Responsive Parallel Computation

Stefan K. Muller

CMU-CS-18-120

September 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Umut Acar (Chair)
Guy Blelloch
Mor Harchol-Balter
Robert Harper
John Reppy (University of Chicago)
Vijay Saraswat (Goldman Sachs)

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

This research was sponsored by Intel, Microsoft, and the National Science Foundation under grant numbers CCF-1320563, CCF-1408940, and CCF-16294444. 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, the U.S. government or any other entity.

Keywords: Parallelism, Concurrency, Cost Semantics, Work Stealing, Response Time

Abstract

Multicore processors are becoming increasingly prevalent, blurring the lines between traditional parallel programs, which use *cooperative threading* to reduce execution time, and interactive programs which use *competitive threading* to increase responsiveness. Many applications, from games to database systems, can benefit from a combination of the cooperative and competitive threading models. Unfortunately, languages and tools such as cost models designed for cooperatively threaded programs do not extend easily to features of competitive programs, such as thread priorities.

In this thesis, we develop a model that extends the cooperative paradigm to account for features of competitively threaded programs. The contributions of this model span several levels of abstraction. First, we include a cost model that extends existing models of cooperative threading in order to allow programmers to reason about the parallel running time and the responsiveness of interactive parallel applications. Second, we propose a language that neatly combines abstractions for both forms of threading, and enables reasoning about efficiency and responsiveness at the level of the source code. Third, we develop a scheduling algorithm that efficiently handles threads with both responsiveness and throughput requirements. Finally, we implement the language as part of a compiler for Standard ML, and evaluate it on a benchmark suite including a number of realistic interactive parallel applications.

Acknowledgments

My graduate school career and this thesis would certainly not be what they are if it weren't for my advisor, Umut Acar, who took an excited first-year student with a crazy idea and helped him navigate through initial rejections, several course changes and eventually to a successful thesis project. Thanks so much for all your time and energy over the years!

Throughout grad school, I also had the privilege of working closely with Bob Harper and Guy Blelloch, whose expertise was also enormously helpful in shaping my research and my career. Thank you both for your help and advice. Finally, thank you to the remaining members of my thesis committee, Mor Harchol-Balter, John Reppy, and Vijay Saraswat, for their invaluable input on this document.

The research that makes up this thesis has benefited greatly from many interactions I've had with, among others I'm probably forgetting, Frank Pfenning, Kristy Gardner, Ziv Scully, Danny Zhu, Noam Brown, Ram Raghunathan, Sam Westrick, and Tom Murphy VII.

One of my first lessons at CMU was that, in grad school, one learns at least as much from one's peers as from professors. For that reason, I must also thank all of the Principles of Programming students, from whom I've learned so much.

Some of the most formative experiences I've had in developing my love of programming languages and parallelism have come from being able to share this love with others. Thank you to Bob Harper for everything that I learned TAing 15-814, and to Umut Acar and Danny Sleator for introducing me to parallel algorithms through TAing 15-210. Teaching 15-150 in the summer of 2018 was an exhilarating and incredibly rewarding experience I'll never forget. Thanks to Frank Pfenning for convincing me to do it, and to Jacob Neumann and the rest of the TAs for making it run so smoothly. Finally, thanks to all of the students in these classes for everything I learned from you.

The grad school experience would have been much less enjoyable (though possibly slightly shorter) without all of the great friends I made along the way. Thanks to everyone who was there through five SCS Musicals, six SIGBOVIKs (SIGsBOVIK?), and numerous ThursDz gatherings, lunches, and Avalon games.

Last but not least, a number of very important people helped support me through this whole process. Hannah, I can't say how grateful I am for you putting up with all of the late nights and stress, and for the kind words and deeds throughout. All of it has meant so much. And, of course, innumerable thanks to my family for everything they've done to get me to this point. I quite literally couldn't have done it without all of you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Admit me Chorus to this history;
> Who prologue-like your humble patience pray,
> Gently to hear, kindly to judge, our play.
>
> *Henry V* (Prologue.33–35)

The increasing proliferation of multicore processors has led to renewed interest in writing parallel programs that can improve throughput by dividing up a computation among multiple processors. This idea of exploiting parallelism to improve throughput, which is commonly called *cooperative threading*, is hardly new; it has been the subject of a large body of work in both academia and industry over the last several decades. The research arising from this problem has led to numerous parallel languages such as Id [92], Sisal [46], Multilisp [60], Cilk [23], NESL [14] and X10 [35], as well as parallel extensions to languages ranging from Java [68, 78] to Haskell [32, 93] and ML [50, 99].

