.

Figure 7.12 shows LM's code for the MiniMax program using coordination facts. The following predicates are specified: `parent` connects nodes to represent a tree; `score(A, Score, Play)` and `new-score(A, Score, Play)` represent the score of a node, along with the game move in the `Play` argument; `minimize` and `maximize` are counters that either maximize or minimize the scores coming from the children nodes; `expand` expands and creates children in the current node; and `play` starts the MiniMax expansion and computation at the current node.

The LM program starts with a root node (with the initial game state) that is expanded with the available moves at each level. The graph of the program is dynamic since nodes are created and then deleted once they are no longer needed. The latter happens when the leaf scores are computed or when a node fully minimizes or maximizes the children scores. When the program ends, only the root node has facts in its database.

The first two rules in lines 14-20 check if the current game is final, namely, if a player has won or the game drew: the first rule generates the score for the final state while the second expands the game state by generating all the possible plays for player `NextPlayer`.

The expansion rules create the children for the current node and are implemented in lines 22-40. The first two rules create either a `maximize` or `minimize` fact that will either maximize or minimize the scores of the children nodes. The third and fourth expansion rules simulate a player move and, for that, create a new node `B` using the `exists` language construct. We link `B` with `A` using `parent(B, A)` and kickstart the recursive expansion of node `B` by deriving a `play` fact. Finally, the rule in lines 39-40 is for the case when the current player cannot play in the current game position.

At the end of execution, there should be a single fact, namely, `score(@1, Score, Play)`, where `Score` is the MiniMax score for the best play `Play` that can be made by the main player for the initial game state. Note that the full proof of correctness of the MiniMax program is presented in Appendix G.2.

As noted in Section 7.3, LM's default scheduler uses a FIFO approach, which results in a breadth-first expansion of the MiniMax tree. This results in $\mathcal{O}(n)$ space complexity, where n is the number of nodes in the tree, since the tree must be fully expanded before the scores at the leaves are actually computed. With coordination, we set the priority of a node to be its depth (lines 29 and 30) so that the tree is expanded in a depth-first fashion, leading to $\mathcal{O}(dt)$ memory complexity, where d is the depth of the tree and t is the number of threads. Since threads prioritize deeper nodes, the scores of the first leaves are immediately computed and then sent to the parent node. At this point, the leaves are deleted and reused for other nodes in the tree, resulting in minimal memory usage. As an example, consider a system with 2 threads, T_1 and T_2, where T_1

```
type list int game.                                                                    // Type declaration
type linear parent(node, node).                                                      // Predicate declaration
type linear score(node, int, int).
type linear new-score(node, int, int).
type linear minimize(node, int, int, int).
type linear maximize(node, int, int, int).
type linear expand(node, game FirstPart, game SecondPart, int Descendants, int Player, int Play, int Depth).
type linear play(node, game Game, int Player, int Play, int Depth).

const root-player = 1. const initial-game = [...].    // Constant declaration: player and initial game state
fun next(P : int) : int = if P = 1 then 2 else 1 end.            // Function declaration: select next player
external fun minimax_score(game, int, int).                                  // Returns the score of a board

play(A, Game, NextPlayer, LastPlay, Depth),                                        // Rule 1: ending game state
Score = minimax_score(Game, NextPlayer, root-player), Score > 0
   -o score(A, Score, LastPlay).

play(A, Game, NextPlayer, LastPlay, Depth),                                         // Rule 2: expand state
0 = minimax_score(Game, NextPlayer, root-player)                    // minimax_score is an external function
   -o expand(A, [], Game, 0, NextPlayer, LastPlay, Depth).

expand(A, Game, [], N, Player, Play, Depth), Player = root-player                   // Rule 3: maximize node
   -o maximize(A, N, -00, 0).

expand(A, Game, [], N, Player, Play, Depth), Player <> root-player                  // Rule 4: minimize node
   -o minimize(A, N, +00, 0).

expand(A, First, [0 | Xs], N, Player, Play, Depth), Depth >= 5           // Rule 5: create static child node
   -o exists B. (set-affinity(A, B),
       set-default-priority(B, float(Depth + 1)),
       play(B, Game ++ [P | Xs], next(P), length(First), Depth + 1), parent(B, A).
       expand(A, First ++ [0], Xs, N + 1, Player, Play, Depth)).

expand(A, First, [0 | Xs], N, Player, Play, Depth), Depth < 5                   // Rule 6: create child node
  -o exists B. (set-default-priority(B, float(Depth + 1)),
       play(B, Game ++ [P | Xs], next(P), length(First), Depth + 1), parent(B, A),
       expand(A, First ++ [0], Xs, N + 1, Player, Play, Depth)).

expand(A, First, [C | Xs], N, Player, Play, Depth), C <> 0                        // Rule 7: next game play
  -o expand(A, First ++ [C], Xs, N, Player, Play, Depth).

score(A, Score, BestPlay), parent(A, B) -o new-score(B, Score, BestPlay).// Rule 8: sending score to parent node

new-score(A, Score, Play), minimize(A, N, Current, BestPlay), Current > Score  // Rule 9: keep current score
   -o minimize(A, N - 1, Score, Play).

new-score(A, Score, Play), minimize(A, N, Current, BestPlay), Current <= Score   // Rule 10: select new best
   -o minimize(A, N - 1, Current, BestPlay).

minimize(A, 0, Score, BestPlay) -o score(A, Score, BestPlay).// Rule 11: score minimized

new-score(A, Score, Play), maximize(A, N, Current, BestPlay), Current < Score // Rule 12: keep current score
   -o maximize(A, N - 1, Score, Play).

new-score(A, Score, Play), minimize(A, N, Current, BestPlay), Current >= Score   // Rule 13: select new best
   -o maximize(A, N - 1, Current, BestPlay).

maximize(A, 0, Score, BestPlay) -o score(A, Score, BestPlay).                    // Rule 14: score maximized

play(@0, initial-game, root-player, 0, 1).                                                 // Initial fact
```

Figure 7.12: *LM code for the MiniMax program.*

first expands the root node and then expands the first child. Since T_2 is idle, it steals half of the root's children nodes and starts expanding one of the nodes in a depth-first fashion.

We also take advantage of memory locality by using `set-affinity` (line 30), so that nodes after a certain level are not stolen by other threads. While this is not critical for performance in shared memory systems where node stealing is efficient, we expect that such coordination to be critical in distributed systems.

The rest of the program contains rules for maximizing and minimizing scores (lines 44-58), through the retraction of `new-score` incoming facts.

Dataset	**Threads**	**Average**		**Final**		**# Malloc**	
		Regular	Coord	Regular	Coord	Regular	Coord
Small	1	1667.9MB	52KB	30KB	31KB	42	11
	2	816.4MB	48KB	60KB	61KB	79	22
	4	375.3MB	45KB	119KB	121KB	149	44
	8	189.7MB	43KB	236KB	240KB	281	88
	16	81.3MB	40KB	472KB	478KB	531	172
	24	26.6MB	39KB	707KB	712KB	734	252
	32	24.4MB	37KB	937KB	953KB	932	332
Big	1	13.5GB	62KB	30KB	31KB	48	12
	2	6.7GB	54KB	60KB	61KB	92	23
	4	2.4GB	52KB	119KB	121KB	171	44
	8	1545.6MB	50KB	236KB	240KB	331	88
	16	624.2MB	47KB	472KB	479KB	613	177
	24	400.5MB	45KB	707KB	717KB	886	265
	32	222.7MB	44KB	942KB	955KB	1145	353

Table 7.3: *Comparing the memory usage of the regular and coordinated MiniMax programs.*

In Table 7.3 we compare the memory usage of the coordinated MiniMax program against the regular MiniMax program. The **Average** column presents the average memory usage during the lifetime of the program, the **Final** column indicates the amount of memory used at the end of the program, while the **# Malloc** column represents the number of calls made to the `malloc` function by the threaded allocator. The coordinated version uses significantly less memory than the regular version. Both the Big and the Small dataset use about the same memory when using coordination, which is an excellent result since the Big dataset is ten times larger than the Small dataset. As the number of the threads increases, the average memory usage of the regular version decreases because there are more threads that are able to complete different parts of the tree at the same time. The same happens for the coordinated version, but at a much smaller scale since threads are exploring the tree in a depth-first fashion. Finally, as expected, there is also a large reduction in the number of `malloc`'s called by the threaded allocator in the coordinated version.

The run time and scalability results are presented in Fig. 7.13. There is a clear performance improvement when coordinating the MiniMax program due to the low memory usage. The Big dataset needs only 5 threads to reach the performance of the sequential C++ program and, when using 32 threads, the LM program is more than four times faster than the C++ version. The use

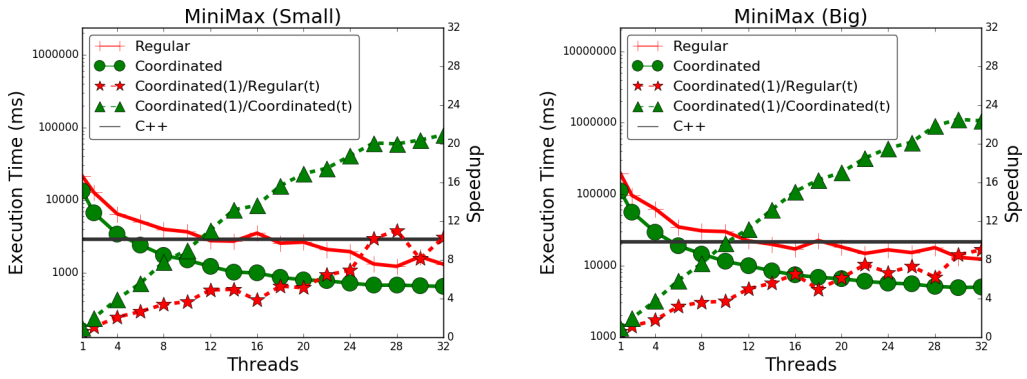


Figure 7.13: *Scalability for the MiniMax program when using coordination.*

of coordination facts allows us to improve the execution of this declarative program and make it run significantly faster than it would be otherwise.

7.8.2 N-Queens

The N-Queens puzzle is the problem of placing N chess queens on an NxN chessboard so that no pair of queens attack each other [HLM69]. The specific challenge of finding all the distinct solutions to this problem is a good benchmark in designing parallel algorithms. Most popular parallel implementations of the N-Queens problem distribute the search space of the problem by assigning incomplete boards as tasks to threads. Our approach is unusual since we see the squares of the board as the LM's nodes of the program, where each square can communicate with other 4 squares: the adjacent right and the adjacent left on the same row, and the first non-diagonal square to the right and to the left on the row below. To represent a partial/valid board state, we use a list of integers, where each integers represents the column in which a queen is placed. For example $[2, 0]$ means that a queen is placed in square $(0, 0)$ and another in square $(1, 2)$. At any given time, many partial board states may use the same squares and each square can belong to several board states. An example final database for the 8-Queens problem is presented in Fig. 7.16, where only 8 valid states of the possible 92 states are shown.

Figure 7.15 presents our solution. The predicates `left`, `right`, `down-left` and `down-right` represent each square's four connections to the squares on the same row and to the row below. Predicate `coord` represents the coordinate of the given square. Predicates `propagate-left` and `propagate-right` propagate a given board state to the left or to the right in the current row of the board. Predicates `test-y`, `test-diag-left` and `test-diag-right` take one valid board state (last argument) and then determine if the current square is able to add itself to that state by checking the three restrictions of the N-Queens puzzle. Predicate `new-state` represent a new valid board state in the current square, while `send-down` forces a square to send a new state to the squares in the row below. Finally, predicate `final-state` represents a full board state. Once the program completes, the last row of squares of the board should have several `final-state` facts which represent all the distinct solutions to the puzzle.

In terms of facts and rules, we have an initial fact that represents an empty state (line 51),

which is instantiated in the top-left square and must be propagated to all squares in the same row (rule in lines 16-17). If a square S receives a new state L, it checks if S can be incorporated into L. For that, it checks if there is no queen on S's column (rules in lines 21-27), if there is no queen on S's left diagonal (rules in lines 29-35) and if there is no queen on S's right diagonal (rules in lines 37-43).

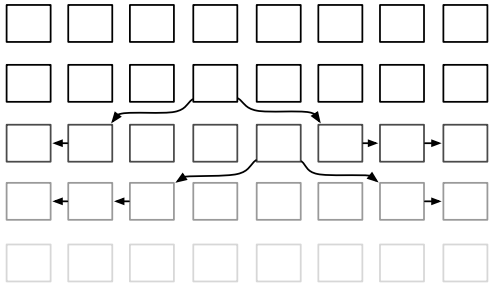


Figure 7.14: *Propagation of states using the* `send-down` *fact.*

If there is any conflict, we do not derive anything and for that we use the language expression `1` (lines 25, 33 and 41). If there are no conflicts, this means that it is possible to add a queen to the current state (line 39). The fact `send-down` is used to either complete the computation of a valid state (lines 45-46) or to propagate the state to the row below (lines 47-49) as shown in Fig. 7.14. The full proof of correctness for the N-Queens program is presented in Appendix G.3.

Since computation goes from the top row to the bottom row, not all static placements of nodes to threads will perform equally well. This is especially true because the bottom rows perform the most work. The best placement is then to split the board vertically with axiom `set-thread(A, vertical(X, Y, side, side))` so each thread gets the same number of columns, where `vertical` is a function that accepts the coordinates `X` and `Y` and the `size` of the board and then returns the thread for node `A`. Our experiments show that, on a shared memory system, it does not matter much if we start with a bad placement since node stealing overcomes those issues by load balancing dynamically.

However, the N-Queens program incrementally builds and shares lists representing valid board states that are transmitted from top to bottom. In order to improve memory locality, we should then manipulate board states that share a significant number of elements because each board state needs to be iterated before being extended with a new position. To accomplish this, we used the coordination fact `set-default-priority(A, X)` (not shown in Fig. 7.15) to experiment with two main configurations: **Top**, where top rows are prioritized; and **Bottom**, where squares closer to the bottom are computed first. We also decided to optionally pin nodes to threads, as mentioned earlier, to see if memory locality affects the multi-threaded performance of the program. These configurations are referred as **Static**.

The complete results are shown in Fig. 7.17 (for the 13-Queens problem) and Fig. 7.18 (for the 14-Queens problem). We present the 4 possible configurations side by side and compare their run time against the version without coordination. The line **Regular(1)/Config(t)** represents the speedup of the configuration **Config** against the run time of the regular version (no coordination) using 1 thread.

The first observation is that using the **Top** configuration for a small number of threads results

```
type list int state.                                                                    // Type declaration
type left(node, node).  type right(node, node).                                         // Predicate declaration
type down-left(node, node).  type down-right(node, node).
type coord(node, int, int).
type linear propagate-left(node, state).
type linear propagate-right(node, state).
type linear test-y(node, int, state, state).
type linear test-diag-left(node, int, int, state, state).
type linear test-diag-right(node, int, int, state, state).
type linear new-state(node, state).
type linear send-down(node, state).
type linear final-state(node, state).

propagate-left(A, State)                                                 // Rule 1: propagate to the left
  -o {L | !left(A, L), L <> A -o propagate-left(L, State)}, new-state(A, State).
propagate-right(A, State)                                               // Rule 2: propagate to the right
  -o {R | !right(A, R), R <> A -o propagate-right(R, State)}, new-state(A, State).

new-state(A, State), !coord(A, X, Y) -o test-y(A, Y, State, State).   // Rule 3: check if queen can be added

// check if there is no queen on the same column
test-y(A, Y, [], State), !coord(A, OX, OY)                                  // Rule 4: check vertical
  -o test-diag-left(A, OX - 1, OY - 1, State, State).
test-y(A, Y, [Y1 | RestState], State), Y = Y1                              // Rule 5: check vertical
  -o 1. // fail
test-y(A, Y, [Y1 | RestState], State), Y <> Y1                              // Rule 6: check vertical
  -o test-y(A, Y, RestState, State).

// check if there is no queen on the left diagonal
test-diag-left(A, X, Y, _, State), X < 0 || Y < 0, !coord(A, OX, OY)          // Rule 7: check left diagonal
  -o test-diag-right(A, OX - 1, OY + 1, State, State).
test-diag-left(A, X, Y, [Y1 | RestState], State), Y = Y1                      // Rule 8: check left diagonal
  -o 1. // fail
test-diag-left(A, X, Y, [Y1 | RestState], State), Y <> Y1                     // Rule 9: check left diagonal
  -o test-diag-left(A, X - 1, Y - 1, RestState, State).

// check if there is no queen on the right diagonal
test-diag-right(A, X, Y, [], State), X < 0 || Y >= size, !coord(A, OX, OY)  // Rule 10: check right diagonal
  -o send-down(A, [OY | State]). // add new queen
test-diag-right(A, X, Y, [Y1 | RestState], State), Y = Y1                   // Rule 11: check right diagonal
  -o 1. // fail
test-diag-right(A, X, Y, [Y1 | RestState], State), Y <> Y1                  // Rule 12: check right diagonal
  -o test-diag-right(A, X - 1, Y + 1, RestState, State).

send-down(A, State), !coord(A, size - 1, _)                                  // Rule 13: final state
  -o final-state(A, State).
send-down(A, State), !coord(A, CX, _), CX <> size - 1                 // Rule 14: propagate the state down
  -o {B | !down-right(A, B), B <> A -o propagate-right(B, State)},
     {B | !down-left(A, B), B <> A -o propagate-left(B, State)}.

propagate-right(@0, []).                                                                // Initial fact
```

Figure 7.15: *N-Queens problem solved in LM.*

in better run times (see Fig. 7.18(a) for an example), however that advantage is lost when adding more threads. We argue that **Top** allows threads to build all the valid states on row i and then process them all at once on row $i + 1$, improving overall memory locality since LM rules run for all possible board configurations of row i. However, when the number of threads increases, it results in reduced scalability since there is less work available per thread since row i must be processed before row $i + 1$.

In the **Bottom** configuration, there is little change when compared to the uncoordinated ver-

```
final-state(@57, [0, 4, 7, 5, 2, 6, 1, 3]).                // States at square (7, 0)
final-state(@57, [0, 5, 7, 2, 6, 3, 1, 4])
final-state(@57, [0, 6, 3, 5, 7, 1, 4, 2])
final-state(@57, [0, 6, 4, 7, 1, 3, 5, 2])

(...)

final-state(@63, [7, 1, 3, 0, 6, 4, 2, 5])                 // States at square (7, 7)
final-state(@63, [7, 1, 4, 2, 0, 6, 3, 5])
final-state(@63, [7, 2, 0, 5, 1, 4, 6, 3])
final-state(@63, [7, 3, 0, 2, 5, 1, 6, 4])
```

Figure 7.16: *Final database for the 8-Queens program.*

sion of N-Queens. However, interesting things happen in the **Bottom Static** configuration when the number of rows is equal to the number of threads. For an example, observe Fig. 7.18d where using 14 threads takes almost the same run time as the regular version when using 32 threads. In this case, each thread is assigned a single column of the chess board and is able to process only the states for that particular column. This behavior was investigated in the Valgrind's CacheGrind tool, which showed that this particular combination has the smallest number of cache misses. This indicates that it is helpful to match the problem to the number of available CPU's in the system in order to increase memory locality.

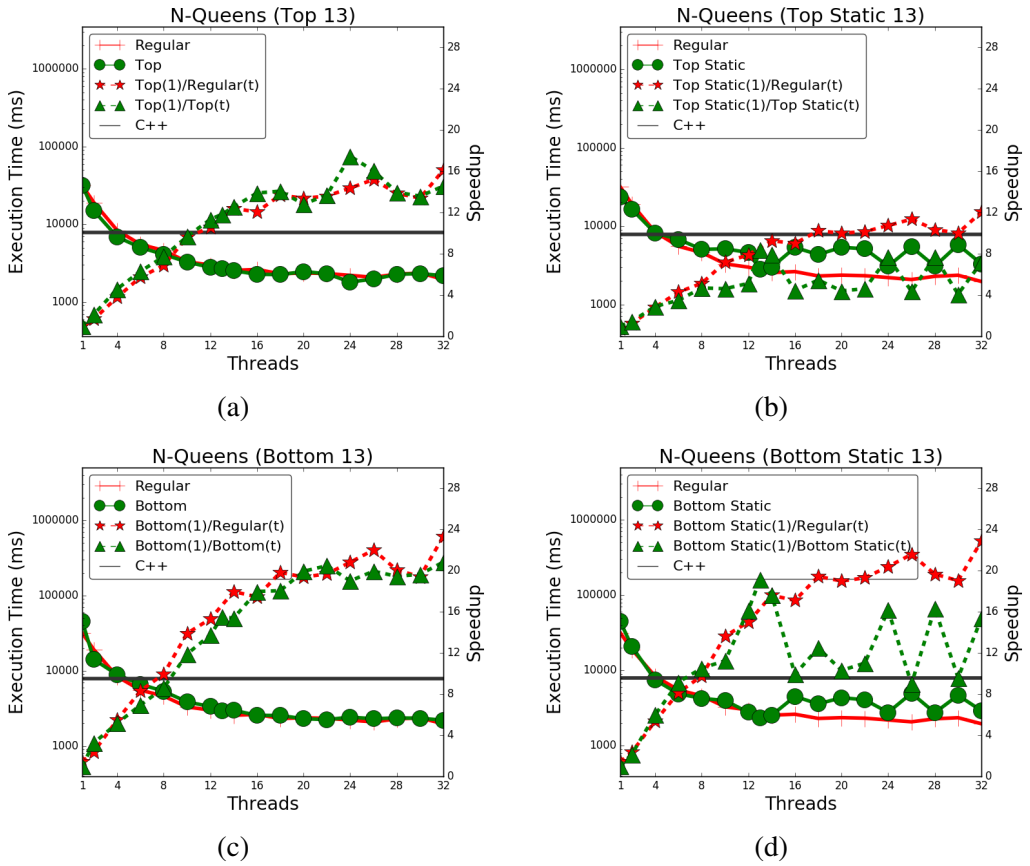


Figure 7.17: *Scalability for the N-Queens 13 program when using different scheduling and partitioning policies.*

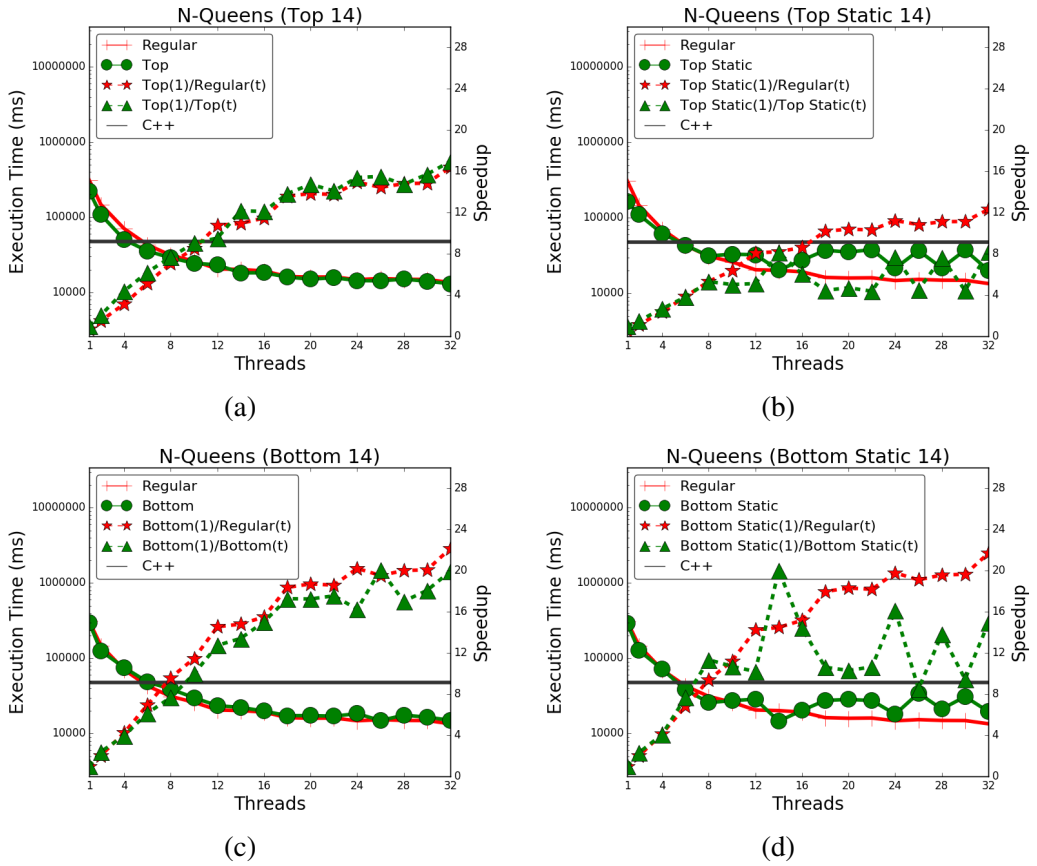


Figure 7.18: *Scalability for the N-Queens 14 program when using different scheduling and partitioning policies.*

7.8.3 Heat Transfer

In the Heat Transfer (HT) algorithm, we have a graph where heat values are exchanged between nodes. The program stops when, for every node i, the new heat values H_i differ only a small ϵ from the old values H_{i-1}, where $\delta = |H_i - H_{i-1}| \leq \epsilon$. The algorithm works asynchronously, i.e., heat values are updated using information as it arrives from neighboring nodes. This increases concurrency since nodes do not need to synchronize between iterations.