A common feature of all of the above languages and systems is that they allow the programmer to express at a high level *where* in the program opportunities for parallelism exist without specifying in detail *how* the parallel computations should be mapped onto physical processors. These details are left to the runtime system. In particular, many of these languages encourage a style known as *fine-grained parallelism* in which the programmer expresses all or most of the opportunities for parallelism, and the system decides dynamically how much of that parallelism may be feasibly or profitably exploited given the available processors. As a result of this style, fine-grained parallel programs may spawn millions of threads.

In addition to alleviating programmer burden, this high-level style of expressing parallelism enables reasoning about the amount of parallelism available in a program or algorithm, and therefore the speedups that could be achieved by running it in parallel, at an abstract level using *cost models* (e.g., [15, 16, 113]). Furthermore, because the programmer has not specified how the work of a program is to be scheduled on processors, the runtime system is free to schedule the program in a way that maximizes throughput. Many languages and systems do so using low-overhead, non-preemptive techniques such as work stealing [26, 60].

Despite all the advances that cooperative threading has brought, it suffers from a substantial limitation: until now, most of the work on cooperative languages and systems has focused on purely computational workloads. This is largely a reflection of the domains in which par-

allel computing has typically been possible and/or necessary: large batch computations, such as data processing or scientific applications, running on remote clusters. Over the last several years, however, multicore has increasingly entered the mainstream, with multicore chips becoming the norm in consumer desktops, laptops, tablets, phones and even smart watches. As hardware evolves, so does the scope of parallel applications, as consumer applications can now be programmed to make use of multiple processors. This represents a significant opportunity to increase the performance of software, but also presents a substantial challenge: consumer applications are generally interactive and not purely computational, making them incompatible with the traditional techniques of cooperative threading.

Interactive programs have leveraged multithreading for decades, but for very different purposes from cooperative threading. They frequently use threads to hide latency (if one thread is performing a blocking operation like accepting user input or retrieving data from disk, another thread may be scheduled in its place to perform useful work) or to more naturally structure applications that consist of many simultaneous processes (such as a GUI that listens for many events while also performing work). We will refer to such uses of threads as *competitive*, in contrast to cooperative threading. Because the goal of these uses of threads is generally not to increase throughput, competitively threaded systems may be implemented by time sharing on a single processor. In this case, threads may be assigned different priorities so that threads performing essential user interaction can be scheduled more frequently to improve the responsiveness of the program. It is in this sense that the threads are *competing* for the computational resources rather than *cooperating* to perform a single large task. Although it is useful for interactive applications, competitive threading is generally not equipped to handle the scale of threads generated by fine-grained parallel applications, nor does it enable the same kind of cost models.

Many applications perform user interaction as well as substantial parallelizable computation. These applications could benefit significantly from a combination of the throughput improvements and models that come from the cooperative threading model with the responsiveness that comes from the competitive threading model. A particularly common paradigm is an application in which many computational threads run in the background alongside one or more foreground interaction threads. Some examples of this paradigm are:

- A database system must respond quickly to queries in the foreground while performing heavily computational tasks such as indexing or compressing in the background. The background computation may be highly parallelizable.
- A computer game takes input from the user through the mouse and keyboard (and possibly from remote players over the network). To maintain a high-quality user experience, the game must react quickly to user input. At the same time, it is computing the strategy for an AI player in the background using a large, parallelizable, search algorithm.
- A robot path planner can be designed using a hierarchical structure, as suggested by [73], with a slow planning algorithm running in the background to compute an overarching long term plan (e.g., which hallways to use to move between two rooms in a building) and a faster planning algorithm running responsively in the foreground to compute a more immediate plan (e.g., navigating around obstacles and people in a room or hallway). Both planning algorithms may be parallel.