Figure 7.19 shows the HT rules that send new heat values to neighbor nodes. In the first rule we added an `add-priority` action fact to increase the priority of the neighbor nodes for the case when the current node has a large δ. The idea is to prioritize the computation of heat values of nodes (using `update`) that have a neighbor that changed significantly. Multiple `add-priority` facts will increase the priority of a node so that nodes with multiple large deltas will have higher priority.

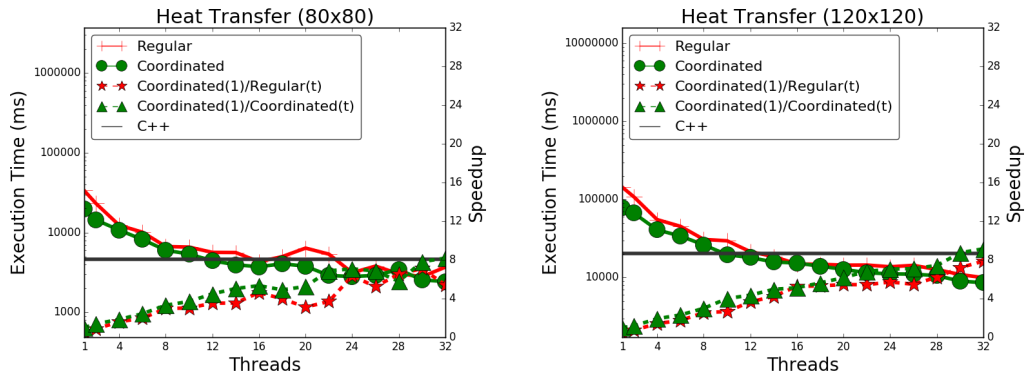
```
1  new-heat(A, New, Old),
2  fabs(New - Old) > epsilon
3     -o {B | !edge(A, B) -o
4           new-neighbor-heat(B, A, New),
5           update(B), add-priority(B, Delta)}.
6
7  new-heat(A, New, Old)
8  fabs(New - Old) <= epsilon
9     -o {B | !edge(A, B) -o
10          new-neighbor-heat(B, A, New)}.
```

Figure 7.19: *Coordination code for the Heat Transfer program.*

Fig. 7.20 presents the scalability results for the regular and coordinated version. The dataset used in the experiments was already used in Section 6.4.1, which is a square grid with an inner square with high heat nodes. Comparing the coordinated version with the regular version, with 1 thread there is a 50% reduction in run time, while for 32 threads there is, on average, a 15% reduction. The increasing number of threads makes selecting the higher priority nodes less likely since each thread is only able to select the higher priority node from its own (ever decreasing) subgraph.

To further improve locality, we modified the second rule to avoid sending small δ values if the target node is in another thread (Fig. 7.21). We used `thread-id` to retrieve the thread `T` of the node `A` and match the `thread-id` of each neighbor `B` against `T`. The comprehension in lines 10-11 only generates `new-neighbor-heat` facts if `B` is in the same thread. Note that in this version, the heat values computed by the program will have increased errors since they are computed with less accurate information. However, it is possible to write more complicated rules where nodes could accumulate incoming heat values and then compute and propagate new heat values when appropriate. This would also increase locality but without increasing the statistical error.

The results for the improved version are presented in Fig. 7.22. There is a clear improvement when compared to the previous version, since when using 32 threads, there is a 25% run time reduction for the coordinated version. For the same number of threads, the line **Coordinated(1)/Coordinated(t)** reaches a 10-fold speedup, while the old version achieves a 9-fold speedup. However, this comes at the price of reduced accuracy in the computed heat values.



(a) *80x80 grid dataset.* (b) *120x120 grid dataset.*

Figure 7.20: *Scalability for the Heat Transfer program when using coordination. We used the datasets used in Section 6.4.1*

```
1  new-heat(A, New, Old),
2  fabs(New - Old) > epsilon
3     -o {B | !edge(A, B) -o
4           new-neighbor-heat(B, A, New),
5           update(B), add-priority(B, Delta)}.
6
7  new-heat(A, New, Old)
8  fabs(New - Old) <= epsilon,
9  thread-id(A, T)
10    -o {B, T | !edge(A, B), thread-id(B, T) -o
11          new-neighbor-heat(B, A, New), thread-id(B, T)},
12       thread-id(A, T).
```

Figure 7.21: *To improve locality, we add an extra constraint to the second rule to avoid sending small δ values if the target node is in another thread.*

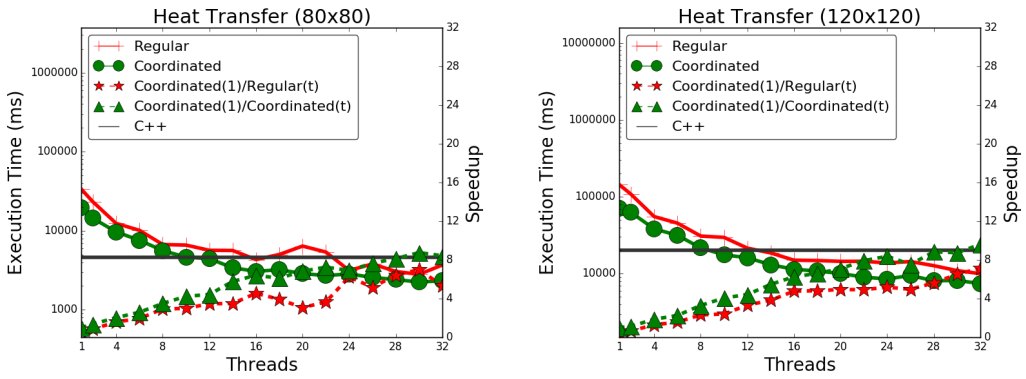
7.8.4 Remarks

We showed four applications that make use of the new coordination facilities of LM. Coordination makes it possible for programmers to fine tune, if necessary, the scheduling and load balancing of declarative programs in the same way that regular LM computation is accomplished.

In the Single Source Shortest Path program, we demonstrated that coordination makes it easy to avoid redundant computations and thus significantly reduce the run time of several, relatively large, datasets.

In the MiniMax program, we used default priorities to minimize the memory usage of the program and allow the declarative program to be far more competitive with the corresponding program implemented in C++.

In the N-Queens program, we used partitioning facts to pin columns of the chess board to threads and then improve the scalability of the program for configurations where the number of columns is equal to the number of threads. The partitioning mechanism allows the programmer



(a) *80x80 grid dataset.* (b) *120x120 grid dataset.*

Figure 7.22: *Scalability for the Heat Transfer program when using coordination and avoiding some communication between nodes of different threads.*

to improve the locality of the program by deciding where nodes and computation should be placed during execution.

Finally, in the Heat Transfer program, we made use of both scheduling and partitioning facts to improve performance and reduce inter-thread communication. The sensing facts for partitioning makes it trivial to selectively do computation by reasoning about the state of the underlying parallel system.

7.9 Related Work

As mentioned in Section 2.3, Linda [ACG86] and Delirium [LS90] are two programming languages that support coordination. When compared to LM, Linda and Delirium are limited in the sense that the programmer can only coordinate the scheduling of processing units, while placement of data is left to the implementation. LM differs from those languages because coordination acts on data instead of processing units. Coordination facts as used in this chapter raise the abstraction by considering data and algorithmic aspects of the program instead of focusing on how processing units are used. Furthermore, LM is both a coordination language and a computation language and the distinction between the two components is small.

There are also several programming models also support some kind of coordination primitives that allow explicit scheduling and load balancing of work between available processing units but are not considered proper programming languages.

The Galois [PNK+11] programming model implements autonomous scheduling by default, where activities may be rolled back in case of conflict. However, it is possible to employ a concrete scheduling strategy for coordinating parallel execution in order to improve execution and avoid conflicts. First, there is *compile-time coordination*, where the scheduling ordered is computed during compilation and is pre-defined before the program is executed. Secondly, there is *runtime coordination*, where the order of activities is computed during execution. The execution of the algorithm proceeds in rounds: first, a set of non-conflicting activities is computed and

then executed by applying the operator; conflicting activities are postponed to the next round. The third and last scheduling strategy is *just-in-time coordination* where the order of activities is defined by the underlying data structure where the operator is applied (for instance, computing on a graph may depend on its topology).

In the context of the Galois model, Nguyen et al. [NP11] expanded the concept of runtime coordination with the introduction of a flexible approach to specify scheduling policies for Galois programs. This approach was motivated by the fact that some algorithms can be executed faster if computations use better scheduling strategies. The scheduling language specifies 3 main scheduler types: FIFO (First-In First-Out), LIFO (Last-In First-Out) and OrderedByMetric (order activities by some metric). These schedulers can be composed and synthesized without requiring users to write complex concurrent code.

Elixir [PMP12] is a domain specific language that builds on top of the Galois and allows easy specification of scheduling strategies. The main idea behind Elixir is that the user should be able to specify how operator application is scheduled and the framework will compile this high level specification to low level code using the provided scheduling specification. One of the motivating examples is the Single Source Shortest Path program that can be specified using multiple scheduling specifications, generating different well-known shortest path algorithms such as the Dijkstra or Bellman-Ford algorithm. Unlike the work of Nguyen et all. [NP11], Elixir does not allow graph mutations.

Halide [RKBA+13] is a language and compiler for image processing pipelines with the goal of optimizing parallelism, locality and re-computation. Halide decouples the algorithm definition from its execution strategy, allowing the compiler to find which execution strategy may be the best for optimizing for locality and parallelism. The language allows the programmer to specify the scheduling strategy, allowing the programmer to decide the order of computations, what intermediate results need to be stored, how to split the data among processing units and how to use vectorization and the well-known sliding window mechanism. However, the compiler is able to use stochastic search to find good schedules for Halide pipelines. Notably, experimental results indicate that automatic search sometimes leads to better execution than hand-written code.

In contrast to the previous systems, LM stands alone in making coordination (both scheduling and partitioning) a first-class programming construct and semantically equivalent to computation. Furthermore, LM distinguishes itself by supporting data-driven dynamic coordination, particularly for irregular data structures. Elixir and Galois do not support coordination for data partitioning, and, in Elixir, the coordination specification is separated from computation, limiting the programmability of coordination. Compared to LM, Halide is targeted for regular applications and therefore only supports compile time coordination.

7.10 Chapter Summary

In this chapter, we presented the set of coordination facts, a new declarative mechanism for coordinating declarative parallel programs. Coordination facts are implemented as sensing and action facts and allow the programmer to write derivation rules that change how the runtime system schedules computation and partitions the data in the parallel system, thus improving the executing time. In terms of programming language design, our coordination mechanisms are

unique in the sense that they are treated like regular computation, which allows for complex run-time coordination policies that are declarative and is part of the main program's logic.

Chapter 8

Thread-Based Facts

In the previous chapter, we introduced coordination facts, a mechanism that can be used by the programmer to coordinate execution. While these facilities retain the implicit parallelism of the language, they do not allow the programmer to fully reason about the underlying parallel architecture since the only reasoning allowed relates to node partitioning and data placement. In principle, it should be advantageous to reason about thread state, that is, to perform rule inference about facts stored on each thread and allow threads to communicate and coordinate between them depending on their current state. This introduces a kind of explicit parallelism into the implicit parallel model of LM. However, this explicit parallelism remains declarative so that the programmer is able to reason about thread coordination as well as regular computation.

8.1 Motivating Example: Graph Search

Consider the problem of checking if a set of nodes S in a graph G is reachable from an arbitrary node N. An obvious solution to this problem is to start at N, gather all the neighbor nodes into a list and then recursively visit all those reachable nodes, until S is covered. This reduces to a problem of performing a breadth or depth-first search on graph G. However, this solution is sequential and does not have much concurrency. An alternative solution to the problem is to recursively propagate the search to all neighbors and aggregate the results in the node where the search started. The code for this later solution is shown in Fig. 8.1, where nodes are searched by analyzing their unique `value` facts.

Each distinct reachability search is represented by a number (`Id`) and a `search` initial fact. Associated to each search `Id` is a list of nodes to reach. The predicate `visited` marks nodes that have already participated in search, while predicate `do-search` is used to propagate a specific search. The first rule (lines 12-14) starts a particular search by deriving a `do-search` and an `lookup` fact. The `lookup` fact is used as an accumulator and is stored in the starting node. The third rule (lines 19-21) avoids visiting the same node twice in the presence of a `visited` fact. This visited fact is derived in the next two rules (lines 25 and 32). If the node has a `value` in the reachability set (`ToReach`) then we remove the node value from the list and propagate the search to the neighbor nodes (line 27). Otherwise, the search is propagated but no value is removed from `ToReach`.

```
type int id.                                                    // Type declaration
type list int reach-list.

type edge(node, node).                                      // Predicate declaration
type value(node, int).
type linear search(node, id, reach-list).
type linear do-search(node, id, node, reach-list).
type linear lookup(node, id, reach-list, int Val).
type linear new-lookup(node, id, int Val).
type linear visited(node, id).

search(A, Id, ToReach)                                  // Rule 1: initialize search
   -o do-search(A, Id, A, ToReach),
      lookup(A, Id, ToReach, []).

lookup(A, Id, ToReach, Found), new-lookup(A, Id, Val)// Rule 2: new reachable node found
   -o lookup(A, Id, remove(ToReach, Val), [Val | Found]).

do-search(A, Id, Node, ToReach),          // Rule 3: node has already seen this search
visited(A, Id)
   -o visited(A, Id).

do-search(A, Id, Node, ToReach),            // Rule 4: node found and propagate search
!value(A, Val), Val in ToReach                                  // New node was found.
   -o visited(A, Id),
      new-lookup(Node, Id, Val),
      {B | !edge(A, B) -o do-search(B, Id, Node, remove(ToReach, Val))}.

do-search(A, Id, Node, ToReach),        // Rule 5: node not found and propagate search
!value(A, Val), ~ Val in ToReach                    // Not the node we are looking for.
   -o {B | !edge(A, B) -o do-search(B, Id, Node, ToReach)},
      visited(A, Id).
```

Figure 8.1: *LM code to perform reachability checking on a graph.*

As an example, consider Fig. 8.2, which shows 2 reachability checks on a graph with 10 nodes. For instance, the search with `Id = 0` starts at node `@1` and checks if nodes `@1`, `@2`, and `@3` are reachable from `@1`. Since `@1` is the starting node, `1` is immediately removed from the reachable list, including the propagated `do-search` facts but also the `lookup` fact that is stored at node `@1`. Once `do-search` reaches node `@3`, the value `3` is removed from the list and a new `do-search` is propagated to node `@1` (not shown in the figure) and `@2`. At the same time, node `@2` receives the list `[2,3]`, removes `2` and propagates `[3]` to node `@3` and `@1`. Node `@1` receives two `new-lookup` facts, one from `@3` and another from `@2`, due to successful searches and the `lookup` fact becomes `lookup(@1,0,[], [1,2,3])`.

The attentive reader will notice that node `@1` already knows that all the nodes have been reached and that nodes `@7` and `@4` will, needlessly, continue to check if `[2,3]` are reachable. This is an issue that arises because the programmer has valued concurrency by increasing redundancy and reducing communication between nodes. It would be expensive to share reachability information between nodes. An alternative solution is to store the results of the search on the

thread performing the search and then coordinate the results with other threads since the number of threads is usually smaller than the number of nodes. Before showing how the reachability program is solved using thread-based facts, we first present the changes to the language.

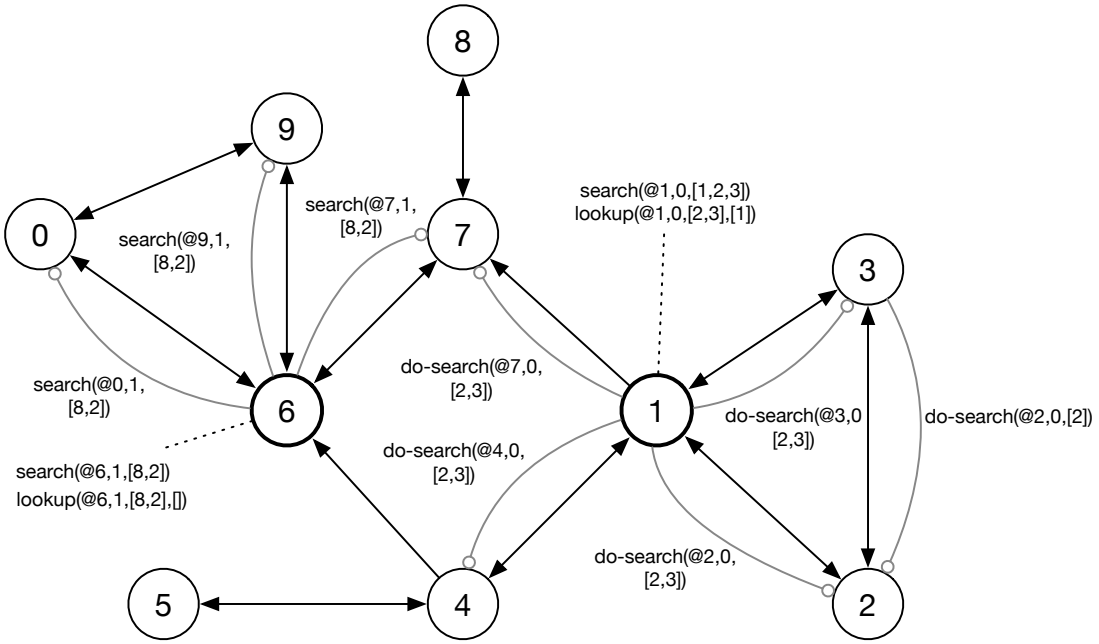


Figure 8.2: *Performing reachability checks on a graph using nodes* `@1` *(*`Id = 0`*) and* `@6` *(*`Id = 1`*). Search with* `Id = 0` *wants to reach nodes* `@1`, `@2`, *and* `@3` *from node* `@1`. *Since* `@1` *is part of the target nodes, the fact* `do-search` *propagated to neighbor nodes does not include* `1`.

8.1.1 Language Changes

In the previous chapter, we presented coordination facts such as `thread-id` and `set-thread` that already bring some awareness about the underlying parallel system. Furthermore, such facts also introduce the `thread` type for predicate arguments, which refers to a thread in the system that is related to a core in a multi core processor. We now introduce the concept of *thread facts*, which are logical facts stored at the thread level, meaning that, each thread is now an entity with its own logical facts. The type `thread` is also now the type of the first argument of *thread predicates*, indicating that the predicate is related and is to be stored in a specific thread. We also view the available threads as forming a separate graph from the data graph, a graph of the processing units which are operating on the data graph.

The introduction of thread facts increases the expressiveness of the system in the sense that it is now possible to write inference rules that reason about the state of the threads. This creates optimization opportunities since we can now write algorithms with global information stored in the thread, while keeping the LM language fully declarative. Moreover, threads are now allowed to explicitly communicate with each other, and in conjunction with coordination predicates, enable the writing of complex scheduling policies.

We discriminate between two new types of inference rules. The first type is the *thread rule* and has the form `a(T), b(T) -o c(T)`, and can be read as: if thread `T` has fact `a(T)` and `b(T)` then derive fact `c(T)`. The second type is the *mixed rule* and has the form `a(T), d(N) -o e(N)` and can be read as: if thread `T` is executing node `N` and has the fact `a(T)` and node `N` has the fact `d(N)` then derive `e(N)` at node `N`. Thread rules reason solely at the thread level, while mixed rules allow reasoning about both thread and node facts. Logically, the mixed rule uses an extra fact `running(T, N)`, which indicates that thread `T` is currently executing node `N`. The `running` fact is implicitly retracted and asserted every time the thread selects a different node for execution. This makes our implementation efficient since a thread does not need to look for nodes that match mixed rules and it is then the scheduling of the program that drives the matching of such rules.

8.1.2 Graph Of Threads

Figure 8.3 represents a schematic view of the two graph data structures of a program with three threads: thread $T1$ is executing node `@5`, $T2$ is executing node `@4`, and $T3$ is executing node `@3`. Note that every thread has access to its own facts and to the node facts.

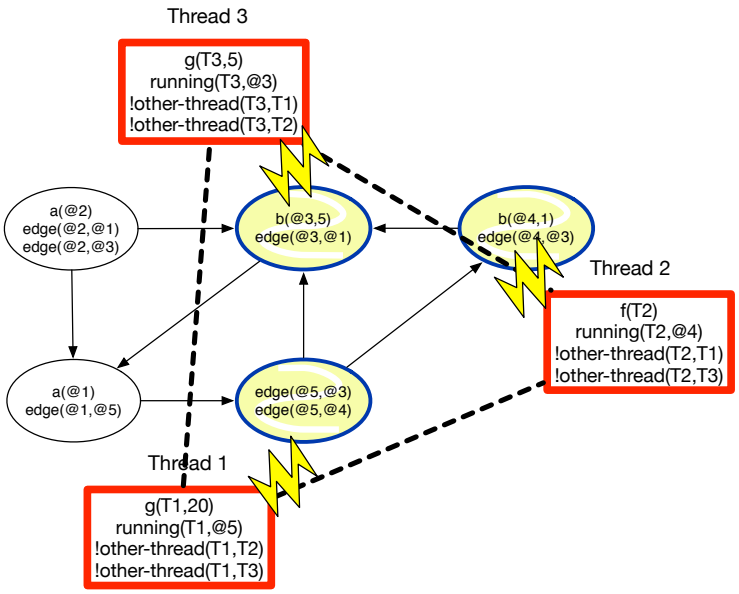


Figure 8.3: *An example program being executed with three threads. Note that each threads has a* `running` *fact that stores the node currently being executed.*

We added several helpful predicates that allow the programmer to inspect the graph of threads and reason about the state of computation as it relates to threads:

- `!thread-list(T, L)`: Fact instantiated in all threads where `L` is a list of all threads executing in the system.
- `!other-thread(T1, T2)`: Connects thread `T1` to all the other threads `T2` executing in the system. Note that in Fig. 8.3, we use `!other-thread` fact to specify the graph of threads.

- `!leader-thread(T, TLeader)`: Fact instantiated in all threads where `TLeader` refers to a selected thread (the first thread in `L` of `!thread-list(T, L)`).
- `running(T, A)`: Used to retrieve the current node `A` running on thread `T`.

With the exception of `running`, every other fact is added at the beginning of the program as a persistent fact.

8.1.3 Reachability With Thread Facts

We now update the graph reachability program presented in Fig. 8.1 to use thread facts in order to avoid needless searches on the graph. The search process is still done concurrently as before, but the search state is now stored in each thread, allowing the thread to store partial results and coordinate with other threads. The code for this new version is shown in Fig. 8.4.

Lines 1-4 start the search process by assigning a thread `Owner` to search `Id` using the persistent fact `!thread-list` which contains the list of all available threads in the system. Next, in line 3, a fact `thread-search` is created for all threads using a comprehension. We use predicate `do-search` to propagate the search through the graph and a predicate `visited` to mark nodes already processed for a specific search. The two rules in lines 14-27 propagate the search process to the neighbor nodes and check if the current node is part of the list of nodes we want to reach.

An interesting property of this version is that each owner thread responsible for a search keeps track of the remaining nodes that need to be reached. In line 18, we derive `remove-thread-search` in order to inform owner threads about new reachable nodes. Once an owner thread detects that all nodes have been reached (lines 33-36), all the other threads will know that and update their search state accordingly (lines 42-44). When every thread knows that all nodes were reached, they will consume `do-search` facts (lines 6-8), effectively pruning the search space.