To demonstrate the extent to which existing systems for cooperative and competitive thread-

```
fun fib n =
  if n <= 1 then 1
  else
      let val (a, b) = fork (fn () => fib (n - 1),
                             fn () => fib (n - 2))
      in
          (a + b)
      end

fun quest n =
  if n <= 0 then []
  else
    let val _ = print "What is your name?"
        val nm = inputLine ()
        val _ = print "What is your quest?"
        val qu = inputLine ()
    in
      (nm, qu)::(quest (n - 1))
    end

fork (fn () => fib 42, fn () => quest 100)
```

Figure 1.1: An interactive parallel program in Parallel ML.

ing are not designed to handle such a combination of the two models, consider the simple example of a program that computes the 42^{nd} Fibonacci number while also interacting with the user in a loop. This example is contrived, but the structure is an abstraction of the paradigm above, with one interaction thread and many background computation threads. We could write this program in a language like Parallel ML (using the construct `fork` to execute two functions in parallel), as shown in Figure 1.1.

Because Parallel ML has no way of distinguishing the hundreds of millions (approximately $\varphi^{42} \approx 600,000,000$) of Fibonacci threads from the single interaction thread, the interaction is likely to be starved and the program may become unresponsive. Furthermore, the calls to `inputLine ()` will block waiting for user input. Because Parallel ML threads run on top of one system-level thread per processor, an entire processor will be blocked waiting for the I/O to complete.

We would hardly fare better writing the application in a purely competitive system such as C using POSIX threads (pthreads). C code for the program is in Figure 1.2. If run with a small (less than approximately 20) argument to `fib`, this code remains quite responsive because of the operating system's preemptive thread scheduling. If we desired, we could even set the main thread to a higher priority than the Fibonacci threads to ensure the responsiveness of the interactive loop. Unfortunately, as written (with the argument 42), this program will not even run. Because of the overhead associated with pthreads, the large number of threads spawned by

```c
struct Fibargs { int n; int *ret; };

void *fib (void *arg) {
  struct Fibargs *args = (struct Fibargs *)arg;
  if (args->n <= 1) {
    *(args->ret) = 1;
    return NULL;
  }
  int a, b;
  pthread_t t;
  struct Fibargs aargs, bargs;
  aargs.n = args->n - 1;
  aargs.ret = &a;
  bargs.n = args->n - 2;
  bargs.ret = &b;
  pthread_create(&t, NULL, fib, (void *)&aargs);
  fib (&bargs);
  pthread_join(t, NULL);
  *(args->ret) = a + b;
  return NULL;
}

void quest (int n) {
  if (n <= 0) return;
  char nm[64], qu[64];
  printf("What is your name?\n");
  fgets(nm, 64, stdin);
  printf("What is your quest?\n");
  fgets(qu, 64, stdin);
  quest(n - 1);
}

int main () {
  pthread_t t;
  struct Fibargs args;
  int res;
  args.n = 42;
  args.ret = &res;
  pthread_create(&t, NULL, fib, (void *)&args);
  quest(1);
  pthread_join(t, NULL);
  return 0;
}
```

Figure 1.2: An interactive parallel program in C.

the `fib` function quickly overwhelms the system's resources.

In some ways, both of the above examples are strawmen. We do not expect Parallel ML or pthreads to be able to handle these programs as this was not the sort of application either system was designed to execute, but that is precisely the point. In an effort to make these systems work for our purposes, we could investigate the code for the Parallel ML scheduler to better understand which threads it executes when and tune the program to this understanding (e.g., the scheduler will always execute the first thunk locally, so by switching the arguments to `par`, we could improve responsiveness). We could also simply use a foreign function call to spawn an additional pthread to handle interaction alongside the Parallel ML program. Conversely, we could improve the C program by spawning a fixed number of threads and manually dividing the work of the Fibonacci computation among them. In all of these cases, however, we would be creating unnecessary work. Breaking or circumventing the abstraction barrier of a cooperative scheduler defeats the purpose of working at the high level of abstraction that cooperative languages support, and manually parallelizing a C program duplicates work done automatically by cooperative languages. It would be better to have a language that natively supports this type of program.

Just as the languages described above and their runtime thread schedulers do not have the facilities to allow threads with responsiveness requirements, traditional cost models used for reasoning about parallel programs, do not permit reasoning about responsiveness. These models traditionally represent programs as dependency graphs which represent both the amount of work to be done and the amount of parallelism available. These two measures allow for tight bounds on the amount of time required to schedule the program on several processors. However, at least three important properties of these models prevent them from being directly applied to interactive programs. First, the analyses on these models are inherently global: they can reason about the total running time (or, inversely, throughput) of the program, but not the completion time of individual threads, which is important for discussing responsiveness. Second, although these results do not assume particular scheduling algorithms for determining when to run threads, they do assume particular scheduling *principles* such as the greedy scheduling principle which simply requires running as many available threads as possible. Thus far, all such scheduling principles (except those in our prior work) treat threads uniformly and have no notion of prioritizing interactive threads. Third, the traditional cost models assume that a program spends all of its time performing computational work, while interactive programs can also exhibit *latency*, time when a thread is unable to do work but need not be assigned to a processor. Latency is evident in operations such as I/O and system calls.

Many systems for cooperative threading have, over the years, realized the benefits of adding support for competitive features. The origins of the idea of combining cooperative and competitive threading go back at least as far as Manticore [51], which aimed to unite a number of multithreading paradigms and scheduling policies into a single system. A number of systems, such as Manticore, Concurrent Cilk [123], Go [56] and Sisal [46], have implemented strategies for scheduling threads that may block for a variety of reasons, including I/O and system calls. In addition, some schedulers for languages with fine-grained threads such as those of Parallel ML (e.g., [69, 100, 120, 121]) have investigated assigning priorities to threads.

As far as we are aware, however, none of the systems above have formally bounded the running times of their schedulers using models such as the work-span model described above. In contrast, a major contribution of this thesis is using formal models to bound the cost of programs,

much as the traditional work on cooperative threading does. This goal is summarized in the thesis statement.

Thesis Statement

It is possible to extend existing language and cost models for cooperatively threaded parallelism in a natural way to account for competitive threading constructs, and to design and implement scheduling algorithms that account for both throughput and responsiveness.

To demonstrate this statement, we develop a parallel language, and associated cost models, equipped with:

- A small but powerful set of threading primitives that are useful for both cooperative and competitive paradigms.
- Facilities for assigning priorities to threads in order to improve the responsiveness of certain (e.g., interactive) components.
- A type system to prevent *priority inversions*, in which a low-priority thread waits on a high-priority thread, which could harm responsiveness.
- A notion of *fairness*, which keeps high-priority interactive threads responsive without starving low-priority computation threads.
- The ability to *hide the latency* of a user-level thread that performs I/O or a blocking system call by switching to another thread that can do useful work.

Most of this thesis addresses the many issues associated with extending cooperative threading systems to handle thread priorities, and so we further introduce these issues in the following section.

1.1 Thread Priorities

Most competitive threading systems, such as pthreads, provide facilities for assigning priorities to threads, so that threads with essential responsiveness requirements (such as real-time deadlines) receive enough processor time. Such a feature could solve the problem with our interactive Fibonacci example by allowing us to assign a higher priority to the interaction thread to ensure that it runs. Unfortunately, most existing approaches to thread priorities have shortcomings in at least two areas: modularity issues and priority inversions [97]. These issues pose difficulties in writing standard competitive code with priorities, but become especially challenging when we attempt to add priorities to cooperative threading.