An alternative implementation could force every thread to share its reached nodes to all the other threads in the system. However, this would generate a lot of traffic between threads, which would actually make the program perform worse. Our final solution is a good trade off since it only forces threads to coordinate when pruning can actually happen.

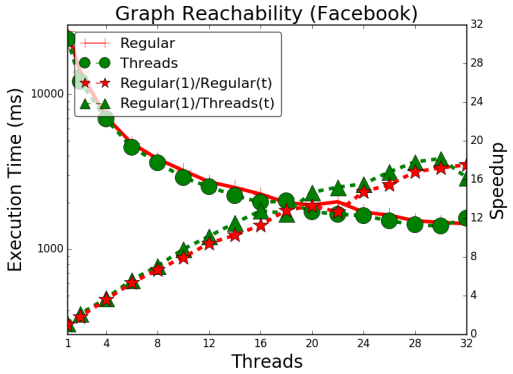
Figure 8.5 presents experimental results of the graph reachability program using 4 different datasets. Except for Random, all the datasets were already used in the MSSD program and were presented before. The Random dataset is a randomly generated graph with 50000 nodes and about a million edges. In the plots, we show the run time of the version without thread facts (**Regular**) and the version using thread facts called **Threads**. We also show the speedup of the **Threads** and **Regular** versions against the **Regular** version with 1 thread.

Our results indicate that using thread facts produces a significant reduction in run time. This is especially true in the case of datasets with large number of edges, since less facts are produced and propagated in the graph when the threads know that the search has been completed. The results also show that, in the case of the Facebook dataset, in which the number of queries is the same as the number of nodes, the use of thread facts does not produce great improvements due to the costs of managing the reachability results on each thread's database. These costs are related to the need to index and lookup `thread-search` facts on the `Id` argument every time a node is inspected.

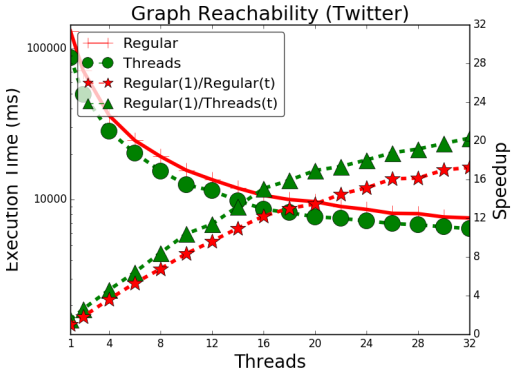
The graph reachability program shows how to introduce complex coordination policies between threads by reasoning about the state of each thread. In addition, the use of linear logic programming makes it easier to prove properties of the program since computation is done by applying controlled changes to the state.

```
search(A, Id, ToReach),                                    // Rule 1: initialize search
!thread-list(T, L), Owner = nth(L, Id % @threads)       // Allocate search to a thread
   -o {T2 | !other-thread(T, T2) -o thread-search(T2, Id, ToReach, Owner)},
      do-search(A, Id).

thread-search(T, Id, [], Owner),                           // Rule 2: search completed
do-search(A, Id)
   -o thread-search(T, Id, [], Owner).

do-search(A, Id),                                       // Rule 3: node already visited
visited(A, Id)
   -o visited(A, Id).

do-search(A, Id),                                                  // Rule 4: node found
thread-search(T, Id, ToReach, Owner),
!value(A, Val), Val in ToReach
   -o thread-search(T, Id, remove(ToReach, Val), Owner),
      remove-thread-search(Owner, Id, Val),             // Tell owner thread about it.
      {B | !edge(A, B) -o do-search(B, Id)},
      visited(A, Id).

do-search(A, Id),                        // Rule 5: node not found but propagate search
thread-search(T, Id, ToReach, Owner),
!value(A, Val), ~ Val in ToReach
   -o thread-search(T, Id, ToReach, Owner),
      visited(A, Id),
      {B | !edge(A, B) -o do-search(B, Id)}.

remove-thread-search(T, Id, Val), thread-search(T, Id, ToReach, Owner)// Rule 6: node found
   -o thread-search(T, Id, remove(ToReach, Val), Owner),
      check-results(T, Id).

check-results(T, Id),                                      // Rule 7: search is completed
thread-search(T, Id, [], Owner)
   -o thread-search(A, Id, [], Owner),
      {B | !other-thread(T, B) -o signal-thread(B, Id)}.

 check-results(T, Id),                                 // Rule 8: search not completed yet
 thread-search(T, Id, ToReach, Owner), ToReach <> []
   -o thread-search(T, Id, ToReach, Owner).

signal-thread(T, Id),                               // Rule 9: thread knows search is done
thread-search(T, Id, ToReach, Owner)
   -o thread-search(T, Id, [], Owner).
```

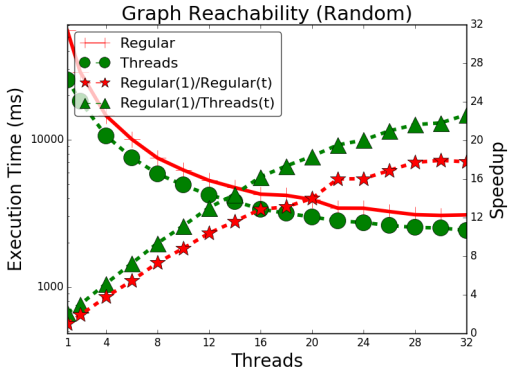
Figure 8.4: *Coordinated version of the reachability checking program. Note that* `@threads` *represent the number of threads in the system.*



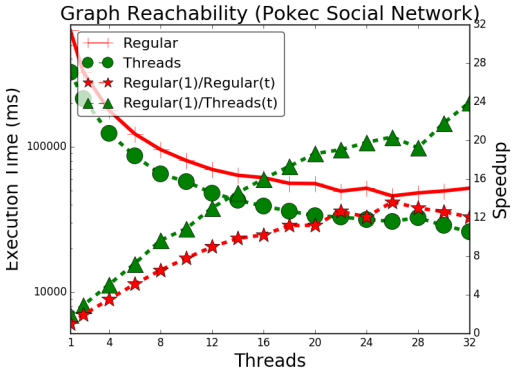
(a) *Facebook has 2000 nodes and 20000 edges. The dataset makes 2000 graph queries to 5% of the graph's nodes.*



(b) *Twitter has 81306 nodes and 1768149 edges. The dataset makes 100 graph queries to 1% of the graph's nodes.*



(c) *Random is a graph with 50000 nodes and 1052674 edges. The dataset makes 20 graph queries to 5% of the graph's nodes.*



(d) *Pokec has 1632803 nodes and 30622564 edges. The dataset makes 3 graph queries to 1% of the graph's nodes.*

Figure 8.5: *Measuring the performance of the graph reachability program when using thread facts.*

8.2 Implementation Changes

To support thread-based facts, both the compilation and runtime system described in Chapter 6 require some changes.

8.2.1 Compiler

The compiler needs to recognize rules that use thread facts. For thread rules, the compiler checks if the rule's body is using facts from the same thread by checking the first argument of each fact. For mixed rules, the rule's body may use a thread `T` and a node `A` and all the node facts have to use `A`, while all threads facts must use `T` as the first argument. If the programmer was to retrieve either the thread or the node for the current computation, she may use `running(T, A)`.

Once rules are type checked, the iteration code for thread-based facts needs to be adapted. When a rule requires facts from the thread, it must use the data structures from the thread. The runtime API used for inserting thread facts is also different since they have to be added to the thread's database.

8.2.2 Runtime

In the runtime system, thread-based facts are implemented just like regular facts. Each thread has its own database of facts and uses exactly the same data structures for facts, as presented before for regular nodes. The major difference between a regular node and a thread node is that a thread node is never put into the work queue of its thread. As shown in the updated work loop presented in Fig. 8.6, the thread node executes alongside the regular node when $TH.process\_node$ is called. It is also important to note that, before a thread becomes idle, it may have potential candidate thread rules that are now derivable because another thread has derived thread facts in the current thread. In particular, it is entirely possible to have programs that only deal with thread facts.

Thread-based facts also introduce new synchronization logic in the runtime system. For instance, when a rule derives a new thread fact on another thread, it needs to synchronize with that thread (using the appropriate thread node locks) to add the facts to the thread's database. When a thread is executing its own node or a regular node, it also must lock the thread node's *DB Lock* in order to protect its data structures from being manipulated by other threads concurrently.

Matching rules using thread facts requires special care since they may require both facts from the regular node and from the thread's node. Before a node is executed, the rule engine (Section 5.3) of the regular node is updated to take into account the facts of the thread's node so that mixed rules (rules that use both thread and regular facts) execute. In this scheme, mixed rules may be unsuccessfully fired repeatedly until a node which has matching facts gets to execute. Since the LHS of mixed rules use an implicit `running(T, N)` fact, it is enough that a different node is running to fire mixed rules as long as the running node has the required facts. Without using an implicit `running` fact, the system would need to lookup for a regular node that would successfully activate a given mixed rule. This would be expensive since some programs might have millions of regular nodes.

```
Data: Thread TH
while true do
    TH.work_queue.lock();
    node ⟵ TH.work_queue.pop_node() ;
    TH.work_queue.unlock();
    if node then
        TH.process_node(node, TH.thread_node);
    else
        // The thread's node may have candidate rules using incoming
           thread facts
        TH.process_node(nil, TH.thread_node);
        // Attempt to steal some nodes.
        if !TH.steal_nodes() then
            TH.become_idle();
            while len(TH.work_queue) == 0 do
                // Try to terminate
                if TH.synchronize_termination() then
                    terminate;
                end
                if TH.steal_nodes() then
                    // Thread is still in the stealing state
                    break;
                end
            end
            // There's new nodes in the queue.
            TH.become_active();
        end
    end
end
```

Figure 8.6: *Thread work loop updated to take into account thread-based facts. New or modified code is underlined.*

8.3 Applications

This section presents more applications that demonstrate the usefulness and power of thread-based facts.

8.3.1 Binary Search Trees: Caching Results

In Section 3.1.2, we have presented an algorithm for replacing a key's value in a BST dictionary. To make the program more interesting, we consider a sequence of n lookup or replace operations for different keys in the BST (which may or may not be repeated). A single lookup or replace has worst-case time complexity $\mathcal{O}(h)$ where h is the height of the BST, therefore performing n operations takes $\mathcal{O}(h \times n)$ time.

In order to reduce the execution time of the new program, we can cache the search and replace operations so that repeated operations become faster. Instead of traversing the entire height of the BST, we look in the cache and send the operation immediately to the node where the key is located. Without thread facts, we might have cached the results at the root node, however, this is not a scalable approach as it would introduce a serious bottleneck.

Figure 8.7 shows the updated BST code with a thread cache. We just added two more predicates, `cache` and `cache-size`, that are facts placed in the thread and represent cached keys and the total size of the cache, respectively. We also added three new rules that handle the following cases:

1. A key is found and is also in the cache (lines 8-12 in Fig. 8.7);
2. A key is found but is not in the cache (lines 14-19 in Fig. 8.7);
3. A key is in the cache, therefore a `replace` fact is derived in the target node (lines 21-24).

In Appendix G.4 we prove several properties about the extended BST program. Also note that it is quite easy to extend the cache mechanism to use an LRU type approach in order to limit the size of the cache.

In order to understand the effectiveness of the new cached version of the program, we created synthetic datasets that stress the algorithm using a complete binary search tree and an arbitrary number of (repeated) tree operations such as replace and lookup. The tree operations apply to a subset of the tree and each node is associated with a list of operations that need to be performed. We have auxiliary nodes (not connected to the tree) that coordinate the tree operations for a given node. An auxiliary node will start the first operation by sending it to the tree root node. Once the operation is performed on the target node, the auxiliary node will start the next operation in the list for that target node. By using auxiliary nodes, we are able to experiment with operations that target the same node and also allow for some parallelism when doing tree operations. In our experiments, we generated two trees with the following characteristics:

- 19 levels: a tree with 19 levels ($2^{20} - 1$ nodes) and two million operations. Each targeted node has, on average, 40 operations.
- 21 levels: a tree with 21 levels ($2^{22} - 1$ nodes) and five million operations. Each targeted node has, on average, 20 operations.

Note that in both datasets, the operations target 10% of the tree nodes.

Table 8.1 presents fact statistics for the **Regular** version (without a cache) and the **Cached** version (with a cache). Figure 8.8 presents the run time scalability of both versions when using the **Regular** with 1 thread as the baseline. In terms of facts, the **Cached** version sees an 80% reduction in number of derived facts for 1 thread. When the number of threads increases, more facts are being set since there is less opportunity to cache tree nodes since there are more threads doing more work at the same time. For instance, when using 32 threads, the reduction is only about 30%.

When comparing these reduction numbers to the reduction in run time, we see the same pattern but the reduction is not quite as large. For instance, when using 1 thread, the **Cached** version is only able to reduce the run time in about 50%, and for 32 threads, only a 12% reduction is seen. Threads need to index and lookup many cache items using the node's key. This process is somewhat expensive since threads will lookup keys as each operation travels down the tree looking for a node. For instance, if a thread does not have a given node in the cache, it may need to perform about h (the height of the tree) lookup into the database hash tree before the node arrives at the destination. However, a given tree operation is usually not handled by just one thread as it travels through the tree and another thread may have the target node in its cache.

Dataset	**Threads**	**# Derived**		**# Deleted**		**# Final**	
		Regular	Cached	Regular	Cached	Regular	Cached
19 levels	1	45.6M	23%	42.4M	17%	3.1M	3.2M
	2	45.6M	30%	42.4M	24%	3.1M	3.2M
	4	45.6M	35%	42.4M	30%	3.1M	3.2M
	8	45.6M	38%	42.4M	33%	3.1M	3.2M
	16	45.6M	51%	42.4M	48%	3.1M	3.2M
	24	45.6M	64%	42.4M	62%	3.1M	3.2M
	32	45.6M	70%	42.4M	67%	3.1M	3.2M
21 levels	1	126.5M	24%	116.9M	18%	9.6M	9.8M
	2	126.5M	28%	116.9M	22%	9.6M	9.8M
	4	126.5M	34%	116.9M	28%	9.6M	9.8M
	8	126.5M	41%	116.9M	36%	9.6M	9.8M
	16	126.5M	50%	116.9M	46%	9.6M	9.8M
	24	126.5M	58%	116.9M	54%	9.6M	9.8M
	32	126.5M	66%	116.9M	63%	9.6M	9.8M

Table 8.1: *Fact statistics for the **Regular** version and the **Cached** version. The first two Cached columns show the ratio of the **Cached** version over the **Regular** version. Percentages less than 100% means that the **Cached** version produces fewer facts.*

```
type left(node, node).                                          // Predicate declaration
type right(node, node).
type linear value(node, int Key, string Value).
type linear replace(node, int Key, string Value).
type linear cache(thread, node, int).
type linear cache-size(thread, int).

replace(A, Key, RValue),                // Rule 1: key exists and is also in the cache
value(A, Key, Value),
cache(T, A, Key)
   -o value(A, Key, RValue),
      cache(T, A, Key).

replace(A, Key, RValue),                 // Rule 2: key exists and is not in the cache
value(A, Key, Value),
cache-size(T, Total)
   -o value(A, Key, RValue),
      cache-size(T, Total + 1),
      cache(T, A, Key).

replace(A, RKey, RValue),                              // Rule 3: cached by the thread
cache(T, TargetNode, RKey)
   -o replace(TargetNode, RKey, RValue),
      cache(T, TargetNode, RKey).

replace(A, RKey, RValue),                                        // Rule 4: go left
value(A, Key, Value),
!left(A, B),
RKey < Key
   -o value(A, Key, Value),
      replace(B, RKey, RValue).

replace(A, RKey, RValue),                                       // Rule 5: go right
value(A, Key, Value),
!right(A, B),
RKey > Key
   -o value(A, Key, Value),
      replace(B, RKey, RValue).

cache-size(T, 0).                                  // Initial cache size of each thread
```

Figure 8.7: *LM program for performing lookups in a BST extended to use a thread cache.*

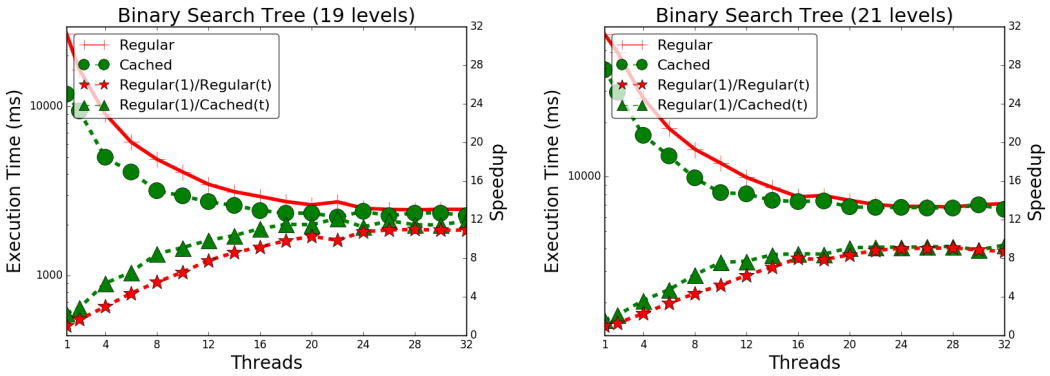


Figure 8.8: *Scalability for the Binary Search Tree program when using thread based facts.*

8.3.2 PowerGrid Problem

Consider a power grid with C consumers and G generators. We are interested in connecting each consumer to a single generator, but each generator has a limited capacity and the consumer draws a certain amount of power from the generator. A valid power grid is built in such a way that all consumers are serviced by a generator and that no generator is being overdrawn by too many consumers. Although consumers and generators may be connected through a complex network, we analyze the simple case where any consumer is able to attach to any generator.

A straightforward distributed implementation for the PowerGrid problem requires that each consumer is able to connect to any generator. Once a generator receives a connection request, it may or may not accept it. If the generator has no power available for the new consumer, it will disconnect from it and the consumer must select another generator. This randomized algorithm works but may take a long time to converge, depending on the amount of power available in the generators. Figure 8.10 shows the LM code for this solution. Consumer and generators node types are declared in lines 1-2 using the `node` declaration, allowing us to have different predicates for consumers and generators. The `consumer` and `generator` types become a subtype of *node*, that is, $consumer <: node$ and $generator <: node$. These subtypes allow us to declare initial facts that only apply to either the `consumer` or `generator` subtype.

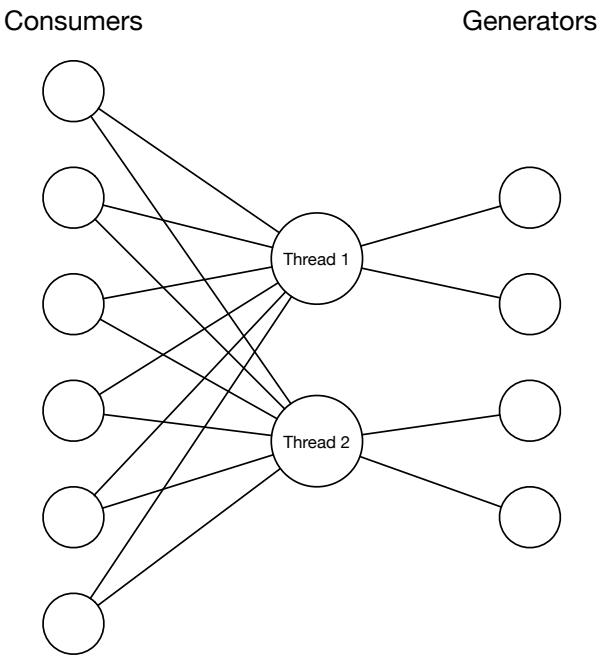


Figure 8.9: *Configuration of a power grid with 6 consumers, 4 generators and 2 threads, with each thread responsible for 2 generators.*

An example PowerGrid configuration with its initial facts is presented in Fig. 8.11. Consumers have a persistent fact `!power(A, P)`, where `P` is the amount of power required by the consumer. Consumers also start with a `reconnect` fact that is used in lines 39-42 in order to randomly select a generator from list `L` in the `!generators(A, L)` fact. The generators have a `connect-to-list(A, L)` fact that manages the list of connected consumers. The generator fact `capacity(A, Total, Used)`, stores the `Total` capacity of the generator and the amount of power currently being `Used` (`Used < Total` at any point in the program).

Consumers and generators complete a connection when the generator receives a `connect` fact which is used in lines 28-32 when the generator has enough power for the new consumer. When there is not enough power (`Used + Power > Total`), the generator disconnects the consumer in lines 34-37. Note that each generator maintains a `fail` fact that counts the number of times the consumers have failed to connected. If there is too many failures, then the generator decides to disconnect one consumer already connected in lines 16-26, allowing for different combinations

```
node generator.                                                    // Type declaration
node consumer.
type linear capacity(generator, int Total, int Used).          // Predicate declaration
type linear connected-to(generator, consumer, int).
type linear connected-to-list(generator, list consumer).
type power(consumer, int).
type linear disconnected(consumer).
type linear connected(consumer, generator).
type generators(consumer, list generator).
type linear fails(generator, int).
type linear random-reconnect(generator).
type linear reconnect(consumer).
type linear connect(generator, consumer, int).
type linear disconnect(consumer, generator).

fails(G, Fails), Fails > maxfails                    // Rule 1: disconnect one consumer
   -o random-reconnect(G).

capacity(G, Total, Used), random-reconnect(G),       // Rule 2: disconnect one consumer
connected-to-list(G, L), L <> [], C = nth(L, randint(length(L))),
connected-to(G, C, Power)
   -o fails(G, 0), capacity(G, Total, Used - Power),
      connected-to-list(G, remove(L, C)), disconnect(C, G).

capacity(G, Total, Used), random-reconnect(G)// Rule 3: unable to disconnect one consumer
   -o capacity(G, Total, Used), fails(G, 0).

connect(G, C, Power), capacity(G, Total, Used),             // Rule 4: connect consumer
fails(G, Fails), connected-to-list(G, L), Used + Power <= Total
   -o capacity(G, Total, Used + Power),
      fails(G, max(Fails - 1, 0)), connected-to(G, C, Power),
      connected-to-list(G, [C | L]).

connect(G, C, Power), capacity(G, Total, Used),   // Rule 5: unable to connect consumer
Used + Power > Total, fails(G, Fails)
   -o capacity(G, Total, Used), disconnect(C, G),
      fails(G, Fails + 1).

!generators(C, L), !power(C, Power),                 // Rule 6: connect to a generator
reconnect(C), disconnected(C),
G = nth(L, randint(num-generators))
   -o connected(C, G), connect(G, C, Power).

disconnect(C, G), connected(C, G)                      // Rule 7: finish disconnection
   -o disconnected(C), reconnect(C).

connected-to-list(G, []). fails(G, 0).                                 // Initial facts
disconnected(C). reconnect(C). !generators(C, all-generators).
```

Figure 8.10: *LM code for the regular PowerGrid program.*

to happen. In Fig. 8.12 we present the final database of the example PowerGrid configuration, which shows that all consumers have been able to find a suitable generator.

```
const generators = [@7, @8, @9, @10].

reconnect(@1).   !generators(@1, generators).   !power(@1, 5).
reconnect(@2).   !generators(@2, generators).   !power(@2, 10).
reconnect(@3).   !generators(@3, generators).   !power(@3, 5).
reconnect(@4).   !generators(@4, generators).   !power(@4, 10).
reconnect(@5).   !generators(@5, generators).   !power(@5, 10).
reconnect(@6).   !generators(@6, generators).   !power(@6, 5).

connected-to-list(@7, []).    capacity(@7, 15, 0).   fail(@7, 0).
connected-to-list(@8, []).    capacity(@8, 15, 0).   fail(@8, 0).
connected-to-list(@9, []).    capacity(@9, 10, 0).   fail(@9, 0).
connected-to-list(@10, []).   capacity(@10, 5, 0).   fail(@10, 0).
```

Figure 8.11: *Initial facts for a PowerGrid configuration of 6 consumers and 4 generators.*

```
connected(@1, @7).    !power(@1, 5).
connected(@2, @7).    !power(@2, 10).
connected(@3, @8).    !power(@3, 5).
connected(@4, @8).    !power(@4, 10).
connected(@5, @9).    !power(@5, 10).
connected(@6, @10).   !power(@6, 5).

connected-to-list(@7, [@1, @2]).   connected-to(@7, @1, 5).   connected-to(@7, @2, 10).
connected-to-list(@8, [@3, @4]).   connected-to(@8, @3, 5).   connected-to(@8, @4, 10).
connected-to-list(@9, [@5]).       connected-to(@9, @5, 10).
connected-to-list(@10, [@6]).      connected-to(@10, @6, 5).
capacity(@7, 15, 15).
capacity(@8, 15, 15).
capacity(@9, 10, 10).
capacity(@10, 5, 5).
```

Figure 8.12: *Final facts for a PowerGrid configuration of 6 consumers and 4 generators.*

The issue with this initial implementation presented in Fig. 8.10 is that it lacks a global view of the problem, which introduces inefficiencies and more communication between consumers and generators. A better algorithm will require a more sophisticated communication pattern between the nodes. As we have seen before, thread local facts are an excellent mechanism to introduce a global view of the problem without complicating the original algorithm written in a declarative style. For our solution, we will partition the set of generators G among the threads in the system and make each thread assume the ownership of its generators. Each thread can then process consumers with a global view over its set of generators, allowing the immediate assignment of consumers to generators. Figure 8.9 shows how the configuration presented previously is adjusted to take into account the number of available threads.

The LM code using thread facts shown in Fig. 8.13. It uses four thread predicates: predicates `thread-connected-to` and `thread-connected-to-list` assign consumers to the generators

owned by the thread; `thread-capacity` stores the capacity of each generator assigned to the thread; and predicate `thread-total-capacity` provides a capacity overview of all the generators owned by the thread. The program starts with the rule in lines 7-8 by moving generators to their corresponding threads using `set-thread`. Once the generator is executing on the proper thread, the rule in lines 10-12 is derived with the `just-moved` coordination fact and the state of the thread is initialized. Consumers connect with the thread's generators with the rule in lines 37-43 by selecting the first generator with enough power and then by updating the state of the thread. Otherwise, if the thread does not have a suitable generator, a generator is randomly selected using the method described before. For such cases, the thread assigned with the selected generator will derive the rules in lines 26-35 and the thread state is updated accordingly.

The experimental results for the PowerGrid program are presented in Fig. 8.14. In the plots, we show the run time of the version without thread facts, named **Regular**, and the version using thread facts, called **Threads**. We also show the speedup of the **Threads** and **Regular** versions against the **Regular** version with 1 thread. We experimented with four datasets:

1. 0.5 M C / 2000 G: Half a million consumers connected to 2000 generators. This is the baseline dataset.
2. 0.5 M C / 64 G: Half a million consumers connected to 64 generators. This is similar to the previous dataset, but the number of generators is much smaller.
3. 1 M C / 4000 G / Low Variability: 1 million consumers connected to 4000 generators. The capacity of each generator has low variability so that each generator has a similar capacity.
4. 1 M C / 4000 G / High Variability: 1 million consumers connected to 4000 generators. The capacity of each generator has high variability so that some generators have no capacity or three times the average the capacity.

All the datasets were generated randomly so that the total capacity of the generators is about 2% more than the required consumer capacity. The capacity of each generator is generated randomly and, except for the last two datasets, the capacity is between 0 and twice the average capacity (total capacity divided by the number of generators).

The first important observation relates to the first two datasets. In the 0.5 M C / 2000 G, the **Threads** version has more than a 150-fold speedup for 32 threads while the 0.5 M C / 64 G dataset only reaches a 70-fold speedup. Having only 64 generators instead of 2000 makes it harder to scale the program since there is only a few generators to distribute among threads. However, it must be noted that the 0.5 M C / 64 G dataset performs proportionally faster when using 1 thread than the 2000 G dataset since that one thread has a small number of generators to choose from when processing consumers. The rule that assigns generators to consumers (lines 26-32 in Fig. 8.13) needs to iterate through all the generators to find one with enough power to handle the consumer, therefore, a smaller number of generators is a clear advantage over having many generators in the case of using just 1 thread.

The second important observation relates to the last two datasets where we experimented with a variable capacity for the generators. For the Low Variability dataset, the consumers have identical capacities, while in the High Variability dataset, generators have a more variable capacity, which should make it harder for the algorithm to find a valid generator/consumer assignment. Our results show exactly that: the Low Variability shows a small difference between the **Reg-**

```
type linear thread-connected-to(thread, generator, consumer, int).                  // Predicate declaration
type linear thread-connected-to-list(thread, generator, list consumer).
type linear thread-capacity(thread, generator, int, int).
type linear thread-total-capacity(thread, int, int).
type linear start(generator).

start(G), !generator-id(G, Id)
   -o set-thread(G, Id % @threads).

just-moved(G), capacity(G, Total, Used), thread-total-capacity(T, TotalCapacity, TotalUsed)
   -o thread-connected-to-list(T, G, []), thread-capacity(T, G, Total, Used),
      thread-total-capacity(T, Total + TotalCapacity, Used + TotalUsed).

fails(G, Fails), Fails > maxfails
   -o random-reconnect(G).

thread-capacity(T, G, Total, Used), thread-total-capacity(T, TotalCapacity, TotalUsed),
random-reconnect(G), thread-connected-to-list(T, G, L), L <> [], C = nth(L, randint(length(L))),
thread-connected-to(T, G, C, Power), NewUsed = Used - Power
   -o fails(G, 0), thread-capacity(T, G, Total, NewUsed),
      thread-total-capacity(T, TotalCapacity, TotalUsed - Power),
      thread-connected-to-list(T, G, remove(L, C)), disconnect(C, G).

random-reconnect(G) -o fails(G, 0).

connect(G, C, Power), thread-total-capacity(T, TotalCapacity, TotalUsed),
thread-capacity(T, G, Total, Used), fails(G, Fails), thread-connected-to-list(T, G, L),
NewUsed = Used + Power, NewUsed <= Total
   -o thread-capacity(T, G, Total, NewUsed),
      thread-total-capacity(T, TotalCapacity, TotalUsed + Power),
      fails(G, max(Fails - 1, 0)), thread-connected-to(T, G, C, Power),
      thread-connected-to-list(T, G, [C | L]).

connect(G, C, Power), thread-capacity(T, G, Total, Used), Used + Power > Total, fails(G, Fails)
   -o thread-capacity(T, G, Total, Used), disconnect(C, G), fails(G, Fails + 1).

!power(C, Power), reconnect(C), disconnected(C), thread-total-capacity(T, TotalCapacity, TotalUsed),
TotalNewUsed = TotalUsed + Power, NewUsed <= TotalCapacity,
thread-capacity(T, G, Total, Used), thread-connected-to-list(T, G, ConnectList),
NewUsed = Used + Power, NewUsed <= Total
   -o connected(C, G), thread-capacity(T, G, Total, NewUsed),
      thread-connected-to-list(T, G, [C | ConnectList]), thread-connected-to(T, G, C, Power),
      thread-total-capacity(T, TotalCapacity, TotalNewUsed).

!generators(C, L), !power(C, Power), reconnect(C), disconnected(C), G = nth(L, randint(num-generators))
   -o connected(C, G), connect(G, C, Power).

disconnect(C, G), connected(C, G)
   -o disconnected(C), reconnect(C).

thread-total-capacity(T, 0, 0).                                                     // Initial facts
fails(G, 0).  disconnected(C).  reconnect(C).  !generators(C, all-generators).  start(G).
```

Figure 8.13: *LM code for the optimized PowerGrid program.*

ular and **Threads** version, while in the High Variability dataset, the **Threads** version is much faster than the **Regular** version. However, for the High Variability dataset, we were expecting a speedup that was closer to the 0.5 M C / 2000 G dataset, since the number of generators is much higher. Furthermore, if we compare the run times of the **Regular** and **Threads** version when using 1 thread, we notice that the **Threads** version is actually slower. As noted before, this may be due to the fact that the LM rule for assigning generators to consumers needs to perform a

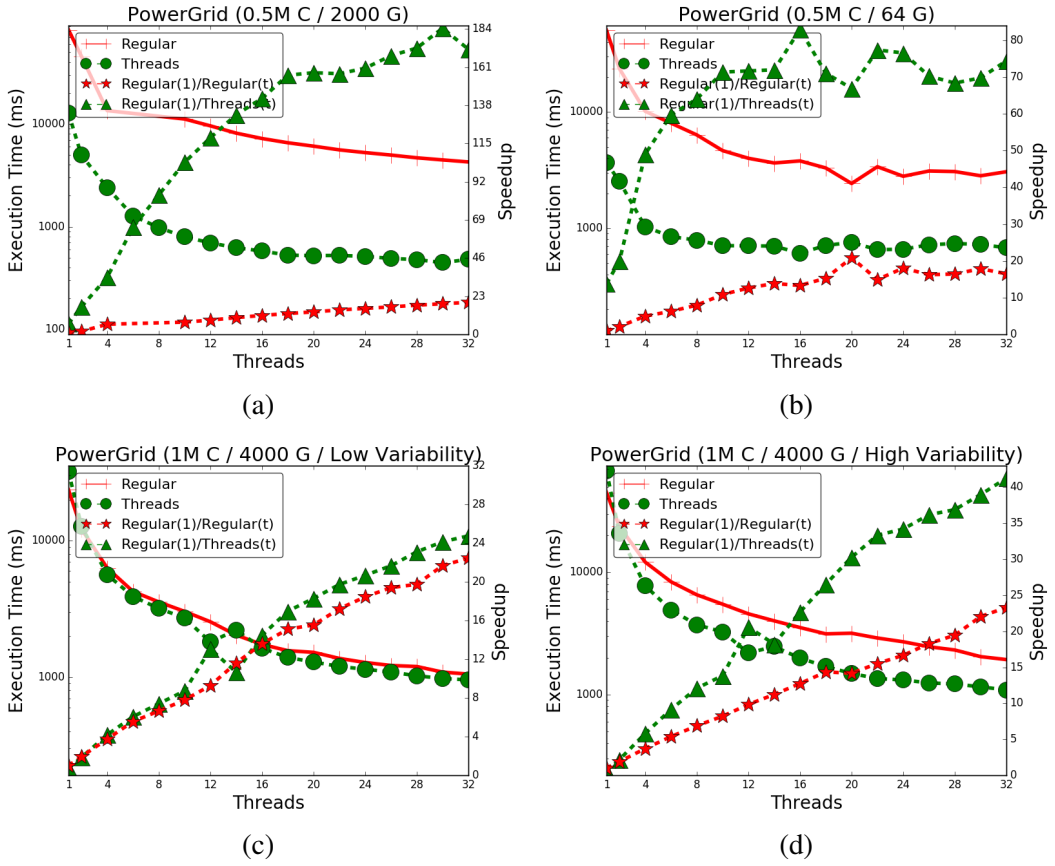


Figure 8.14: *Measuring the performance of the PowerGrid program when using thread facts.*

linear scan on the available generators to find a suitable generator which then negatively impacts performance. This is a clear drawback of the logic programming model that could potentially be solved by maintaining a sorted list of `thread-capacity` facts.

In Table 8.2, we present several fact statistics that compare the **Regular** version with the **Threads** version when executing with multiple threads. The **# Derived** column indicates the number of derived facts, **# Deleted** indicates the number of retracted facts, while **# Final** is the number of facts in the database after the program terminates. The table results clearly show that using thread-based facts results in a decrease in the number of generated facts, which is more significant in the 0.5M C / 2000 G dataset (10-fold reduction). The table also explains why this dataset performs much better than the 1M C / 4000 G High Variability dataset, which only sees a 2-fold reduction in derived facts.

When comparing the number of facts derived when using a different number of threads, the overall trend indicates that having more threads slightly increases the number of derived facts. This is especially true for the 0.5 M C / 64 G datasets, where twice as many facts are generated when using 32 threads when compared to 1 thread.

8.3.3 Splash Belief Propagation

Approximation algorithms can obtain significant benefits from using customized scheduling policies since they follow important statistical properties and thus can trade correctness for faster convergence. An example of such algorithm is the Loopy Belief Propagation (LBP) [MWJ99]. LBP is an approximate inference algorithm used in graphical models with cycles which employs a sum-product message passing algorithm where nodes exchange messages with their immediate neighbors and apply some computations to the messages received.

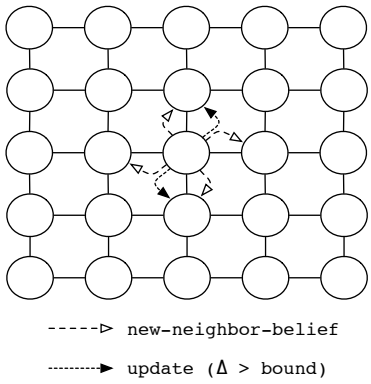


Figure 8.15: *LBP communication patterns.* `new-neighbor-belief` *facts are sent to the neighborhood when the node's belief value is updated. If the new value sent to the neighbor differs significantly from the value sent before the current the round, then an* `update` *fact is also sent (to the node above and below in this case).*

LBP is an algorithm that maps well to the graph-based model of LM. The original algorithm computes the belief of all nodes using several iterations with synchronization between iterations. However, it is possible to avoid the synchronization step, if we take advantage of the fact that LBP

will converge even when using an asynchronous approach. So, instead of computing the belief iteratively, we keep track of all messages sent/received (and overwrite them when we receive a new one) and recompute the belief asynchronously. Figure 8.15 presents the communication patterns of the program, while Fig. 8.16 presents the LM code for the asynchronous version of LBP.

Dataset	Threads	# Derived		# Deleted		# Final	
		Regular	Threads	Regular	Threads	Regular	Threads
0.5M C / 2000 G	1	49.7M	8%	48.2M	5%	2.5M	2.5M
	2	48.5M	8%	47.7M	5%	2.5M	2.5M
	4	49.4M	8%	47.9M	5%	2.5M	2.5M
	8	49.4M	8%	47.9M	5%	2.5M	2.5M
	16	50.8M	8%	49.3M	5%	2.5M	2.5M
	24	49.8M	8%	48.3M	5%	2.5M	2.5M
	32	49.3M	9%	47.8M	6%	2.5M	2.5M
0.5M C / 64 G	1	20.2M	20%	18.5M	13%	2.5M	2.5M
	2	19.9M	21%	18.4M	14%	2.5M	2.5M
	4	20.7M	22%	18.5M	15%	2.5M	2.5M
	8	19.9M	28%	18.4M	22%	2.5M	2.5M
	16	19.8M	34%	18.3M	28%	2.5M	2.5M
	24	19.7M	42%	18.2M	38%	2.5M	2.5M
	32	19.9M	45%	18.4M	40%	2.5M	2.5M
1M C / 4000 G Low Variability	1	9.4M	80%	6.4M	70%	5.1M	5.1M
	2	9.4M	79%	6.4M	70%	5.1M	5.1M
	4	9.4M	79%	6.4M	70%	5.1M	5.1M
	8	9.4M	80%	6.4M	71%	5.1M	5.1M
	16	9.4M	80%	6.4M	71%	5.1M	5.1M
	24	9.4M	81%	6.3M	72%	5.1M	5.1M
	32	9.3M	80%	6.3M	71%	5.1M	5.1M
1M C / 4000 G High Variability	1	16.4M	53%	13.4M	43%	5.1M	5.1M
	2	16.5M	53%	13.5M	43%	5.1M	5.1M
	4	16.5M	53%	13.4M	43%	5.1M	5.1M
	8	16.5M	54%	13.5M	43%	5.1M	5.1M
	16	16.5M	53%	13.5M	43%	5.1M	5.1M
	24	16.5M	54%	13.5M	44%	5.1M	5.1M
	32	16.5M	54%	13.5M	44%	5.1M	5.1M

Table 8.2: *Measuring the reduction in derived facts when using thread-based facts. The first two Threads columns show the ratio of the* ***Threads*** *version over the* ***Regular*** *version. Percentages less than 100% means that the* ***Threads*** *version produces fewer facts.*

```
type list float belief.                                          // Type declaration.

type potential(node, belief).                                  // Predicate declaration
type edge(node, node).
type linear neighbor-belief(node, node, belief).
type linear new-neighbor-belief(node, node, belief).
type linear sent-neighbor-belief(node, node, belief).
type linear check-residual(node, float, node).
type linear belief(node, belief).
type linear update-messages(node, belief).
type linear update(node).

neighbor-belief(A, B, Belief),                  // Rule 1: update neighbor belief value
new-neighbor-belief(A, B, NewBelief)
   -o neighbor-belief(A, B, NewBelief).

check-residual(A, Residual, B),                               // Rule 2: check residual
Residual > bound
   -o update(B).

check-residual(A, _, _) -o 1.                                 // Rule 3: check residual

update-messages(A, NewBelief),  // Rule 4: compute belief to be sent to a neighbor node
   -o {B, OldIn, OldOut, Cavity, Convolved, OutMessage, Residual |
         !edge(A, B),
         neighbor-belief(A, B, OldIn),
         sent-neighbor-belief(A, B, OldOut),
         Cavity = normalize(divide(NewBelief, OldIn)),
         Convolved = normalize(convolve(global-potential, Cavity)),
         OutMessage = damp(Convolved, OldOut, damping)
         Residual = residual(OutMessage, OldOut)
         -o check-residual(A, Residual, B),
            new-neighbor-belief(B, A, OutMessage),
            neighbor-belief(A, B, OldIn),
            sent-neighbor-belief(A, B, OutMessage)}.


update(A), update(A) -o update(A).        // Rule 5: prune redundant update operations

update(A),                                          // Rule 6: initiate update operation
!potential(A, Potential),
belief(A, MyBelief)
   -o [sum Potential => Belief; B, Belief |
         neighbor-belief(A, B, Belief) -o
         neighbor-belief(A, B, Belief) ->
         Normalized = normalizestruct(Belief),
         update-messages(A, Normalized), belief(A, Normalized)].
```

Figure 8.16: *LM code for the asynchronous version of the Loopy Belief Propagation problem.*

Belief values are arrays of floats and are represented by `belief/2` facts. The first rule (lines 13-15) updates a given neighbor belief whenever a new belief value is received. This is the highest priority rule since we want to update the neighbor beliefs before doing anything else. In order to store the belief values of the neighbor nodes, we use `neighbor-belief/3` facts, where the second argument is the neighbor address and the third argument is the belief value.

The last two rules (lines 37-47) update the belief value of a node. An `update` fact starts the process. The first rule (line 38) simply removes redundant `update` facts and the second rule (lines 40-47) performs the belief update by aggregating all the neighbor belief values. The aggregate in lines 43-47 also derives copies of the neighbors beliefs that need to be consumed in order to compute the belief value that is going to be sent to the target neighbor. The aggregate uses a custom accumulator that takes two arrays and adds the floating point numbers at each index of the array.

The rule in lines 23-35 iterates through the neighbor belief values and sends new belief values by performing the appropriate computations on the new belief value of the current node and on the belief value sent previously. For each neighbor update, we also check in lines 17-21 if the change in belief values is greater than bound (a program constant) and then force the neighbor nodes to update their belief values by deriving `update(B)`. This allows neighbor nodes to use updated neighbor values and recompute their own belief values using more up-to-date information. The computation of belief values will then start to converge to their true belief values, independently of the node scheduling used.

However, if we prioritize nodes that receive new neighbor belief values with a larger `Residual` then we may converge faster. Figure 8.17 shows the fourth rule modified with a `add-priority` fact, which increases the priority of neighbor nodes when the source node has large changes in its belief value.

```
1  update-messages(A, NewBelief),  // Rule 4: compute belief to be sent to a neighbor node
2     -o {B, OldIn, OldOut, Cavity, Convolved, OutMessage, Residual |
3           !edge(A, B),
4           neighbor-belief(A, B, OldIn),
5           sent-neighbor-belief(A, B, OldOut),
6           Cavity = normalize(divide(NewBelief, OldIn)),
7           Convolved = normalize(convolve(global-potential, Cavity)),
8           OutMessage = damp(Convolved, OldOut, damping)
9           Residual = residual(OutMessage, OldOut)
10          -o check-residual(A, Residual, B),
11             new-neighbor-belief(B, A, OutMessage),
12             neighbor-belief(A, B, OldIn),
13             add-priority(B, Residual),
14             sent-neighbor-belief(A, B, OutMessage)}.
```

Figure 8.17: *Extending the LBP program with priorities.*

The proposed asynchronous approach has shown to be an improvement over the synchronous version because it leads to faster convergence time. An improved evaluation strategy is the Splash Belief Propagation (SBP) [GLG09], where belief values are computed asynchronously by first building a tree and then by updating the beliefs of each node twice, first from the leaves to the root and then from the root to the leaves. These *splash trees* are built by starting at a node whose

belief changed the most in the last update. The trees must be built iteratively until convergence is achieved.

In an environment with T threads, it is then possible to build T splash trees concurrently. First, we partition the nodes into T regions and then assign each region to a thread. A thread is then responsible for iteratively building splash trees on that region until convergence is reached. Fig. 8.18 shows a grid of nodes that has been partitioned in two regions where splash trees will be built. To build a splash tree, a thread starts from the highest priority node (the tree's root) from its region and then performs a breadth-first search from that node to construct the rest of the tree. The belief values are then computed in order.

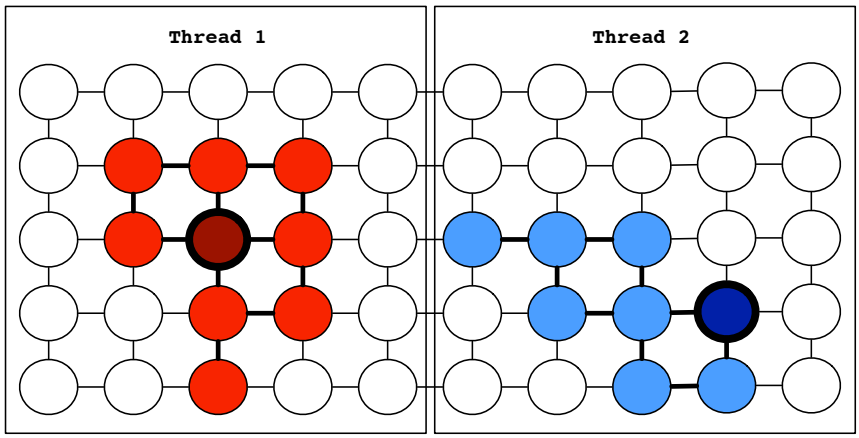


Figure 8.18: *Creating splash trees using two threads. The graph is partitioned into two regions and each thread is able to build separate splash trees starting from the highest priority node.*

The LM implementation for SBP is shown in Fig. 8.19. First, in lines 14-18, we partition the nodes into regions using `set-thread` and then we start the creation of the first splash tree (line 18) by deriving `start-tree(T)`. The remaining phases of the algorithm are explained next.

Tree building: Starts after the rule in lines 20-21 is derived. Since the thread always picks the highest priority node, we start by adding that node to the list that represents the tree. In lines 24-27, we use an aggregate to gather all the neighbor nodes that have a positive priority (due to a new belief update) and are in the same thread. Nodes are collected into list `L` and appended to list `Next` (line 27).

First phase: When the number of nodes in the tree reaches a certain limit, a `first-phase` is generated to update the beliefs of all nodes in the tree (line 30). As the nodes are updated, starting from the leaves and ending at the root, an `update` fact is derived to update the belief values (lines 37 and 39).

Second phase: Performs the computation of beliefs from the root to the leaves and the belief values are updated a second time (lines 42 and 44).