Most prior thread priority implementations allow priorities to be selected from among integers in a fixed range. For our Fibonacci example, two priorities (0 and 1) would suffice. Consider instead an email client that performs three tasks simultaneously: compressing old emails in the background, running an event loop (to perform user interaction) in the foreground and asynchronously sending an email. If we wish for the sending of the email to happen at an intermediate priority higher than the compression but lower than the event loop, we would need a third priority. Furthermore, if using an intermediate priority for sending emails were a design decision made in the middle of development, a programmer might need to globally reorganize

the code base to switch the event loop from priority 1 to priority 2. This reorganization may not even always be possible, for example, if the event loop used code from a closed-source library that hardcoded it to run at priority 1. This example shows how the use of a fixed set of priorities hinders modular software development, a cornerstone of the high-level languages in which cooperatively threaded programming is generally done.

The problems in the above example could be slightly alleviated by simply offering a larger set of priorities, as many practical systems do. For example, implementations of the pthreads API may support schedulers with as many as 100 priorities. This surfeit of priorities poses its own set of problems, because priorities cease to have intuitive meanings. If there are 100 priorities, should an event loop be 80 or 90 or 91? A survey by Hauser et al. [65] suggests that programmers had trouble assigning meaning to even the 7 priorities used in the Mesa system. These issues are only compounded by the lack of modularity: all of the programmers working on different parts of a system need to agree on what priorities to use for what operations, because these decisions have global repercussions across the system.

Several authors have argued that a partially ordered, as opposed to totally ordered, set of priorities is more modular and intuitive (e.g., [9, 47]), but such a priority model is rarely supported in practice (see Section 2.3.1 for one partial exception). The language model we develop in this thesis supports such a model by allowing threads to be annotated with priorities from a programmer-defined partial order. Programmers define a set of priorities for a given program, and specify only those ordering constraints between priorities that are meaningful in the program. This allows thread priorities to closely match the intuitively desired behavior of the threads. For example, a programmer could specify the total ordering on the three priorities of the email client (even if the event loop priority is defined in a library, since priorities are abstract symbols in the program and not fixed integers). If a new module added to the email client needs to run at a new priority higher than that of the compression thread, the programmer can, in two lines of code, define the new priority and specify the additional ordering constraint without deciding how the new priority relates to every priority already in the program.

Most implementations of thread priorities can also suffer from priority inversions, in which a high-priority thread is delayed waiting for a low-priority thread to execute. Priority inversions can be extremely costly in practice. For example, Mars Pathfinder, a spacecraft that landed on Mars on July 4, 1997, suffered from a priority inversion in its software that caused the on-board computer to crash and restart. It took over two weeks to diagnose and patch the problem from Earth. In the context of cooperative threading, where we desire a formal accounting of cost, priority inversions pose a challenge: in the presence of priority inversions, it is impossible to prove that a program will remain responsive. In this thesis, we equip our responsive parallel language with a type system to prevent priority inversions that would violate the bounds guaranteed by our cost models.

1.2 Contributions

The contributions of this thesis fall into four main categories, following the thesis statement: language models, cost models, a scheduling algorithm and a runtime implementation.

Language models. We present a language PriML for writing fine-grained parallel programs that perform responsive interaction. Programmers express parallelism by *spawning* and *synchronizing* lightweight threads. The language tracks priorities through the execution of a program using a monadic separation between *expressions*, which evaluate independently of priority, and *commands* which operate at a certain priority. This separation allows the type system of PriML to rule out thread synchronizations that would cause a priority inversion. This work builds on our PLDI 2017 paper [87], which considers only two priorities. Most of the work in this chapter can be found in our ICFP 2018 paper [88].

Cost models. We extend traditional dependency graph-based cost models for cooperative parallel programs to account for I/O operations that incur latency, and for threads of differing priorities. We use these cost models to give bounds on both the parallel execution time of the whole program and the response time of individual threads. These bounds reflect the ability of a scheduler to hide the latency of I/O-blocked threads, as well as the fact that the response time of high priority threads should not depend on the amount of computation at lower priorities, assuming the absence of priority inversions.

We then give a language-based cost semantics that allows the bounds above to be applied to PriML programs. We show that the type system of PriML ensures the invariants required for the response time bounds to hold. This work builds on our SPAA 2016 paper [86] on cost models for blocking I/O and latency hiding, as well as our PLDI 2017 two-priority paper and our ICFP 2018 multi-priority paper.