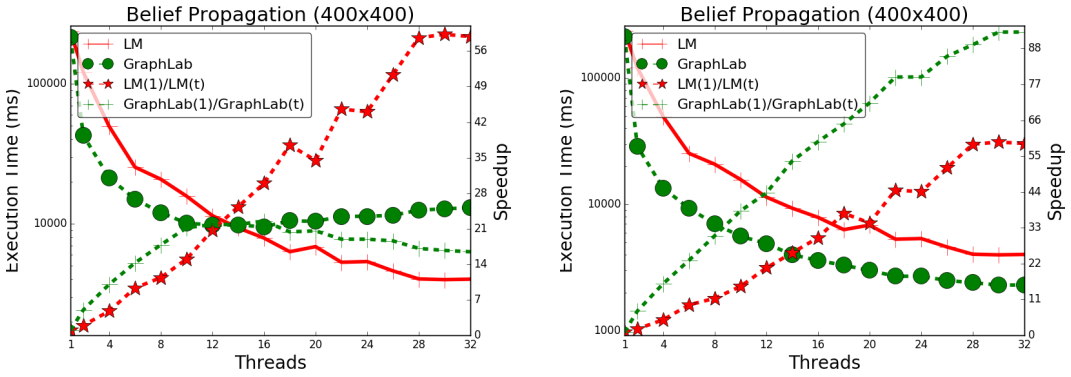
SBP is also implemented in GraphLab [LGK+10], a C++ framework for writing machine learning algorithms. GraphLab provides different schedulers that change how machine learning algorithms are computed and one of the available schedulers is the **splash** scheduler, which

```
type list node tree.

type linear partitioning(thread, int).                    // Number of nodes to receive.
type linear start-tree(thread).
type linear new-tree(thread, tree, tree).
type linear expand-tree(thread, tree).
type linear first-phase(thread, tree, tree).
type linear second-phase(thread, tree).
type linear start(node).

start(A).
partitioning(T, @world / @threads).                       // Move @world/@threads nodes.

!coord(A, X, Y), start(A)                                          // Moving this node.
   -o set-thread(A, grid(X, Y)).
just-moved(A), partitioning(T, Left)                    // Thread received another node.
   -o partitioning(T, Left - 1).
partitioning(T, 0) -o start-tree(T).

start-tree(T), priority(A, P), P > 0.0                                 // Tree building
   -o priority(A, P), expand-tree(T, [A], []).
expand-tree(T, [A | All], Next)
   -o thread-id(A, Id),
      [collect => L ; | !edge(A, L), ~ L in All, ~ L in Next, priority(L, P), P > 0.0,
         thread-id(L, Id2), Id1 = Id2 -o priority(L, P), thread-id(L, Id2) ->
         new-tree(T, [A | All],
            if len(All) + 1 >= maxnodes then [] else Next ++ L end)].

new-tree(T, [A | All], [])
   -o schedule-next(A), first-phase(T, reverse([A | All]), [A | All]).
new-tree(T, All, [B | Next])
   -o schedule-next(B), expand-tree(T, [B | All], Next).

first-phase(T, [A], [A]), running(T, A)                              // First phase
   -o running(T, A), update(A), remove-priority(A), start-tree(T).
first-phase(T, [A, B | Next], [A]), running(T, A)
   -o running(T, A), update(A), schedule-next(B), second-phase(T, [B | Next]).
first-phase(T, All, [A, B | Next]), running(T, A)
   -o running(T, A), update(A), schedule-next(B), first-phase(T, All, [B | Next]).

second-phase(T, [A]), running(T, A)                                 // Second phase
   -o running(T, A), update(A), remove-priority(A), start-tree(T).
second-phase(T, [A, B | Next]), running(T, A)
   -o running(T, A), update(A), schedule-next(B), second-phase(T, [B | Next]).
```

Figure 8.19: *LM code for the Splash Belief Propagation program.*

implements the scheduling described above. For comparison purposes with LM, we also experimented with two other GraphLab schedulers: **fifo**, a first-in first-out scheduler and **multiqueue**, a first-in first-out scheduler that also allows for *work stealing* (we used 1 queue per thread).

The default LM scheduling policy of LM is somewhat similar to both the **fifo** and the **multi-**



(a) *Comparing the scalability of LM and of GraphLab's* ***fifo*** *scheduler*

(b) *Comparing the scalability of LM and of GraphLab's* ***multiqueue*** *scheduler*

Figure 8.20: *Comparing the scalability of LM against GraphLab's (**fifo** and **multiqueue** schedulers.*

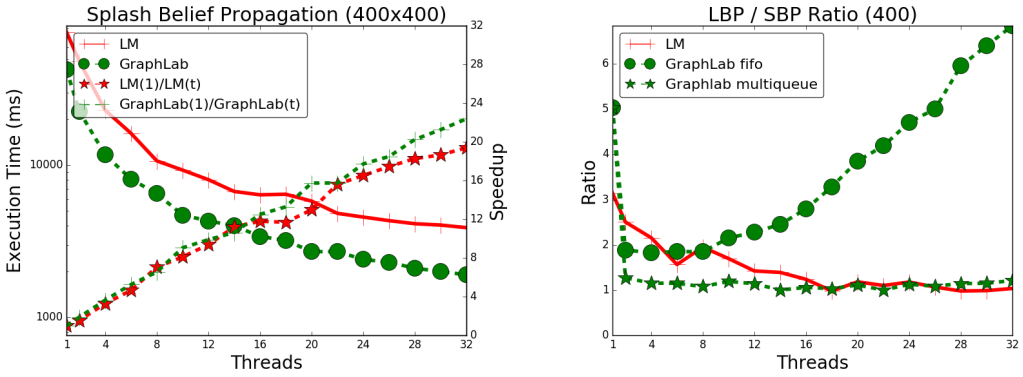
queue schedulers, however, since LM also implements node stealing by default, the **multiqueue** scheduler will be our main focus. We measured the run time of LBP and SBP for both LM and GraphLab. For SBP, we used splash trees of 100 nodes in both systems.

Fig. 8.20 compares the performance of the LBP program in LM against both the **fifo** and **multiqueue** schedulers. The results show that GraphLab's **fifo** scheduler performance deteriorates with more 10 threads, while both LM and **multiqueue** tend to scale well and have a similar behavior. For 1 thread, LM is about 1.5 times slower than GraphLab but that ratio increases to about to about 2 once the number of threads increases. We think this is because the LBP program is sensitive to scheduling policies and the **multiqueue** scheduler is better than LM's default scheduling policy.

The scalability and run time of the SBP program is compared in Fig. 8.21(a). When compared to LM, GraphLab is about twice as fast and that advantage is constant across the number of threads since both systems have a similar scalability. Finally, in Fig. 8.21(b), we compare the performance of LBP program against the performance of SBP by calculating $LBP(t)/SBP(t)$, where $LBP(t)$ is the run time of LBP for t threads, while $SBP(t)$ is the run time of SBP for t threads. For LM, SBP improves noticeably over LBP but that advantage is reduced as the number of threads increases. The same behavior is also seen in GraphLab's **multiqueue** scheduler but since this scheduler works so well under a shared memory setting, the advantages of the **splash** scheduler are reduced. LM is able to improve its performance with SBP since it is a slower language and reducing the number of facts derived provides an improved performance.

8.4 Modeling the Operational Semantics in LM

The introduction of thread-based facts allows for explicit, but declarative, parallelism in a language that used mostly implicit parallelism. This introduces issues when attempting to prove the correctness of programs because the behavior of threads and the scheduling strategy is now also part of the program's logic. Some of this behavior is hidden from programs because it is part of



(a) *Measuring and comparing the performance of SBP in LM and the same program using GraphLab's* ***splash*** *scheduler.*

(b) *Comparing the GraphLab's* ***splash*** *scheduler against the* ***fifo*** *and the* ***multi-queue*** *schedulers.*

Figure 8.21: *Evaluating the performance of SBP over LBP in LM and GraphLab.*

how coordination facts and thread scheduling works on the virtual machine.

Consider the SBP program in Fig. 8.19 where in lines 14-18 the graph of nodes is partitioned into regions. In order to prove the correct partitioning, we need to know how the VM initially randomly assigns nodes to threads and also how coordination facts `set-thread` and `just-moved` are used by the VM. Fortunately, since linear logic is the foundation of LM, it is possible to model the semantics of LM by using LM rules. In Chapter 6, we have seen that threads and nodes transition between different states during execution and now we are going to model that. We first define the following node facts:

- `inactive(node A)`: Fact present on nodes that are not currently running on a thread. Facts `running(T, A)` and `inactive(A)` are mutually exclusive.
- `owner(node A, thread T)`: Fact that indicates the thread `T` that currently owns node `A`.
- `available-work(node A, bool F)`: Fact that indicates if node `A` has new facts to be processed.

In terms of thread facts we have the following:

- `active(thread T)`: Fact exists if thread `T` is currently active.
- `idle(thread T)`: Fact exists if thread `T` is currently idle. Facts `idle(T)` and `active(T)` are mutually exclusive.

Figure 8.22 presents how the operational semantics for a given LM program is modeled using the LM language itself.

First, we define the initial facts: `owner(A, T)` on line 11, which assigns a node to a thread; `available-work(A, F)` on line 12, where `F = true` if node `A` has initial facts, otherwise `F = false`; `is-moving(A)` on line 13 so that all nodes can move between threads; and `active(T)` on line 14 to mark each thread as *active*;

Each program rule is translated as shown in lines 16-21. The original rule was `node-fact(A, Y), other-fact(A, B) -o remote-fact(B), local-fact(A)`, so we have a local derivation of `local-fact(A)` and a remote-derivation of `remote-fact(B)`. In the translation, we update

available-work of node B to true because there is a new derivation for B. The fact running(T, A) is used to ensure that thread T is running on node A. Note that for thread rules we do not need to use running(T, A) on the rule's LHS and the thread running the rule does not even need to have active(T). This enforces the non-deterministic semantics for thread rules.

After the program rules are translated, we have the rule in lines 23-27 which forces thread T to stop running on node A. Here, we use the coordination fact default-priority to update the priority of node A. The thread's state switches to idle(T), while the node's state changes to inactive(A). Note that this rule must appear after the program's rules because the rule priorities are exploited in order to force thread T to derive all the candidate rules for A.

If a thread is idle, then it is able to derive the rule in lines 29-34 in order to select another node for execution. We use a max selector to select the node A with the highest priority Prio. If there is such node, the node changes to running(T, A) and thread T changes to active(T).

Finally, the rule in lines 36-41 allows for threads to steal nodes owned by other threads. If a node is not currently being executed (inactive(A)), can be moved (is-moving(A)), and is owned by another thread (owner(A, Other)), then the thread owner is updated, potentially allowing the previous rule to execute.

We now model several coordination facts presented in Chapter 7 using LM rules. We focus on set-thread, set-priority, just-moved, and schedule-next. The rules are presented in Fig. 8.23 and should be the highest priority rules in LM programs.

We start with the axiom priority(A, initial-priority) and default-priority(A, initial-priority) (lines 7 and 8) to define the initial priorities of nodes. In line 10 we have the rule for the schedule-next coordination fact, which simply re-derives a set-priority but with an infinite priority. Fact set-priority is processed in lines 12-16 by updating the priority values in the priority facts. As explained in Chapter 7, only higher priorities are taken into account.

For the set-thread coordination fact, we have lines 18-28. The first rule applies when the node is currently executing on some thread, forcing the thread to stop executing the node and to derive just-moved(A). In the second rule, node A is not being executed and the owner fact is simply updated to the new thread.

Note that the rules for updating the coordination sensing facts do not require the running predicate in the rule's body, therefore it should not matter which thread does the update as long as it is done. In the VM, and for efficiency reasons, the update is always done by the thread that derives the coordination fact.

8.5 Related Work

As already seen in the previous chapter, there are several programming models such as Galois [NP11], Elixir [PMP12] and Halide [RKBA+13] which allow the programmer to apply different scheduling policies to programs. Unfortunately, these models only reason about the data or program being computed and not about the parallel architecture.

In the logic programming community, there have been some attempts at exposing a low level programming interface in Prolog programs to permit explicit programmer control. An example is the proposal by Casas et al. [CCH07] which exposes execution primitives for AND-parallelism,

```
type linear running(thread, node).
type linear inactive(node).
type linear priority(node, float).
type linear default-priority(node, float).
type linear available-work(node, bool).
type linear active(thread).
type linear idle(thread).
type linear owner(node, thread).
type linear is-moving(node).

owner(A, T).                                          // Initial node assignment.
available-work(A, F).                              // Some nodes have available work.
is-moving(A).                                         // All nodes can be stolen.
active(T).                                             // All threads are active.

node-fact(A, Y),                                        // Program rules go here.
other-fact(A, B),
running(T, A), available-work(B, _)
   -o remote-fact(B), local-fact(A),
      running(T, A),
      available-work(B, true).

active(T), running(T, A), priority(A, Prio),              // Switching to another node.
default-priority(A, DefPrio), available-work(A, T)
   -o inactive(A), priority(A, DefPrio),
      default-priority(A, DefPrio),
      available-work(A, false), idle(T).

[desc => Prio |                                     // Select next node to be processed.
   idle(T), owner(A, T),
   priority(A, Prio), available-work(A, true)]
   -o active(T), owner(A, T),
      running(T, A), available-work(A, false),
      priority(A, Prio).

idle(T), !other-thread(T, Other)                          // Attempt to steal a node.
owner(A, Other), inactive(A),
available-work(A, true),
is-moving(A)
   -o idle(T), owner(A, T), is-moving(A),
      inactive(A), available-work(A, true).
```

Figure 8.22: *Modeling the operational semantics as a LM program. The underlined code represents how an example rule* `node-fact(A, Y), other-fact(A, B) -o remote-fact(B), local-fact(A)` *needs to be translated for modeling the semantics.*

allowing for different scheduling policies. Compared to LM, this approach offers a more fine grained control to parallelism but has limited support for reasoning about thread state.

```
type linear is-static(node).
type linear is-moving(node).
type linear set-priority(node, float).
type linear just-moved(node).
type linear move-to-thread(node, thread).

priority(A, initial-priority).
default-priority(A, default-priority).

schedule-next(A) -o set-priority(A, +00).

set-priority(A, P1), priority(A, P2), P2 < P1
   -o priority(A, P1).

set-priority(A, P1), priority(A, P2), P2 >= P1
   -o priority(A, P2).

running(T, A), set-thread(A, T),
available-work(A, _), is-moving(A)
   -o available-work(A, true),
      inactive(A), is-static(A),
      just-moved(A).

inactive(A), set-thread(A, T),
owner(A, TOld), is-moving(A),
available-work(A, _)
   -o is-static(A), owner(A, T), just-moved(A),
      available-work(A, true).
```

Figure 8.23: *Modeling the operational semantics for coordination facts as a LM program.*

8.6 Chapter Summary

In this chapter, we have extended the LM language with a declarative mechanism for reasoning about the underlying parallel architecture. LM programs can be first written in a data-driven fashion and then optimized by reasoning about the state of threads, enabling the move from implicit parallelism to some form of declarative explicit parallelism. We have presented four programs that showcase the potential of the new mechanism and several experimental results that validate our approach.

Chapter 9

Conclusions

In this chapter, we summarize the main contributions of this thesis and suggest several directions for further research.

9.1 Main Contributions

The goal of our thesis was to show the potential of using a forward-chaining linear logic programming language to exploit parallelism in a declarative, efficient and scalable way. For this, we designed and implemented LM, a programming language suitable for writing concurrent algorithms over graph data-structures. LM programs are composed of a set of inference rules that apply over a database of logical facts. Since LM is based on linear logic, facts used in rules may be retracted, making it possible for the programmer to declaratively manage structured state.

Concurrency is achieved by partitioning the database across a graph data structure and then forcing rule derivation to happen at the node level. The use of graphs allows LM to solve the task granularity problem that is common in implicitly parallel languages. By localizing computation to nodes, it is possible to group nodes together as sub-graphs that can be processed independently. Even if certain sub-graph partitioning proves to have a poor task balance, nodes of sub-graphs can be moved into other sub-graphs, allowing for easy load balance between processing cores.

We introduced coordination facts in LM to allow the programmer to fine-tune and optimize their declarative programs. This is possible due to the existence of linear facts, which make it possible for different scheduling decisions to have an effect on program computation. Without them, a logic program would always compute the same result. Coordination facts are also another way to combat the granularity problem since they allow the programmer to control how nodes are grouped together.

We also introduced explicit parallelism in LM by adding support for thread facts. Thread facts are facts stored at the thread level and allow the programmer to write rules that reason about thread state. This opens new opportunities for program optimization and improved parallelism because programs are aware of the existence of threads. Furthermore, the availability of both thread and coordination facts allows and is convenient for implementing more sophisticated scheduling parallel algorithms.

We now highlight in more detail the main contributions of this dissertation:

Structured State Since LM is based on linear logic, LM enables programs to manage state in a structured manner. Due to the restriction over the inference rules, rules are derived independently on different nodes of the graph data structure, which makes it possible to run LM programs in parallel.

Implementation Writing efficient implementations of declarative languages is challenging, especially when adding support for parallel execution. In this thesis, we have shown a compilation strategy and memory layout organization for LM programs, which allows programs to run less than one order of magnitude slower than hand-written C++ programs. We also described how the LM runtime supports multi core execution by partitioning the graph among threads. To the best of our knowledge, LM is the fastest linear logic based programming language available.

Semantics and Abstract Machine We showed how LM semantics are specified in order to allow concurrent programs. We demonstrated how the language is based upon the sequent calculus of linear logic and we specified a low level abstract machine that closely resembles the real implementation. We also proved the soundness of the abstract machine.

Coordination We presented new coordination features that improve the expressive power of the language by making coordination a first class programming construct that is semantically equivalent to regular computation. In turn, this allows the programmer to specify how declarative programs are scheduled by the runtime system, therefore improving overall run time and scalability.

Coordination facts are divided into scheduling and partitioning facts. Scheduling facts change how nodes are scheduled by the system while partitioning facts change how nodes are assigned to threads and how they move between threads. Both these two types of facts are divided into sensing facts (with information about the state of the runtime system) and action facts (which perform actions on the runtime system). The interplay between regular facts, sensing facts and action facts results in faster execution time and improved parallelism because regular facts affect how action facts are derived and, conversely, action facts may affect which regular facts are derived.

The coordination facts do not affect the correctness of programs and are well integrated into proofs since they do not change how rules are scheduled but how nodes are scheduled during execution.

Explicit Parallelism We also introduced the concept of thread facts, which enable LM programs to exploit the underlying architecture by making it possible to reason about the state of threads. Thread facts allow the programmer to escape the default implicit parallelism of LM and allows the implementation of structured scheduling algorithms which require explicit communication between threads. To the best of our knowledge, this is the first time that such paradigm is available in a logic programming language and we are not aware of competing systems that allow the programmer to reason directly about thread state in a structured fashion.

Experimentation We compared LM sequential and multi threaded execution to hand-written sequential C++ programs and against frameworks such as GraphLab and Ligra. We showed how well the LM runtime is able to scale using different programs and datasets. We mea-

sured the run time, scalability and memory usage effects of using coordination facts and their overheads. For thread facts, we analyzed different applications and measured the performance improvements of using explicit parallelism.

In our experiments, we noted that the memory layout of applications, especially the memory allocator, tends to have a significant effect on the overall performance. In modern architectures, good memory locality is as important as having efficient algorithms and in LM this is no different. We experimented with two allocators to analyze how performance and scalability may be affected by using different strategies. It is our belief that it is important to focus on faster sequential execution at the expense of scalability in order to make declarative parallel languages more competitive with sequential programs written in languages such as C++.

9.2 Drawbacks of LM

While LM provides answers to several known problems in implicitly parallel languages, it suffers from several disadvantages:

Graph Of Nodes Even though graph data structures are flexible and can be used to model many interesting problems, they are not always the best abstraction. Using graphs as an abstraction for concurrency requires some problems to be adapted to run on graphs in non intuitive ways. For instance, in the N-Queens problem in Section 7.8.2, we had to map the chess board into a grid of nodes and then parallelize the program by considering each square as a unit that builds solutions to the problem. In general, the graph model of LM is suitable for irregular algorithms but not as suitable for regular algorithms such as numerical algorithms or algorithms that don't use graphs as their main data structure.

Thread Reasoning Coordination facts and thread-based facts come with a cost when used to create complex scheduling algorithms because reasoning with coordinated programs requires that the programmer understands the semantics of the underlying virtual machine. However, we think it will be impossible to avoid this issue unless the runtime system performs all the optimizations, leaving no control to the programmer since optimization always requires some knowledge about the underlying hardware and software. In LM, the programmer pays a lower cost because the semantics of coordination provide a higher level of abstraction. Lastly, it remains to be shown if LM provides the right kind of abstraction for writing such programs.

Modularity Program composition is still an unsolved problem in linear logic programming since it is hard to compose programs that manipulate the same state. Martens [Mar15] introduces the concept of staged logic programming which allows sets of logical rules to be grouped into stages. Each stage is computed separately and up to quiescence, allowing stages to have precedence over others. However, it still remains to be studied how separate rules can be grouped together without introducing conflicts.

9.3 Future Work

While much progress has been achieved with this thesis, many new research avenues have been opened with this work. We now enumerate further research goals that should be interesting to pursue.

Faster Implementation LM is still not competitive enough to replace efficient parallel frameworks such as Ligra. LM is a programming language on its own right and thus requires more engineering effort to be competitive with frameworks implemented in languages such as C or C++. Better compilation and runtime systems will be required in order to reduce the overhead even further, especially as it relates to memory usage.

Aggressive code analysis should be employed to prove invariants about predicates and introduce more specialized data structures for storing logical facts. The goal should be to recover more of the *imperative flavor* that is present in linear logic programs in order to make them more efficient. The restrictions of LM rules that make concurrency possible makes this task harder since there is an inherent tension between concurrency and execution speed since concurrency implies communication. However, local node computation has still some room for improvement.

Provability We need automated tools for reasoning about correctness and termination of programs. While we have shown that writing informal proofs is relatively easy because programs tend to be small, automated proof systems will increase the faith that programs will work correctly. Ideally, the programmer should be able to write invariants about the program and the compiler should be able to prove if such invariants are or are not being met with the given inference rules.

Expressiveness Although LM programs are expressive, some work must be done in order to reduce the restrictions on LM rules and allow for more programmer freedom. LM currently only allows rules where the LHS refers to the same node, however, it should be possible to allow rules that use facts from different nodes. The use of linear logic facts makes this hard because we need to ensure that a linear fact is used only once, therefore the compiler should generate code to enforce this, probably through the use of transactions. In the CHR community, Lam et al. [LC13] have developed an encoding for distributed rules using at most one immediate neighbor into rules that run locally. It should be relatively straightforward to provide a similar encoding for LM and then assess how performance is affected by such encoding.

Implicit and Explicit Parallelism We need more applications that take advantage of the mixed parallelism that is available with thread facts. We feel that this paradigm needs to be further explored in order to make it possible to write new scheduling and parallel algorithms that could be compiled to efficient code in other application areas. Furthermore, mechanized proofs about such algorithms could then be automatic, improving the correctness and reliability of parallel algorithms.

9.4 Final Remark

We argue that our work makes LM the ideal framework for prototyping new (correct) graph algorithms since LM programs tend to be relatively short and the programmer only needs to reason about the state of the graph, without the need to understand how the framework must be used to express the intended algorithms. Furthermore, the addition of coordination facts and thread facts help the programmer exploit the underlying parallel architecture in order to create better programs that take advantage of those architectures without radically changing the underlying parallel algorithm. Finally, the good performance of the LM system allows programs to run reasonably fast when executed on multi core systems.

Appendix A

Sequent Calculus

$$\frac{\Psi;\Gamma;\Delta \vdash A \quad \Psi;\Gamma;\Delta' \vdash B}{\Psi;\Gamma;\Delta,\Delta' \vdash A \otimes B}\otimes R \qquad \frac{\Psi;\Gamma;\Delta,A,B \vdash C}{\Psi;\Gamma;\Delta,A \otimes B \vdash C}\otimes L$$

$$\frac{\Psi;\Gamma;\Delta,A \vdash B}{\Psi;\Gamma;\Delta \vdash A \multimap B} \multimap R \qquad \frac{\Psi;\Gamma;\Delta \vdash A \quad \Psi;\Gamma;\Delta',B \vdash C}{\Psi;\Gamma;\Delta,\Delta',A \multimap B \vdash C} \multimap L$$

$$\frac{\Psi;\Gamma;\Delta,A \vdash C}{\Psi;\Gamma;\Delta,A \,\&\, B \vdash C}\,\&L_1 \qquad \frac{\Psi;\Gamma;\Delta,B \vdash C}{\Psi;\Gamma;\Delta,A \,\&\, B \vdash C}\,\&L_2 \qquad \frac{\Psi;\Gamma;\Delta \vdash A \quad \Psi;\Gamma;\Delta \vdash B}{\Psi;\Gamma;\Delta \vdash A \,\&\, B}\,\&R$$

$$\frac{\Psi;\Gamma;\cdot \vdash A}{\Psi;\Gamma;\cdot \vdash\, !A}\, !R \qquad \frac{\Psi;\Gamma,A;\Delta \vdash C}{\Psi;\Gamma;\Delta,!A \vdash C}\, !L \qquad \frac{\Psi;\Gamma,A;\Delta,A \vdash C}{\Psi;\Gamma,A;\Delta \vdash C}\,\mathsf{copy}$$

$$\frac{}{\Psi;\Gamma;\cdot \vdash \mathbf{1}}\,\mathbf{1}R \qquad \frac{\Psi;\Gamma;\Delta \vdash C}{\Psi;\Gamma;\Delta,\mathbf{1} \vdash C}\,\mathbf{1}L$$

$$\frac{}{\Psi;\Gamma;A \vdash A}\,id_A$$

$$\frac{\Psi,m:\tau;\Gamma;\Delta \vdash A\{m/n\}}{\Psi;\Gamma;\Delta \vdash \forall_{n:\tau}.A}\,\forall R \qquad \frac{\Psi \vdash M:\tau \quad \Psi;\Gamma;\Delta,A\{M/n\} \vdash C}{\Psi;\Gamma;\Delta,\forall_{n:\tau}.A \vdash C}\,\forall L$$

$$\frac{\Psi \vdash M:\tau \quad \Psi;\Gamma;\Delta \vdash A\{M/n\}}{\Psi;\Gamma;\Delta \vdash \exists_{n:\tau}.A}\,\exists R \qquad \frac{\Psi,m:\tau;\Gamma;\Delta,A\{m/n\} \vdash C}{\Psi;\Gamma;\Delta,\exists_{n:\tau}.A \vdash C}\,\exists L$$

$$\frac{\Psi;\Gamma;\Delta \vdash A \quad \Psi;\Gamma;\Delta',A \vdash C}{\Psi;\Gamma;\Delta,\Delta' \vdash C}\,cut_A \qquad \frac{\Psi;\Gamma;\cdot \vdash A \quad \Psi;\Gamma,A;\Delta \vdash C}{\Psi;\Gamma;\Delta \vdash C}\,cut!_A$$

Appendix B

High Level Dynamic Semantics

B.1 Step

$$\frac{\mathsf{run}^{\Gamma_i;\Pi}\ \Delta_i;\Phi \to \Xi';[\Gamma'_1;\cdots;\Gamma'_n];[\Delta'_1;\cdots;\Delta'_n]}{\mathsf{step}\ [\Gamma_1;\cdots;\Gamma_i;\cdots;\Gamma_n];[\Delta_1;\cdots;\Delta_i;\cdots;\Delta_n];\Phi} \ \mathsf{step}$$

$$\Longrightarrow$$

$$[\Gamma_1,\Gamma'_1;\cdots;\Gamma_i,\Gamma'_i;\cdots;\Gamma_n,\Gamma'_n];[\Delta_1,\Delta'_1;\cdots;(\Delta_i-\Xi'),\Delta'_i;\cdots;\Delta_n,\Delta'_n]$$

B.2 Application

$$\frac{\mathsf{m}^{\Gamma}\ \Delta_1 \to A \quad \mathsf{der}^{\Gamma;\Pi}\Delta_2;\Delta_1;\cdot;\cdot;B \to \mathcal{O}}{\mathsf{app}_{\Psi}^{\Gamma;\Pi}\ \Delta_1,\Delta_2;\Pi \to \mathcal{O}}\ \mathsf{app\ rule}$$

$$\frac{\mathsf{app}^{\Gamma;\Pi}_{\Psi,m:M:\tau}\ \Delta;\Pi \to \mathcal{O} \quad \Psi \vdash M:\tau}{\mathsf{app}_{\Psi}^{\Gamma;\Pi}\ \Delta;\Pi \to \mathcal{O}}\ \mathsf{app}\ \forall$$

$$\frac{\mathsf{app}_{\cdot}^{\Gamma;\Pi}\ \Delta;\Pi \to \mathcal{O}}{\mathsf{run}^{\Gamma;\Pi}\ \Delta;R,\Phi \to \mathcal{O}}\ \mathsf{run\ rule}$$

B.3 Match

$$\frac{}{\mathsf{m}^{\Gamma}\ \cdot \to \mathbf{1}}\ \mathsf{m}\mathbf{1}$$

$$\frac{}{\mathsf{m}^{\Gamma}\ p \to p}\ \mathsf{m}p \qquad \frac{}{\mathsf{m}^{\Gamma,p}\ \cdot \to\ !p}\ \mathsf{m}!p$$

$$\frac{\mathsf{m}^{\Gamma}\ \Delta_1 \to A \quad \mathsf{m}^{\Gamma}\ \Delta_2 \to B}{\mathsf{m}^{\Gamma}\ \Delta_1,\Delta_2 \to A \otimes B}\ \mathsf{m}\otimes$$

B.4 Derivation

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;p,\Delta_1;\Omega\to\mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;p,\Omega\to\mathcal{O}}\ \mathsf{der}\ p$$

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1,p;\Delta_1;\Omega\to\mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;!p,\Omega\to\mathcal{O}}\ \mathsf{der}\ !p$$

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;A,B,\Omega\to\mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;A\otimes B,\Omega\to\mathcal{O}}\ \mathsf{der}\ \otimes$$

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;\Omega\to\mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;\mathbf{1},\Omega\to\mathcal{O}}\ \mathsf{der}\ \mathbf{1}$$

$$\frac{}{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi';\Gamma';\Delta';\cdot\to\Xi';\Gamma';\Delta'}\ \mathsf{der\ end}$$

$$\frac{\begin{array}{c}\Pi(\mathsf{agg})=\forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')}\multimap((\lambda x.Cx)\Sigma'\,\&\,(\forall_{\widehat{x},\sigma}.(A\multimap B\otimes\mathcal{R}_{agg}^{(\widehat{v},\Sigma'+\sigma)}))))\\ \mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;\forall_{\widehat{x},\sigma}.(A\multimap B\otimes\mathcal{R}_{agg}^{(\widehat{v},\Sigma+\sigma)})\{\widehat{V}/\widehat{v}\}\{\Sigma/\Sigma'\},\Omega\to\mathcal{O}\end{array}}{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;\mathcal{R}_{agg}^{(\widehat{V},\Sigma)},\Omega\to\mathcal{O}}\ \mathsf{der\ agg}_1$$

$$\frac{\begin{array}{c}\Pi(\mathsf{agg})=\forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')}\multimap((\lambda x.Cx)\Sigma'\,\&\,(\forall_{\widehat{x},\sigma}.(A\multimap B\otimes\mathcal{R}_{agg}^{(\widehat{v},\Sigma'+\sigma)}))))\\ \mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;C\{\widehat{V}/\widehat{v}\}\{\Sigma/\Sigma'\},\Omega\to\mathcal{O}\end{array}}{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;\mathcal{R}_{agg}^{(\widehat{V},\Sigma)},\Omega\to\mathcal{O}}\ \mathsf{der\ agg}_2$$

$$\frac{\mathsf{m}^{\Gamma}\ \Delta_a\to A\quad\mathsf{der}^{\Gamma;\Pi}\Delta_b;\Xi,\Delta_a;\Gamma_1;\Delta_1;B,\Omega\to\mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta_a,\Delta_b;\Xi;\Gamma_1;\Delta_1;A\multimap B,\Omega\to\mathcal{O}}\ \mathsf{der}\ \multimap$$

$$\frac{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;A\{V/x\},\Omega\to\mathcal{O}}{\mathsf{der}^{\Gamma;\Pi}\Delta;\Xi;\Gamma_1;\Delta_1;\forall_x.A,\Omega\to\mathcal{O}}\ \mathsf{der}\ \forall$$

Appendix C

Low Level Dynamic Semantics

C.1 Application

$$\texttt{infer}\ \Delta; R_1, \Phi; \Gamma \quad \mapsto \quad \texttt{apply}\ \cdot; \Delta; \Pi; \Gamma; R \qquad \text{(select rule)}$$

$$\texttt{infer}\ \Delta; \cdot; \Gamma \quad \mapsto \quad \mathsf{next}_{\Gamma;\Delta} \qquad \text{(fail)}$$

$$\texttt{apply}\ \Psi; \Delta; \Pi; \Gamma; \forall_{x:\tau}.A \quad \mapsto \quad \texttt{apply}\ \Psi, x : \_ : \tau; \Delta; \Pi; \Gamma; A \qquad \text{(open rule)}$$

$$\texttt{apply}\ \Psi; \Delta; \Pi; \Gamma; A \multimap B \quad \mapsto \quad {}^{\Psi}\blacktriangleright^{\mathsf{m}^\Gamma\ \cdot \to \mathbf{1}}_{A \multimap B}\ (\Delta; \Phi); \cdot; \Gamma; \Delta; A \qquad \text{(init rule)}$$

C.2 Match

$$\begin{array}{l} {}^{\Psi}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to \Omega'}_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma; \Delta, p_1, \Delta''; p(\widehat{x}), \Omega \quad \mapsto \\ \qquad {}^{\mathsf{extend}(\Psi,\theta)}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta', p_1 \to \Omega' \otimes p(\widehat{x}\theta)}_{A \multimap B}\ \mathcal{R}; (\Delta, p_1; \Delta''; p(\widehat{x}); \Omega; \mathsf{extend}(\Psi, \theta)), \mathcal{C}; \Gamma; \Delta, \Delta''; \Omega \\ \qquad\qquad {}_{\mathsf{m}^\Gamma\ \Delta' \to \Omega'} \\ \qquad\qquad (p_1, \Delta'' \prec p(\widehat{x}) \quad \Delta \not\preceq p(\widehat{x})) \quad \text{(match p ok)} \end{array}$$

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to \Omega'}_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma; \Delta; p(\widehat{x}), \Omega \quad \mapsto \quad \triangleleft_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma \quad (\Delta \not\preceq p(\widehat{x})) \qquad \text{(match p fail)}$$

$$\begin{array}{l} {}^{\Psi}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to \Omega'}_{A \multimap B}\ \mathcal{R}; \mathcal{C}; \Gamma; \Delta; !p(\widehat{x}), \Omega \quad \mapsto \\ \qquad {}^{\mathsf{extend}(\Psi,\theta)}\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta' \to \Omega' \otimes !p(\widehat{x}\theta)}_{A \multimap B}\ \mathcal{R}; [\Gamma''; \Delta; !p(\widehat{x}); \Omega; \mathsf{extend}(\Psi, \theta)], \mathcal{C}; \Gamma, p_1, \Gamma''; \Delta; \Omega \\ \qquad\qquad {}_{\mathsf{m}^\Gamma\ \Delta' \to \Omega'} \\ \qquad\qquad (!p_1, \Gamma'' \prec !p(\widehat{x}) \quad \Gamma \not\preceq !p(\widehat{x})) \quad \text{(match !p ok)} \end{array}$$

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;!p(\widehat{x}),\Omega\ \ \mapsto\ \ \triangleleft_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma\quad(\Gamma\not\leq!p(\widehat{x}))\qquad\text{(match !p fail)}$$

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;\mathbf{1},\Omega\ \ \mapsto\ \ \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;\Omega\qquad\text{(match }\mathbf{1}\text{)}$$

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;X\otimes Y,\Omega\ \ \mapsto\ \ \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;X,Y,\Omega\qquad\text{(match }\otimes\text{)}$$

$$\blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma;\Delta;\cdot\ \ \mapsto\ \ \curvearrowright^{\cdot;\cdot}_{\Delta'}\ \Gamma;\Delta;B\qquad\text{(match end)}$$

C.3 Continuation

$$\underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\triangleleft_{A\multimap B}\ \mathcal{R};(\Delta;p_2,\Delta'';p;\Omega),\mathcal{C};\Gamma}\ \ \mapsto\ \ \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta',p_2\to\Omega'\otimes p}_{A\multimap B}\ \underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\mathcal{R};(\Delta,p_2;\Delta'';p;\Omega),\mathcal{C};\Gamma;\Delta;\Omega}\qquad\text{(next p)}$$

$$\underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\triangleleft_{A\multimap B}\ \mathcal{R};(\Delta;\cdot;p;\Omega),\mathcal{C};\Gamma}\ \ \mapsto\ \ \triangleleft_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma\qquad\text{(next frame)}$$

$$\underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\triangleleft_{A\multimap B}\ \mathcal{R};[!p_2,\Gamma'';\Delta;!p;\Omega],\mathcal{C};\Gamma}\ \ \mapsto\ \ \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'\otimes!p_2}_{A\multimap B}\ \underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\mathcal{R};[\Gamma'';\Delta;!p;\Omega],\mathcal{C};\Gamma;\Delta;\Omega}\qquad\text{(next !p)}$$

$$\underset{\mathsf{m}^\Gamma\ \Delta'\to\Omega'}{\triangleleft_{A\multimap B}\ \mathcal{R};[\cdot;\Delta;!p;\Omega],\mathcal{C};\Gamma}\ \ \mapsto\ \ \triangleleft_{A\multimap B}\ \mathcal{R};\mathcal{C};\Gamma\qquad\text{(next !frame)}$$

$$\triangleleft_{A\multimap B}\ (\Delta;\Phi);\cdot;\Gamma\ \ \mapsto\ \ \texttt{infer}\ \Delta;\Phi;\Gamma\qquad\text{(rule fail)}$$

C.4 Derivation

$$\curvearrowright^{\Gamma_1;\Delta_1}_{\Xi}\ \Gamma;\Delta;p,\Omega\ \ \mapsto\ \ \curvearrowright^{\Gamma_1;\Delta_1,p}_{\Xi}\ \Gamma;\Delta;\Omega\qquad\text{(new p)}$$

$$\curvearrowright^{\Gamma_1;\Delta_1}_{\Xi}\ \Gamma;\Delta;!p,\Omega\ \ \mapsto\ \ \curvearrowright^{\Gamma_1,!p;\Delta_1}_{\Xi}\ \Gamma;\Delta;\Omega\qquad\text{(new !p)}$$

$$\curvearrowright^{\Gamma_1;\Delta_1}_{\Xi}\ \Gamma;\Delta;\mathbf{1},\Omega\ \ \mapsto\ \ \curvearrowright^{\Gamma_1;\Delta_1}_{\Xi}\ \Gamma;\Delta;\Omega\qquad\text{(new }\mathbf{1}\text{)}$$

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; A \otimes B, \Omega \;\; \mapsto \;\; \curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; A, B, \Omega \qquad \text{(new } \otimes\text{)}$$

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; \mathcal{R}_{agg}^{(\widehat{V},\cdot;\cdot)}, \Omega \;\; \mapsto \;\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \cdot\to\mathbf{1}}_{\mathsf{agg};\cdot} \cdot;\cdot;\Gamma;\Delta;A \qquad \text{(new agg)}$$
$$\Pi(\mathsf{agg})=\forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \& (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))$$

$$\curvearrowright_{\Xi}^{\Gamma_1;\Delta_1} \Gamma; \Delta; \cdot \;\; \mapsto \;\; \circlearrowleft \Xi; \Gamma_1; \Delta_1 \qquad \text{(rule finished)}$$

C.5 Aggregates

C.5.1 Match

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta, p_1, \Delta''; p, \Omega \;\; \mapsto$$
$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta',p\to\Omega'\otimes p}_{\mathsf{agg};\Sigma} (\Delta, p_1; \Delta''; p; \Omega), \mathcal{C}; \mathcal{P}; \Gamma; \Delta, \Delta''; \Omega \qquad \text{(agg match p ok)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta; p, \Omega \;\; \mapsto \;\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma \qquad \text{(agg match p fail)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \cdot; \mathcal{P}; \Gamma, p_1, \Gamma''; \Delta; !p, \Omega \;\; \mapsto$$
$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'\otimes !p}_{\mathsf{agg};\Sigma} \cdot; [\Gamma''; \Delta; !p; \Omega], \mathcal{P}; \Gamma, p_1, \Gamma''; \Delta; \Omega \qquad \text{(agg match !p ok } \mathcal{P}\text{)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma, p_1, \Gamma''; \Delta; !p, \Omega \;\; \mapsto$$
$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'\otimes !p}_{\mathsf{agg};\Sigma} [\Gamma''; \Delta; !p; \Omega], \mathcal{C}; \mathcal{P}; \Gamma, p_1, \Gamma''; \Delta; \Omega \qquad \text{(agg match !p ok } \mathcal{C}\text{)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta; !p, \Omega \;\; \mapsto \;\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma \qquad \text{(agg match !p fail)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta; X \otimes Y, \Omega \;\; \mapsto$$
$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta; X, Y, \Omega \qquad \text{(agg match } \otimes\text{)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta; \mathbf{1}, \Omega \;\; \mapsto$$
$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta; \Omega \qquad \text{(agg match } \mathbf{1}\text{)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^{\Gamma}\ \Delta'\to\Omega'}_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma; \Delta; \cdot \;\; \mapsto \;\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma \qquad \text{(agg match end)}$$

C.5.2 Stack Transformation

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} \_, f, \mathcal{C}; \mathcal{P}; \Gamma \;\mapsto\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} f, \mathcal{C}; \mathcal{P}; \Gamma \qquad \text{(agg fix rec)}$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} f; \mathcal{P}; \Gamma \;\mapsto\; \Omega_{\mathcal{I}};\Delta;\Xi;\Delta' \curvearrowright^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}_{\mathsf{agg};V::\Sigma} f'; \mathcal{P}'; \Gamma; B\{\Psi(\widehat{x}), V/\widehat{x}, \sigma\} \qquad \text{(agg fix end1)}$$
$$\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}^{(\widehat{v},\Sigma')}_{agg} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}^{(\widehat{v},\sigma::\Sigma')}_{agg}))))$$
$$f' = \mathtt{remove}(f, \Delta')$$
$$\mathcal{P}' = \mathtt{remove}(\mathcal{P}, \Delta')$$
$$V = \Psi(\sigma)$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta;\Xi;\Delta'\end{smallmatrix} \bowtie_{\mathsf{agg};\Sigma} \cdot; \mathcal{P}; \Gamma \;\mapsto\; \Omega_{\mathcal{I}};\Delta;\Xi,\Delta' \curvearrowright^{\Gamma_{N1};\Delta_{N1}}_{\mathsf{agg};V::\Sigma} \cdot; \mathcal{P}'; \Gamma; B\{\Psi(\widehat{x}), V/\widehat{x}, \sigma\} \qquad \text{(agg fix end2)}$$
$$\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}^{(\widehat{v},\Sigma')}_{agg} \multimap ((\lambda x.Cx)\Sigma' \,\&\, (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}^{(\widehat{v},\sigma::\Sigma')}_{agg}))))$$
$$\mathcal{P}' = \mathtt{remove}(\mathcal{P}, \Delta')$$
$$V = \Psi(\sigma)$$

C.5.3 Backtracking

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} (\Delta; p_1, \Delta''; p; \Omega), \mathcal{C}; \mathcal{P}; \Gamma \;\mapsto\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta',p_1\to\Omega'\otimes p}_{\mathsf{agg};\Sigma} (\Delta, p_1; \Delta''; p; \Omega), \mathcal{C}; \mathcal{P}; \Gamma; \Delta; p, \Omega \quad \text{(agg next p } \mathcal{C}\text{)}$$
$$\mathsf{m}^\Gamma\ \Delta'\to\Omega'$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} [p_1, \Gamma''; \Delta; !p; \Omega], \mathcal{C}; \mathcal{P}; \Gamma \;\mapsto\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'\otimes !p}_{\mathsf{agg};\Sigma} [\Gamma''; \Delta; !p; \Omega], \mathcal{C}; \mathcal{P}; \Gamma; \Delta; p, \Omega \quad \text{(agg next !p } \mathcal{C}\text{)}$$
$$\mathsf{m}^\Gamma\ \Delta'\to\Omega'$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} (\Delta; \cdot; p; \Omega), \mathcal{C}; \mathcal{P}; \Gamma \;\mapsto\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma \qquad \text{(agg next frame } \mathcal{C}\text{)}$$
$$\mathsf{m}^\Gamma\ \Delta'\to\Omega'$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} [\cdot; \Delta; !p; \Omega], \mathcal{C}; \mathcal{P}; \Gamma \;\mapsto\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma \qquad \text{(agg next !frame } \mathcal{C}\text{)}$$
$$\mathsf{m}^\Gamma\ \Delta'\to\Omega'$$

$$\begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \triangleleft_{\mathsf{agg};\Sigma} \cdot; [p_1, \Gamma''; \Delta; !p; \Omega], \mathcal{P}; \Gamma \;\mapsto\; \begin{smallmatrix}\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}\\ \Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi\end{smallmatrix} \blacktriangleright^{\mathsf{m}^\Gamma\ \Delta'\to\Omega'\otimes !p}_{\mathsf{agg};\Sigma} \cdot; [\Gamma''; \Delta; !p; \Omega], \mathcal{P}; \Gamma; \Delta; p, \Omega \quad \text{(agg next !p } \mathcal{P}\text{)}$$
$$\mathsf{m}^\Gamma\ \Delta'\to\Omega'$$

$$\overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} \cdot; [\cdot; \Delta; !p; \Omega], \mathcal{P}; \Gamma \quad \mapsto \quad \overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} \cdot; \mathcal{P}; \Gamma \qquad \text{(agg next !frame } \mathcal{P}\text{)}$$
$$\mathsf{m}^{\Gamma}\ \Delta' \to \Omega'$$

$$\overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta_{\mathcal{I}};\Xi}\triangleleft_{\mathsf{agg};\Sigma} \cdot; \cdot; \Gamma \quad \mapsto \quad \curvearrowright_{\Xi}^{\Gamma_{N1};\Delta_{N1}} \Gamma; \Delta_{\mathcal{I}}; (\lambda x.C\{\Psi(\widehat{v})/\widehat{v}\}x)\Sigma, \Omega_N \qquad \text{(agg end)}$$
$$\Pi(\mathsf{agg}) = \forall_{\widehat{v},\Sigma'}.(\mathcal{R}_{agg}^{(\widehat{v},\Sigma')} \multimap ((\lambda x.Cx)\Sigma' \& (\forall_{\widehat{x},\sigma}.(A \multimap B \otimes \mathcal{R}_{agg}^{(\widehat{v},\sigma::\Sigma')}))))$$

C.5.4 Derivation

$$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \mathcal{C}; \mathcal{P}; \Gamma; p, \Omega \quad \mapsto \quad \Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}},p} \mathcal{C}; \mathcal{P}; \Gamma; \Omega \qquad \text{(agg new p)}$$

$$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \mathcal{C}; \mathcal{P}; \Gamma; !p, \Omega \quad \mapsto \quad \Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}},p;\Delta_{\mathcal{N}}} \mathcal{C}; \mathcal{P}; \Gamma; \Omega \qquad \text{(agg new !p)}$$

$$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \mathcal{C}; \mathcal{P}; \Gamma; X \otimes Y, \Omega \quad \mapsto \quad \Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}},p;\Delta_{\mathcal{N}}} \mathcal{C}; \mathcal{P}; \Gamma; X, Y, \Omega \qquad \text{(agg new } \otimes\text{)}$$

$$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \mathcal{C}; \mathcal{P}; \Gamma; \mathbf{1}, \Omega \quad \mapsto \quad \Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}},p;\Delta_{\mathcal{N}}} \mathcal{C}; \mathcal{P}; \Gamma; \Omega \qquad \text{(agg new } \mathbf{1}\text{)}$$

$$\Omega_{\mathcal{I}};\Delta;\Xi \curvearrowright_{\mathsf{agg};\Sigma}^{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}} \mathcal{C}; \mathcal{P}; \Gamma; \cdot \quad \mapsto \quad \overset{\Gamma_{\mathcal{N}};\Delta_{\mathcal{N}}}{\Omega_{\mathcal{I}};\Delta;\Xi}\triangleleft_{\mathsf{agg};\Sigma} \mathcal{C}; \mathcal{P}; \Gamma \qquad \text{(agg next)}$$

Appendix D

LM Directives

In this section, we list all the extra-logical directives available in LM programs.

The following directives are related to coordination:

- `priority @order ORDER` (`ORDER` can be either `asc` or `desc`): Defines if priorities are to be selected by the smallest or the greatest value, respectively.
- `priority @default P`: Informs the runtime system that all nodes must start with default priority `P`. Alternatively, the programmer can define a `set-default-priority(A, P)` initial fact.
- `priority @base P`: Informs the runtime system that the *base priority* should be `P`. The base priority is, by default, 0 when the priority ordering is `desc` and $+\infty$ when the ordering is `asc` and represents the smallest priority value possible.
- `priority @initial P`: Informs the runtime system that all nodes must start with temporary priority `P`. Alternatively, the programmer can define a `set-priority(A, P)` fact. By default, the initial priority value is equal to $+\infty$ (for `desc`) or 0 (for `asc`).

For predicate indexing, we have the following directive:

- `index pred/arg`: Informs the compiler that predicate `pred` should be indexed by the argument `arg` (number). The runtime system will use hash tree data structures for facts of this predicate indexed by the requested argument.

Appendix E

Intermediate Representation

In this appendix, we present the instructions implemented by our virtual machine. We consider that the machine has an arbitrary number of registers (or variables) that can be assigned to values or facts and a database of facts indexed by program predicates.

Values Values can be assigned to registers or to fact arguments which are stored in the database. The following values are allowed:

- Boolean values;
- Integer numbers;
- Floating point numbers;
- Node pointers;
- Thread pointers;
- String pointers;
- List pointers;
- Array pointers;

E.1 Iteration Instructions

The iteration instructions perform searches on the database of facts. When inferring a rule, each iteration instruction iterates over a predicate in the body of the rule in order to derive all combinations of facts that match the rule.

Match Specification Every instruction points to a `match` specification that filters facts from the database. The specification maps argument indexes to value templates. Value templates include the values described in E and the following variable templates:

- Register: fact argument must equal to the value stored in the register;
- Fact argument: fact argument must be equal to the value stored in another fact argument;
- List template: describes the structure of a list, namely, the head and tail value template;

Iteration Each iteration instruction contains the body, a list of instructions that must be executed for each fact found in the database. The body also returns a status indicating whether rule inference was successful or not.

- `persistent iterate [pred : predicate, dest : register, m : match, body : instructions]`

  Iterate over persistent facts with the predicate `pred` that match `m`. For each iteration, we assign `dest` the next candidate fact and execute instructions `body`.

- `linear iterate [pred: predicate, dest : register, m : match, body : instructions]`

  Same as before but iterate over linear facts.

- `reuse linear iterate [pred: predicate, dest : register, m : match, body : instructions]`

  Iterates over linear facts but, if the rule has been successful, it does not remove the linear fact stored in `dest` since the same fact has been re-derived in the head of the rule.

E.1.1 Return Instructions

Return instructions terminate the current execution flow and force the machine to backtrack to a previous state. In most cases, they are the return values used by iterate instructions and force the iterate instruction to use the next candidate fact.

- `return`

  Return without return value. Used at the end of rules to complete rule inference or in rules where a higher priority rule is available.

- `return derived`

  The rule was successfully derived.

- `return next`

  The matching has failed and the iterate instruction must use the next candidate fact.

E.2 Conditional Instructions

Conditional instructions are used to conditionally execute certain instructions.

- `if [reg : register, body : instructions]`

  If the boolean value stored in `reg` is true then execute `body`.

- `if [reg : register, body1 : instructions, body2 : instructions]`

  If the boolean value stored in `reg` is true then execute `body1`, otherwise execute `body2`.

- `jump [offset : integer]`

  Used as a goto at the end of `body1` in the instruction above. Jumps a certain number of instructions.

E.3 Arithmetic and Boolean Instructions

We briefly describe the most important arithmetic and boolean instructions.

Operations Available operations are as follows:

- Addition +;
- Subtraction -;
- Multiplication *;
- Division /;
- Integer remainder %;
- Equality =;
- Inequality <>;
- Greater >;
- Greater or equal >=;
- Lesser <;
- Lesser or equal <=;
- Logical or or;
- Logical and and;

Instructions These operations are used as follows:

- `test nil [reg : register, dest : register]`
  If the list stored in `reg` is null then set `dest` as `true`.
- `not [reg : register, dest : register]`
  Perform a boolean `not` operation.
- `int op [op1 : register, op2 : register, dest : register]`
  Perform an binary arithmetic operation on two integer numbers `op1` and `op2` and store result in `dest`. Operations include: +, -, *, /, %, =, <>, <, <=, >=, and >.
- `float op [op1 : register, op2 : register, dest : register]`
  Perform an binary arithmetic operation on two floating point numbers `op1` and `op2` and store result in `dest`. Operations include: +, -, *, /, =, <>, <, <=, >=, and >.
- `node op [op1 : register, op2 : register, dest : register]`
  Perform an operation on two node addresses `op1` and `op2` and store result in `dest`. Operations include: = and <>.
- `thread op [op1 : register, op2 : register, dest : register]`
  Perform an operation on two thread addresses `op1` and `op2` and store result in `dest`. Operations include: = and <>.
- `[bool op [op1 : register, op2 : register, dest : register]`

Perform an operation on two boolean values `op1` and `op2` and store result in `dest`. Operations include: `=` and `<>`, `and` and `or`.

E.4 Data Instructions

Data instructions include instructions which move values between registers and fact arguments and instructions which call external functions written in C++ code.

- `move-value-to-reg [val : value, reg : register]`

  Moves any value presented in E to register `reg`.

- `move-arg-to-reg [fact : register, arg : integer, reg : register]`

  Moves the $\texttt{arg}^{th}$ argument of fact stored in `fact` to register `reg`.

- `move-reg-to-arg [reg : register, fact : register, arg : integer]`

  Moves the value stored in `reg` to the $\texttt{arg}^{th}$ argument of fact stored in `fact`.

- `call [id : integer, gc : boolean, dest : register, n : integer, arg1 : register, arg2 : register, ...]`

  Call external function with number `id` with `n` arguments `arg1` up to `argn` and store the return value in `dest`. If `gc` is `true` then garbage collect the return value at the end of the rule.

E.5 List Instructions

List instructions manipulate and access list addresses.

- `cons [head : register, tail : register, gc : boolean, dest : register]`

  Create a new list and store its address in `dest`. The value of `tail` must be a list address or a null list. If `gc` is `true` then the list must be garbage collected after the rule executes.

- `head [ls : register, dest : register]`

  Retrieve the head value from the list address `ls` and store it in `dest`.

- `tail [ls : register, dest : register]`

  Retrieve the tail value from the list address `ls` and store it in `dest`.

E.6 Derivation and Removal Instructions

Derivation instructions create facts and add them to the corresponding node databases. We also include instructions that remove facts from the database.

- `alloc [pred : predicate, dest : register]`

  Allocate new fact for predicate `pred` and assign it to `dest`.

- `add linear [fact : register]`

  Add linear fact stored in `fact` to the current node's database.

- `add persistent [fact : register]`

  Add persistent fact stored in `fact` to the current node's database.

- `send [fact : register, dest : register]`

  Send fact stored in `fact` to the node address stored in `dest`.

- `run action [fact : register]`

  Execute action fact stored in `fact` and then deallocate the fact. This instruction is used for coordination instructions.

- `remove [fact : register]`

  Removes linear fact stored in `fact` from the database. The fact has been retrieved from one of the iterate instructions.

Appendix F

Further Examples

F.1 Asynchronous PageRank

The asynchronous version of the PageRank algorithm avoids synchronization between iterations, thus trading precision for convergence speed. The formulation is similar to the formulation presented in Equation 3.14:

$$x_i(t+1) = G_i[x_0(t_0^i(t)), \cdots, x_n(t_n^i(t)]^T \tag{F.1}$$

The major difference is that now we multiply G_i (the row of inbound links for page i) with a transposed vector with PageRank values that are necessarily not from iteration t. The expression $t_j^i(t)$ refers to the iteration before t when the page i received the PageRank value of page j. The value $x_j(t_j^i(t))$ then refers to the most up-to-date PageRank value of j received at page i that is going to be used to compute the PageRank value of i at $t+1$.

Figure F.1 shows the LM code for this particular version. The program uses seven predicates which are described as follows: `outbound` represents an outbound page link; `numInbound` is the number of inbound page links; `pagerank` represents the current PageRank value of the node; `neighbor-pagerank` is the PageRank value of an inbound page; `new-neighbor-pagerank` represents a new PageRank value of an inbound page; `sum-ranks` is a temporary predicate used for computing new PageRanks; and `update` re-computes the PageRank value from the neighbor's PageRank values. Rules in lines 9-17 update the `neighbor-pagerank` values, while rule in lines 19-25 asynchronously updates the current PageRank value. Finally, the third rule in lines 29-32 informs the neighbor nodes about the newly computed PageRank value by deriving multiple `new-neighbor-pagerank` facts. Note that in this rule, we use the function `fabs` that computes the absolute value of a floating point number.

To build the proof of correctness, we must again prove several program invariants. In what follows, we will prove that this particular program corresponds to the computation on a nonnegative matrix of unit spectral radius, which has been proven to converge [KGS06, LM86].

```
type outbound(node, node, float).                                    // Predicate declaration
type numInbound(node, int).
type linear pagerank(node, float, int).
type linear neighbor-pagerank(node, node Neighbor, float Rank, int Iteration).
type linear new-neighbor-pagerank(node, node Neighbor, float Rank, int Iteration).
type linear sum-ranks(node, float).
type linear update(A).

new-neighbor-pagerank(A, B, New, Iteration),          // Rule 1: update neighbor value
neighbor-pagerank(A, B, Old, OldIteration),
Iteration > OldIteration
   -o neighbor-pagerank(A, B, New, Iteration).

new-neighbor-pagerank(A, B, New, Iteration),          // Rule 2: update neighbor value
neighbor-pagerank(A, B, Old, OldIteration),
Iteration <= OldIteration
   -o neighbor-pagerank(A, B, Old, OldIteration).

sum-ranks(A, Acc),                                    // Rule 3: propagate new pagerank
NewRank = damping/float(pages) + (1.0 - damping) * Acc,
pagerank(A, OldRank, Iteration)
      -o pagerank(A, NewRank, Iteration + 1),
         {B, W, Delta | !outbound(A, B, W), Delta = fabs(NewRank -
               OldRank) * W -o new-neighbor-pagerank(B, A, NewRank, Iteration + 1),
               if Delta > bound then update(B) end}.

update(A), update(A) -o update(A).                      // Rule 4: prune update facts

update(A),                                              // Rule 5: start update process
!numInbound(A, T)
   -o [sum => V; B, Val, Iter | neighbor-pagerank(A, B, Val, Iter),
         V = Val/float(T) -o neighbor-pagerank(A, B, Val, Iter) -> sum-ranks(A, V)].

pagerank(A, 1.0 / float(pages), 0).                                  // Initial facts
update(A).
neighbor-pagerank(A, B, 1.0 / float(pages), 0). // pagerank of B is ...
```

Figure F.1: *Asynchronous PageRank program.*

Invariant F.1.1 (Page invariant)
Each page/node has a single `pagerank(A, Value, Iteration)` and:

- for each outbound link, a single `!outbound(A, B, W)` fact.
- for each inbound link, a single `neighbor-pagerank(A, B, V, Iter)` fact.
- for each `!outbound(A, B, W)`, a single `neighbor-pagerank(B, A, V, Iter)`.

Proof. All initial facts validate the 3 conditions of the variant. Note that the third condition is also validated by the initial facts, although not all facts are shown in the code.

In relation to rule application:

- Rule 1: inbound link re-derived.
- Rule 2: inbound link re-derived.
- Rule 3: pagerank re-derived.
- Rule 4: Nothing happens.
- Rule 5: inbound links re-derived in the comprehension.

□

Lemma F.1.1 (Neighbor rank lemma)
Given a fact `neighbor-pagerank(A, B, V, Iter)` and a set of facts `new-neighbor-pagerank(A, B, New, Iter2)`, we end up with a single `neighbor-pagerank(A, B, V', Iter')`, where `Iter` is the greater of `Iter` and `Iter2'`.

Proof. By induction on the number of `new-neighbor-pagerank` facts.
Base case: `neighbor-pagerank` remains.
Inductive case: given one `new-neighbor-pagerank` fact:

- Rule 1: the new iteration is older and thus `neighbor-pagerank` is replaced. By applying induction, we know that we will select either the new best iteration or a better iteration from the remaining set of `new-neighbor-pagerank` facts.
- Rule 2: the new iteration is not older and we keep the old `neighbor-pagerank` fact. By induction, we select the best from either the current iteration or some other (from the set).

□

Lemma F.1.2 (Update lemma)
Given a new `update` fact, rule 5 will run.

Proof. By induction on the number of `update` facts.
Base case: rule 5 will run for the first update fact.
Inductive case: rule 4 runs first because it has a higher priority, reducing the number of `update` facts by one. By induction, we know that by using the remaining `update` facts, rule 5 will run. □

Lemma F.1.3 (Pagerank update lemma)
(1) Given at least one `update` fact, the `pagerank(A,` V_I`, I)` fact will be updated to become `pagerank(A,` V_{I+1}`, I + 1)`, where $V_{I+1} = damping/P + (1.0 - damping) \sum_{B,I} Val_{I,B} W_B$. with $W_B = 1.0/T$ (where T is the number of outbound links of `B`) and $Val_{I,B}$ from `neighbor-pagerank(A, B,` $Val_{I,B}$`, I)`.
(2) For all `B` outbound nodes (from `!outbound(A, B, W)`, a `new-neighbor-pagerank(B,`

A, V_{I+1}, $I+1$) is generated.
(3) For all B outbound nodes (from !outbound(A, B, W)), a update(B) is generated if $fabs(V_{I+1} - V_I)W > bound$.

Proof. Using the Update lemma, rule 5 will necessarily run, which will derive sum-ranks(A, $\sum_{B,I}(Val_{I,B}W_B)$) and fulfills (3).

Fact sum-ranks will necessarily activate rule 4, computing V_{I+1} and updating pagerank. (2) and (3) are fulfilled through the comprehension of rule 4.

□

Invariant F.1.2 (New neighbor rank equality)
All new-neighbor-pagerank(A, B, V, I) facts are generated from a corresponding pagerank(B, V, I) fact, therefore the iteration of any new-neighbor-pagerank is at least the same or less than the iteration of the current PageRank.

Proof. No initial facts to prove.

- Rule 3: true, new fact is generated.
- Rule 4: the fact is kept.

□

Invariant F.1.3 (Neighbor rank equality)
All neighbor-pagerank(A, B, V, I) facts have one corresponding pagerank(B, V, I) fact and the iteration of the neighbor-pagerank is the same or less than the current iteration of the corresponding pagerank.

Proof. By analyzing initial facts and rules.

Axioms: true.

Rule cases:

- Rule 1: uses new-neighbor-pagerank fact (use new neighbor rank equality invariant).
- Rule 2: same fact is re-derived.

□

Theorem F.1.1 (Pagerank convergence)
The program will compute the PageRank of all nodes that is within bound error of an asynchronous PageRank computation.

Proof. Using the program initial facts, we start with the same PageRank value for all nodes. The `!outbound(A, B, W)` fact forms the $n \times n$ square matrix (number of nodes) and is the Google Matrix. All the initial PageRank values can be seen as a vector that adds up to 1.

The PageRank computation from the "Pagerank update lemma" computes $V_{I+1} = damping/P + (1.0 - damping) \sum_{B,I'}(W_B Val_{I',B})$, where $I' \leq I$ (from Neighbor rank equality invariant).

Consider that each node contains a column G_i of the Google matrix. The PageRank computation can then be represented as:

$$V_{I+1} = G_i fix([Val_{I_1,B_1}, ..., Val_{I_p,B_p}]) \tag{1}$$

Where p is the number of inbound links and Val_{I_j,B_j} is the value of the `neighbor-pagerank(A,` B_j, Val_{I_j,B_j}, I_j`)`. The $fix()$ function takes the neighbor vector and expands it with zeros corresponding to nodes that are not inbound links. This is the expected formulation for the asynchronous PageRank computation [KGS06] as shown in F.1.

From [KGS06, LM86] we know that equation (1) will improve (converge) the PageRank value, given that some new neighbor PageRank values are sent to node i and by the fact that G_i is a nonnegative matrix of unit spectral radius. Let's use induction by assuming that there is at least one `update` fact that schedules a node to improve its PageRank. We want to prove that such fact will not only improve the node's PageRank but also the PageRank vector. If the PageRank vector is now close enough (within `bound`), then the program will terminate.

- Base case: since we have an `update` fact as an axiom, rule 5 will compute a new PageRank (Pagerank update lemma) for all nodes that is improved (from equation (1)). From these updates, a new `update` fact is generated that correspond to nodes that have inbound links from the source node and need to update their PageRank. These `update` facts may not be generated if the PageRank vector is close enough to its real value.
- The induction hypothesis tells us that there is at least one node that has an `update` fact. From PagePank update lemma, this generates `new-neighbor-pagerank` facts if the new value differs significantly from the older value. When this happens and using the "Neighbor rank lemma", the target node will update its `neighbor-pagerank` fact with the newest iteration and then execute a valid PageRank computation that brings the PageRank vector close to its solution.

□

Appendix G

Program Proofs

G.1 Single Source Shortest Path

The most interesting property of the SSSP program presented in Fig. 7.8 is that it remains provably correct, although it applies rules using a different ordering. We now show the complete proof of correctness.

Invariant G.1.1 (Distance)
`relax`(A, D, P) represents a valid distance D and a valid path P from node `@1` to node A. If the shortest distance to `@1` is D', then $D \geq D'$.
`shortest`(A, D, P) represents a valid distance D and a valid path P from node `@1` to node A. If the shortest distance to `@1` is D', then $D \geq D'$. The `shortest` fact may also represent an invalid distance if $D = \infty$, where $P = []$.

Proof. By mutual induction. All the initial facts are valid and the first (lines 7-11) and second rules (lines 13-14) of the program validate the invariant using the inductive hypothesis. □

Lemma G.1.1 (Relaxation)
Shorter distances are propagated to the neighbor nodes exactly once.

Proof. By the first rule, we know that for a new shorter distance, we keep the shorter distance and propagate it. By the second rule, longer distances are ignored. □

Theorem G.1.1 (Correctness)
Assume a graph $G = (V, E)$ where $w(a, b) \geq 0$ and $(a, b) \in E$ (positive weights). Consider that there is a set of nodes $S \in V$ where the shortest distance has been computed and a set

$U \in V$ where the shortest distance has not been computed yet. Sets are S and U are disjoint. At any given point, Σ is the sum of all current distances. For the distance ∞ we assign the value $\Sigma' + 1$, where Σ' is the largest shortest distance of any node reachable from `@1`. Every rule inference will either:

- Maintain the size of S and reduce the total number of facts in the database.
- Increase the size of S, reduce Σ and potentially increase the number of facts in the database.
- Maintain the size of S, reduce Σ and potentially increase the number of facts in the database.

Eventually, set $S = V$ and every `shortest`(A, D, P) will represent the shortest distance from A to `@1` and P is its corresponding path.

Proof. By nested induction on Σ and on the number of facts in the database.

In the base case, we have `relax(@1, 0, [@1])` that will give us the shortest distance for node `@1`, therefore $S = \{@1\}$ and Σ is reduced.

In the inductive case, we have a set S' where the shortest distance was reached and `relax` distances may have been propagated (Relaxation Lemma).

Now consider the two rules of the program:

- The first rule will only apply at nodes in U. If the shortest `relax` is selected, then the node is added to S, otherwise it stays in U but improves the shortest path, reduces Σ and `relax` facts are generated (Relaxation Lemma).
- The second rule is applied in either nodes of S or U. For both sets, the rule retracts the `relax` fact.

The case where the first rule derives the shortest `relax` distance to node $t \in U$ happens when some node $s \in S$ which minimizes $\text{argmin}_t\, d(s) + w(s, t)$ is selected, where $d(s)$ is the shortest distance to node `@1` and $w(s, t)$ the weight of the edge between (s, t). Using `set-priority` increases the probability of that node being selected, but it does not matter since the program always makes progress and the shortest distances will be eventually computed.

□

G.2 MiniMax

To prove that the MiniMax code shown in Fig. 7.12 computes the best score along with the best possible move that the `root-player` can make, we have to generalize the proof and inductively prove that each node selects the best score depending if its a minimizing or a maximizing node. We start with several useful lemmas.

Lemma G.2.1 (Play Lemma)
If a fact `expand`$(A, FirstGame, RestGame, ND, NextPlayer, Play, Depth)$ exists then

$\exists_n$ where n is the number of available plays that $NextPlayer$ can make on the remaining game state $RestGame$. The initial `expand` fact is retracted and generates n children B with facts $\texttt{play}(B, Game', OPlayer, Play', Depth + 1)$ and $\texttt{parent}(B, A)$, where $OPlayer = \texttt{next} - \texttt{player}(NextPlayer)$, $Play'$ is the index of an empty position in $RestGame$ plus the length of $FirstGame$, and $Game'$ represents the concatenation of $FirstGame$ and $RestGame$ where an empty position has been modified. The initial `expand` fact eventually derives either the $\texttt{maximize}(A, ND + Empty, -\infty, 0)$ fact (if $NextPlayer = root - player$) or the $\texttt{minimize}(A, ND + Empty, \infty, 0)$ (otherwise). The value $Empty$ indicates the number of empty positions in the state $RestGame$.

Proof. By induction on the size of the list $RestGame$. There are 5 possible rules.

Rule 1 (lines 22-23: $RestGame = []$ thus a `maximize` fact is derived as expected.

Rule 2 (lines 25-26: $RestGame = []$ thus a `minimize` fact is derived as expected.

Rule 3 (lines 28-32: in this case we have a non-null $RestGame$ and an empty position on index 0. As expected, we create a new B node with the expected facts and a new `expand` fact with a smaller $RestGame$. Apply the induction hypothesis to get the remaining $n - 1$ potential plays for $NextPlayer$.

Rule 4 (lines 34-37: same reasoning as rule 3. Note that the presence of coordination facts do not change the proof because they are not used in the rule's LHS.

Rule 5 (lines 39-40: no free space on the first position of $RestGame$. We derive another `expand` fact with a reduced $RestGame$ and use the induction hypothesis.

□

Lemma G.2.2 (Children Lemma)
The fact $\texttt{play}(A, Game, NextPlayer, LastPlay, Depth)$ is consumed by the program's rule to realize one of the following scenarios:

- A new fact $\texttt{score}(A, Score, LastPlay)$ is derived.
- There exists a number n of available plays for $NextPlayer$ in $Game$, which generate n new children B where each B will have facts $\texttt{play}(B, Game', OPlayer, Play, Depth + 1)$ and $\texttt{parent}(B, A)$. Furthermore, from the use of the initial $play$ fact, one of the following facts is derived:
  - $\texttt{maximize}(A, N, -\infty, 0)$ is derived (if $NextPlayer = root - player$);
  - $\texttt{minimize}(A, N, \infty, 0)$ (otherwise).

  Note that $Game'$ is an updated $Game$ state where an empty position is played by $NextPlayer$.

Proof. A linear fact $\texttt{play}(A, Game, NextPlayer, LastPlay, Depth)$ can only be used in either the first or second rules. If the rule in lines 14-16 executes, (`minimax_score` returns a score) it means that $Game$ is a final game state and there's no available plays for $NextPlayer$. Otherwise, the second rule in lines 18-20 will apply and the Play lemma is used to complete the proof.

□

We now prove that the minimization and maximization rules work given the right amount of scores available.

Lemma G.2.3 (Maximize Lemma)
Given a fact `maximize`$(A, N, BScore, BPlay)$ and N facts `new-score`$(A, OScore, OPlay)$, the program's rule will retract them all and generate a single `score`$(A, BScore', BPlay')$ fact where $BScore'$ is the highest score from $BScore$ or $OScore'$ and $BPlay'$ is the corresponding play.

Proof. By induction on the number of `new-score` facts.
Case 1 ($N = 0$): trivial by applying the rule in line 58.
Case 2 ($N > 0$): by picking an arbitrary `new-score`$(A, OScore, OPlay)$ fact.

- Sub case 2.1 (lines 52-53)
  If $BScore < OScore$ then we derive `maximize`$(A, N - 1, OScore, OPlay)$. Use induction to get the final `score` fact.
- Sub case 2.2 (lines 55-56)
  If $BScore \geq OScore$ then we derive `maximize`$(A, N - 1, BScore, BPlay)$. Use induction to get the final `score` fact.

□

Lemma G.2.4 (Minimize Lemma)
Given a fact `minimize`$(A, N, BScore, BPlay)$ and N `new-score`$(A, OScore, OPlay)$ facts, the program's rules will retract all those facts and derive a single `score`$(A, BScore', BPlay')$ fact where $BScore'$ is the lowest score from either $BScore$ or the N $OScore$ and $BPlay'$ is the corresponding play.

Finally, we are in a position to prove that a `play` fact is retracted and then eventually produces a `score` fact that indicates the best score and best play for the player playing at the given node.

Theorem G.2.1 (Score Theorem)
For every `play`$(A, Game, NextPlayer, LastPlay, Depth)$ fact, we either get a `score` fact (leaf case) or a recursive alternate maximization or minimization (depending if `NextPlayer = root-player` or otherwise) of the children nodes. This max/minimization also results in a `score`$(A, Score, BPlay)$ fact where $Score$ is the max/minimum and $BPlay$ is the corresponding play for that score.

Proof. By applying the Children Lemma and using induction on the number of free positions on the state $Game$.

Case 1 (no free positions or game is final): direct score.

Case 2 (available positions): n children nodes B are created with two facts: `parent`(B, A) and `play`$(B, Game', $`next-player`$(NextPlayer), X, Play, Depth{+}1)$, where $Game'$ has position X filled up. We apply induction on the `play` fact of each child B to get a `score`$(B, Score, Play)$ fact. Since we also derived a `parent`(B, A) fact, rule in line 42 eventually executes, deriving `new` $-$ `score`$(A, Score, Play)$.

We also derived `maximize`$(A, N, -\infty, 0)$ fact (or `minimize`) and n `new-score` facts from the n children nodes, from which we apply the Maximize Lemma to get `score`$(A, BScore, BPlay)$. □

Corollary G.2.1 (MiniMax)
Starting from a `play`$(@0, initial{-}game, root{-}player, 0, 1)$ fact, the fact `score`$(A, BScore, BPlay)$ is eventually derived, where $BPlay$ represents the best play which player $root - player$ is able to make that minimizes the possible loss for a worst case scenario given $initial - game$.

G.3 N-Queens

We prove that the N-Queens program finds all the distinct solutions for the puzzle. In the following lemmas, we assume that (X_0, Y_0) is the coordinate of square/node A.

Lemma G.3.1 (test-y lemma)
From a fact `test-y`$(A, Y, State, OrigState)$, there are two possible scenarios:

- The `test-y` fact and everything derived from it is retracted and there is a $Y \in State$.
- The `test-y` fact is retracted and is used to derive a `test-diag-left`$(A, X_0 - 1, Y_0 - 1, OrigState, OrigState)$ fact since there is no such $Y \in State$.

Proof. Induction on the size of `State`.

First rule: immediately the second scenario.

Second rule: immediately the first scenario.

Third rule: by induction. □

Lemma G.3.2 (test-diag-left lemma)
From a fact `test-diag-left`$(A, X, Y, State, OrigState)$, there are two possible scenarios:

- The `test-diag-left` fact and everything derived from it is retracted and there is a $y' \in State$, where $y' = Y - a$ and a is non-negative number representing the index of y' in $State$.
- The `test-diag-left` is retracted and is used to derive a `test-diag-right`$(A, X_0 -$

$1, Y_0 + 1, OrigState, OrigState)$ fact since there is no $y' \in State$ as specified above.

Proof. Induction on the size of `State`.
First rule: immediately the second scenario.
Second rule: immediately the first scenario.
Third rule: by induction. □

Lemma G.3.3 (test-diag-right lemma)
From a fact `test-diag-right`$(A, X, Y, State, OrigState)$, there are two possible scenarios:

- The `test-diag-right` fact and everything derived from it is retracted and there is a $y' \in State$, where $y' = Y + a$ and a is a non-negative number representing the index of y' in $State$.
- The `test-diag-right` fact is retracted and the fact `send-down`$(A, [Y_0|OrigState])$ is derived.

Proof. Induction on the size of `State`.
First rule: immediately the second scenario.
Second rule: immediately the first scenario.
Third rule: by induction. □

Theorem G.3.1 (State validation)
From a fact `test-y`$(A, Y_0, State, State)$, where (X_0, Y_0) is the coordinate of A and the length of $State$ is X_0, there are two possible scenarios:

- The `test-y` fact and anything derived from it is retracted.
- The `test-y` fact is retracted and a `send-down`$(A, [Y_0|State])$ fact is derived, where $[Y_0|State]$ is a valid board state.

Proof. Use the previous three lemmas. □

Lemma G.3.4 (Propagate left lemma)
If a `propagate-left`$(A, State)$ fact exists then every cell to the left, including `A` will derive `new-state(A, State)`. The original `propagate-left` fact and any other fact derived from it (except `new-state`) are retracted.

Proof. By induction on the number of cells to the left of A and using the only rule that uses `propagate-left`.

□

Lemma G.3.5 (Propagate right lemma)
If a `propagate-right`$(A, State)$ fact exists then every cell to the left, including `A` will derive `new-state(A, State)`. The original `propagate-right` fact and any other fact derived from it (except `new-state`) are retracted.

Proof. By induction on the number of cells to the left of A and using the only rule that uses `propagate-right`. □

Theorem G.3.2 (States theorem)
For a given row, we compute several `send-down`$(A, State)$ facts that represent valid states that include that row and the rows above.

Proof. By induction on the number of rows.

For row 0, we use the initial fact `propagate-right`$(@0, [])$, that will be propagated to all nodes in row 0 (propagate right lemma). By using the state validation theorem, we know that every node will derive `send-down(A, [Y])`, which are all the valid states.

By induction, we know that row X' has derived every `send-down` fact possible. Such facts will be sent downwards to row $X = X' + 1$ using the last rule in the program, deriving `propagate-right` or `propagate-left` that, in turn, will derive a `new-state` fact at each right or left square. Nothing is derived at the square below or the diagonal cells since they are not valid. From the `new-state` fact, a `test-y` fact is derived, which will be checked using the state validation theorem, filtering all new valid states and then finally deriving a new `send-down` fact.

□

G.4 Binary Search Tree

In this section, we prove several properties of the Binary Search Tree (BST) as presented in Fig. 8.7. First, we assume that the initial facts of the program represent a valid binary search tree, therefore, for every node with a given key k, the left branch contains keys that are lesser or equal than k, and the right branch contains keys that are greater than k.

Invariant G.4.1 (Immutable keys)
Every node has a `value`$(A, Key, Value)$ fact and the Key never changes.

Proof. By induction. In the base case, every node has a `value` initial fact. In the inductive case, each rule that uses a `value` fact also re-derives it and the Key argument never changes. □

Invariant G.4.2 (Thread state)
Every thread T in the program has a `cache-size`$(T, Total)$ fact that represents the number of valid cache items in the form of `cache`$(T, Node, Key)$ facts. Furthermore, there are no two `cache`$(T, Node_1, Key_1)$ and `cache`$(T, Node_2, Key_2)$ facts where $Node_1 = Node_2$.

Proof. By induction on each rule application.

In the base case, all threads start with a `cache-size`$(T, 0)$ fact and no `cache`$(T, Node, Key)$ facts.

In the inductive case, we must analyze each rule separately.

- Rule 1: the `cache`(T, A, Key) fact is kept in the database.
- Rule 2: this rule adds a new valid `cache`(T, A, Key) fact and correctly increments the `cache-size` counter. However, we must ensure that the cache item is unique. For that, we know that this rule uses the same `replace` and a `value` facts of rule 1, therefore, if there was already a cache item in the database, rule 1 would run before rule 2 since rule 1 has a higher priority. In a nutshell, rule 1 deals with cases where there is already a `cache` item, while rule 2 adds a new item if it does not exist.
- Rule 3: the `cache`$(T, TargetNode, RKey)$ fact is kept in the database.
- Rules 4 and 5: they do not derive or consume any cache related facts.

□

Theorem G.4.1 (Valid replace)
Assume a valid BST and a set of threads with a valid cache. Given a `replace`$(A, RKey, RValue)$ fact at node A, the node N with key $RKey$ will replace its `value`$(N, RKey, Value)$ fact to `value`$(N, RKey, RValue)$.

Proof. First consider that node A makes up a BST and then use induction on the size of that smaller BST.

In the base case, where $A = N$, rule 1 and 2 will apply. Rule 1 is applied if the executing thread has node N already in the cache. Rule 2 applies when the executing thread does not have the node N in the cache.

In the inductive case, there are two sub-cases:

- The executing thread has a valid `cache` item for key $RKey$ (from Variant G.4.2), therefore rule 3 is applied and a `replace` fact is derived at node N.

- Without a valid `cache` item, rules 4 or 5 are applied. Rule 4 applies if the key is in the left branch and rule 5 applies if the key is in the right branch. Use the induction hypothesis on the selected branch.

□

Bibliography

[ACC+10] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *European Conference on Computer Systems (EuroSys)*, pages 223–236, 2010. 2.2.2

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986. 2.3, 7.9

[ACR11] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 499–518, New York, NY, USA, 2011. 2.2.4

[ACR13] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–228, 2013. 2.2.4

[AK90] KhayriA.M. Ali and Roland Karlsson. Full Prolog and scheduling or-parallelism in muse. *International Journal of Parallel Programming*, 19(6):445–475, 1990. 6.5.1

[Ali88] Khayri A. M. Ali. OR-parallel execution of Prolog on BC-machine. In Robert A. Kowalski and Kenneth A. Bowen, editors, *ICLP/SLP*, pages 1531–1545. MIT Press, 1988. 6.5.1

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992. 4.2

[AP95] Peter Achten and Rinus Plasmeijer. The ins and outs of clean I/O. *Journal of Functional Programming*, 5:81–110, January 1995. 4.1

[ARLG+09] Michael P. Ashley-Rollman, Peter Lee, Seth C. Goldstein, Padmanabhan Pillai, and Jason D. Campbell. A language for large ensembles of independently executing nodes. In *International Conference on Logic Programming*, pages 265–280, 2009. 1, 2.2.2, 2.3, 2.4, 3.6.2, 1

[ARRS+07] Michael P. Ashley-Rollman, Michael De Rosa, Siddhartha S. Srinivasa, Padmanabhan Pillai, Seth C. Goldstein, and Jason D. Campbell. Declarative programming for modular robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS 2007*, 2007. 1, 2.2.2

[Bac06] Lars Backstrom. Group formation in large social networks: membership, growth,

and evolution. In *In KDD 06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM Press, 2006. 6.4.1

[Bae12] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1):1–44, 2012. 4.1.2

[Bag01] Phil Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001. 5.2

[BB06] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, March 2006. 3.6.2

[BDL+88] Ralph Butler, Terry Disz, Ewing L. Lusk, Robert Olson, Ross A. Overbeek, and Rick L. Stevens. Scheduling OR-parallelism: An Argonne perspective. In Robert A. Kowalski and Kenneth A. Bowen, editors, *ICLP/SLP*, pages 1590–1605. MIT Press, 1988. 6.5.1

[BF05] Hariolf Betz and Thom Frühwirth. A linear-logic semantics for constraint handling rules. In Peter Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151. Springer Berlin Heidelberg, 2005. 3.6.3

[BF13] Hariolf Betz and Thom Frühwirth. Linear-logic based analysis of constraint handling rules with disjunction. *ACM Trans. Comput. Logic*, 14(1):1:1–1:37, February 2013. 3.6.3

[BH77] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM. 2.2.1

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, 1995. 2.1

[Bla92] Andreas Blass. A game semantics for linear logic. *Annals of Pure and Applied logic*, 56(1):183–220, 1992. 4.1

[Ble96] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, 1996. 1, 2.2.1

[BM07] David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In *LPAR*, volume 4790, pages 92–106, 2007. 4.1.2

[Bon94] J. Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Usenix Summer 1994 Technical Conference*, pages 87–98. Usenix Association, 1994. 6.3

[Bos11] Johan Bos. A survey of computational semantics: Representation, inference and knowledge in wide-coverage text understanding. *Language and Linguistics Compass*, 5(6):336–366, 2011. 4.1

[BR87] I. Balbin and K. Ramamohanarao. A generalization of the differential approach

to recursive query evaluation. *J. Log. Program.*, 4(3):259–262, September 1987. 6.5.2

[BRF10] Hariolf Betz, Frank Raiser, and Thom W. Frühwirth. A complete and terminating execution model for constraint handling rules. *CoRR*, abs/1007.3829, 2010. 3.6.3

[But97] David R. Butenhof. *Programming with POSIX Threads*. 1997. 1

[CCH07] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. Towards high-level execution primitives for and-parallelism: Preliminary results, 2007. 8.5

[CCP03] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, 2003. 2.2.2

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005. 2.1

[CGSvE93] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993. 6.5.1

[CJvdP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. 2007. 1

[CLJ+07] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon P. Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, pages 10–18, 2007. 2.2.1

[CPSV07] Vittoria Colizza, Romualdo Pastor-Satorras, and Alessandro Vespignani. Reaction-diffusion processes and metapopulation models in heterogeneous networks, 2007. 6.4.1

[CPT+07] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 175–188, 2007. 2.2.2

[CPWW02] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, 2002. 4.5

[CR93] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 37–52, New York, NY, USA, 1993. 2.2.2

[CRGP14] F. Cruz, R. Rocha, S. Goldstein, and F. Pfenning. A Linear Logic Programming Language for Concurrent Programming over Graph Structures. *Journal of Theory and Practice of Logic Programming, 30th International Conference on Logic Programming, Special Issue*, pages 493–507, July 2014. 2.2.2

[DARR+08] Daniel J. Dewey, Michael P. Ashley-Rollman, Michael De Rosa, Seth C. Goldstein, Todd C. Mowry, Siddhartha S. Srinivasa, Padmanabhan Pillai, and Jason D. Campbell. Generalizing metamodules to simplify planning in modular robotic systems. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1338–1345, September 2008. 2.4

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008. 2.2.3

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. 7.1, 7.7

[Dij02] Edsger W. Dijkstra. Cooperating sequential processes. In *The Origin of Concurrent Programming*, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002. 2.1

[DKSD07] Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '07, pages 25–36, New York, NY, USA, 2007. ACM. 3.6.3

[DKSD08] Leslie De Koninck, PeterJ. Stuckey, and GregoryJ. Duck. Optimizing compilation of CHR with rule priorities. In *Functional and Logic Programming*, volume 4989 of *LNCS*, pages 32–47. 2008. 6.5.3

[EP04] Hartmut Ehrig and Julia Padberg. Graph grammars and Petri net transformations. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 496–536. Springer Berlin Heidelberg, 2004. 3.6.4

[For94] Message P Forum. MPI: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994. 2.1

[FRRS10] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in manticore. *Journal of Functional Programming*, 20(5-6):537–576, November 2010. 2.2.1

[GCM05] Seth C. Goldstein, Jason D. Campbell, and Todd C. Mowry. Programmable matter. *IEEE Computer*, 38(6):99–101, 2005. 1, 2.2.2

[GDF+01] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Böhm, Walid Najjar, and Patrick Miller. The sisal project: Real world functional programming. In *Compiler optimizations for scalable parallel systems*, pages 45–72. Springer, 2001. 2.2.1

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. I. *Mathematische Zeitschrift*, 39(1):176–210, 1935. 4.1.1

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European PVM/MPI Users' Group Meeting*, pages 97–104,

2004. 1

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. 2.2.2, 3

[Gir95] Jean-Yves Girard. Linear logic: Its syntax and semantics. In *Advances in Linear Logic*, pages 1–42, 1995. 4.1

[GLG09] Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics (AISTATS)*, 2009. 8.3.3

[Gol94] Seth Copen Goldstein. The implementation of a threaded abstract machine. Technical Report UCB/CSD-94-818, EECS Department, University of California, Berkeley, 1994. 6.5.1

[GPA+01] Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of Prolog programs: A survey. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(4):472–602, 2001. 2.2.2

[GST90] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. A framework for the parallel processing of Datalog queries. In *ACM SIGMOD International Conference on Management of Data*, pages 143–152, New York, NY, USA, 1990. ACM. 2.2.2

[HdlBSD04] Christian Holzbaur, Maria J. García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing compilation of constraint handling rules in HAL. *CoRR*, cs.PL/0408025, 2004. 6.5.3

[Her86] Manuel Victor Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, 1986. AAI8700203. 6.5.1

[HJ90] Seif Haridi and Sverker Janson. Kernel andorra Prolog and its computation model. In *Logic Programming*, pages 31–46. MIT Press, Cambridge, MA, USA, 1990. 6.5.1

[HLM69] E. J. Hoffman, J. C. Loessi, and R. C. Moore. Construction for the solutions of the M queens problem. *Mathematics Magazine*, 42(2):66–72, 1969. 7.8.2

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110:32–42, 1994. 4.5

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM, 2005. 2.2.1

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. 2.4

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. 2.1

[How80] William A. Howard. The formulae-as-types notion of construction. In *ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 480–490, Boston, MA, 1980. 4.1

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007. 2.2.3, 3.6.1

[JLKC08] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 383–414, Dagstuhl, Germany, 2008. 2.2.1

[KDGN97] M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors. *Abstract Machine Models for Parallel and Distributed Computing*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1997. 6.5.1

[KGMG07] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 200–210, New York, NY, USA, 2007. 3.6.2

[KGS06] Giorgios Kollias, Efstratios Gallopoulos, and Daniel B. Szyld. Asynchronous iterative computations with web information retrieval structures: The pagerank case. *CoRR*, abs/cs/0606047, 2006. F.1, F.1

[lan] The computer language benchmarks game. http://benchmarksgame.alioth.debian.org/. Accessed: 2015-05-25. 6.4.1

[LC13] Edmund S. L. Lam and Iliano Cervesato. Decentralized execution of constraint handling rules for ensembles. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 205–216, New York, NY, USA, 2013. ACM. 9.3

[LCG+06] Boon T. Loo, Tyson Condie, Minos Garofalakis, David E. Gay, and Joseph M. Hellerstein. Declarative networking: Language, execution and optimization. In *International Conference on Management of Data*, pages 97–108, 2006. 2.2.2, 2.2.2, 5.3, 6.5.2

[LGK+10] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010. 1.1, 3.6.1, 8.3.3

[LHL95] Bertram Ludäscher, Ulrich Hamann, and Georg Lausen. A logical framework for active rules. In *International Conference on Management of Data*, 1995. 4.1

[Liu98] Mengchi Liu. Extending Datalog with declarative updates. In *International Conference on Database and Expert System Applications (DEXA)*, volume 1873, pages 752–763, 1998. 4.1

[LK88] Yow Lin and Vipin Kumar. And-parallel execution of logic programs on a shared memory multiprocessor:a summary of results. Technical report, Austin, TX, USA, 1988. 6.5.1

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014. 6.4.1, 6.4.4, 6.4.4

[LKF07] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007. 6.4.1

[LM86] Boris Lubachevsky and Debasis Mitra. A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. *Journal of the ACM*, 33:130–150, 1986. F.1, F.1

[LM12] Jure Leskovec and Julian J. Mcauley. Learning to discover social circles in ego networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 539–547. Curran Associates, Inc., 2012. 6.4.1

[Loo06] Boon Thau Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2006. 2.2.2

[LPPW05a] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 35–46, New York, NY, USA, 2005. ACM. 2.2.2, 4.5

[LPPW05b] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 35–46. ACM, 2005. 4.1

[LRS+03] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á J. Rebón, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16(3):203–251, September 2003. 2.2.1

[LS90] S. Lucco and O. Sharp. Delirium: an embedding coordination language. In *Supercomputing '90., Proceedings of*, pages 515–524, Nov 1990. 2.3, 7.9

[LS91] Yves Lafont and Thomas Streicher. Games semantics for linear logic. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 43–50. IEEE, 1991. 4.1

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *International Conference on Management of Data*, pages 135–146, 2010. 3.6.1

[Mar15] Chris Martens. *Programming Interactive Worlds with Linear Logic*. PhD thesis, Milwaukee, USA, July 2015. 9.2

[MBFC13] Chris Martens, Anne-Gwenn Bosser, Joao F Ferreira, and Marc Cavazza. Linear logic programming for narrative generation. In *Logic Programming and Nonmonotonic Reasoning*, pages 427–432. Springer, 2013. 4.1

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991. 6.1

[MFBC14] Chris Martens, Joao F Ferreira, Anne-Gwenn Bosser, and Marc Cavazza. Generative story worlds as linear logic programs. In *Intelligent Narrative Technologies 7 (INT7)*, Milwaukee, USA, July 2014. 4.1

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, March 2005. 3.6.2

[Mil85] Dale Miller. An overview of linear logic programming. In *in Computational Logic*, pages 1–5. Cambridge University Press, 1985. 4.1, 4.5

[Mil99] Robin Milner. *Communicating and mobile systems: the pi calculus*. 1999. 2.4

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association. 6.4.1

[Mis89] Jayadev Misra. A foundation of parallel programming. In *Constructive Methods in Computing Science*, volume 55, pages 397–445. 1989. 2.4

[MML+10] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: Better strategies for Parallel Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 91–102, New York, NY, USA, 2010. ACM. 2.2.1

[MWJ99] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 467–475, 1999. 8.3.3

[MZ10] Karl Mazurak and Steve Zdancewic. Lolliproc: to concurrency from classical linear logic via Curry-Howard and control. In *ACM Sigplan Notices*, volume 45, pages 39–50. ACM, 2010. 4.1

[Nik93] Rishiyur S. Nikhil. An overview of the parallel language Id (a foundation for pH, a parallel dialect of Haskell). Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1993. 2.2.1

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, pages 489–498, New York, NY, USA, 2007. 3.6.2

[NP11] Donald Nguyen and Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 333–344, 2011. 2.3, 7.9, 8.5

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. 6.4.1

[OG76] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976. 2.4

[OP09] Tore Opsahl and Pietro Panzarasa. Clustering in weighted networks. *Social Networks*, 31(2):155 – 163, 2009. 6.4.1

[Ops15] Tore Opsahl. TNET datasets. http://toreopsahl.com/datasets, June 2015. 6.4.1

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, New York, NY, USA, 2008. ACM. 2.2.3

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *Advances in Computers*, pages 329–400, 1998. 2.3

[Pag01] Lawrence Page. Method for node ranking in a linked database. US Patent 6,285,999, 2001. Filed January 9, 1998. Expires around January 9, 2018. 3.5.2

[PCPT12] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 539–558. Springer Berlin Heidelberg, 2012. 4.1

[PMP12] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A system for synthesizing concurrent graph programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 375–394, New York, NY, USA, 2012. ACM. 2.3, 7.9, 8.5

[PNK+11] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011. 2.3, 3.6.1, 7.9

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In *Automated Deduction, CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. 1999. 2.4

[RF11] Frank Raiser and Thom Frühwirth. Analysing graph transformation systems through constraint handling rules. *Theory Pract. Log. Program.*, 11(1):65–109, January 2011. 3.6.4

[RGL+08] Michael De Rosa, Seth C. Goldstein, Peter Lee, Padmanabhan Pillai, and Jason D. Campbell. Programming modular robots with locally distributed predicates. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3156–3162, May 2008. 3.6.2

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo

Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. 7.9, 8.5

[RM97] Gruia-Catalin Roman and Peter J. McCann. An introduction to mobile UNITY. In *Parallel and Distributed Processing*, pages 871–880. 1997. 2.4

[RU93] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993. 2.2.2

[SB13] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013. 1.1, 3.6.1, 6.4.1, 7.7

[SGG08] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008. 2.1

[SJ09] Satnam Singh and Simon Peyton Jones. *A Tutorial on Parallel and Concurrent Programming in Haskell*. Lecture Notes in Computer Science. Springer Verlag, Nijmegen, Holland, May 2009. 2.2.1

[SL91] Jürgen Seib and Georg Lausen. Parallelizing Datalog programs by generalized pivoting. In *Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 241–251, New York, NY, USA, 1991. ACM. 2.2.2

[SP08] Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pages 336–347. Springer Berlin Heidelberg, 2008. 2.2.2

[SPSL13] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A Datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, September 2013. 2.2.2, 3.6.1

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990. 6.5.1

[TZ93] E. Tick and X. Zhong. A compile-time granularity analysis algorithm and its performance evaluation. *New Generation Computing*, 11(3):271–295, 1993. 2.2.4

[Ull90] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. 1990. 2.2.2

[Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997. 4.1

[War83] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983. 6.5.1

[War87] D. H. D. Warren. Or-parallel execution models of Prolog. In *II and Colloquium on Functional and Logic Programming and Specifications (CFLP) on TAPSOFT '87: Advanced Seminar on Foundations of Innovative Software Development*, pages

243–259, New York, NY, USA, 1987. Springer-Verlag New York, Inc. 6.5.1

[WCPW03] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical report, 2003. 4.5

[WCPW04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer Berlin Heidelberg, 2004. 4.5

[WS88] Ouri Wolfson and Avi Silberschatz. Distributed processing of logic programs. *SIGMOD Rec.*, 17(3):329–336, June 1988. 2.2.2

[WS98] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):409–10, 1998. 6.4.1

[WSBC04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services*, pages 99–110, New York, NY, USA, 2004. 3.6.2

[WSD07] Pieter Wuille, Tom Schrijvers, and Bart Demoen. CCHR: the fastest CHR implementation, in C. In *Workshop on Constraint Handling Rules*, pages 123–137, 2007. 6.5.3

[WXDG13] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. CIDR'13, 2013. 3.6.1

[YL12] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, MDS '12, pages 3:1–3:8, New York, NY, USA, 2012. ACM. 6.4.1

[ZWC91] W. Zhang, Ke Wang, and Siu-Cheung Chau. Data partition: a practical parallel evaluation of Datalog programs. In *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pages 98–105, Dec 1991. 2.2.2