Scheduling algorithm. We give an algorithm based on randomized work stealing for scheduling threads according to their priorities and for hiding the latency of blocking operations. Although a formal analysis of the algorithm's running time is out of the scope of this thesis, we explain some intuitions for why we believe the algorithm is efficient.

Implementation. We have implemented the above as an extension to Parallel ML. The implementation consists of three parts:

- A runtime system that implements the scheduling algorithm described above.
- A threading library for Parallel ML that implements the threading and priority features of PriML, and contains an I/O library with operations that support latency hiding.
- A compiler from PriML to Parallel ML that implements the type system to prevent priority inversions at runtime.

We present a thorough evaluation of the implementation for both expressiveness and efficiency. The evaluation includes a suite of parallel interactive benchmarks, ranging from small synthetic examples to an implementation of the hierarchical path planner described earlier, consisting of over 1,000 lines of ML code.

1.3 Outline

The remainder of the thesis proceeds as follows:

- In Chapter 2, we summarize in more detail the prior work that forms the basis of this thesis, including prior work on cooperative threading. We also describe other work related to the goals of combining cooperative and competitive threading.
- Chapter 3 contains the graph-based cost model for responsive parallelism and results bounding the throughput and response time of responsive parallel programs at the level of cost graphs.
- Chapter 4 begins with an overview of the PriML language before presenting a core calculus λ^4 that captures the essence of responsive parallelism and is used to show various safety properties of the type system.
- Chapter 5 describes a cost semantics that analyzes λ^4 programs to produce models of the form presented in Chapter 3. We show that the responsiveness and throughput bounds given earlier for cost graphs apply to well-typed (and therefore priority inversion-free) programs.
- Chapter 6 presents over the prioritized scheduling algorithm.
- Chapter 7 describes the implementation of the runtime scheduler, the threading library and the PriML compiler.
- Chapter 8 provides detail on the hierarchical motion planning case study, with a description of its implementation and empirical evaluations.
- Chapter 9 gives a more thorough empirical evaluation of the runtime scheduler using a suite of small and medium-sized benchmarks.

Chapter 2

Background and Related Work

Sir, a whole history.

Hamlet (III.2.324)

The prior work on which this thesis builds, and the work which solves related problems, falls into several categories. We begin with a discussion of prior work on cooperative multithreading, including language models for cooperative parallelism as well as the cost models on which we build in this thesis. We then discuss other work that has combined elements of cooperative and competitive threading, followed by a discussion of research in other areas that is related to the techniques used in this thesis, including work on competitive threading.

2.1 Cooperative Multithreading

Throughout this thesis, we use the terms *cooperative threading* or *cooperative multithreading* to refer to a setting in which multiple threads are used to increase the throughput of a computation. As the name suggests, the threads *cooperate* to complete the computation; since all threads are working toward the same goal, it is not necessary or beneficial for them to compete for resources. In Subsection 2.1.1, we discuss the programming paradigms developed for cooperative threading, and the many languages and systems in which these paradigms have been implemented.

The remainder of the section further details two benefits of cooperative threading, both to justify our focus on cooperative systems and to introduce two major lines of work on which this thesis builds: cost models, and scheduling algorithms. In Subsection 2.1.2, we describe at a high level the cost models that enable reasoning about the running time of cooperatively parallel programs when run on multicore computers. These cost models form the basis of much of the work of this thesis. In Subsection 2.1.3, we describe some of the scheduling techniques that have arisen in cooperative threading systems, and the analyses that have been used to show their efficiency.

$$\frac{M \mapsto M'}{M\ N \mapsto M'\ N} \qquad \frac{N \mapsto N'}{(\lambda x.M)\ N \mapsto (\lambda x.M)\ N'}$$

(a) Left-to-right evaluation.

$$\frac{M \mapsto^{0,1} M' \qquad N \mapsto^{0,1} N'}{M\ N \mapsto M'\ N'}$$

(b) Parallel evaluation.

Figure 2.1: Two reduction strategies for the lambda calculus.

2.1.1 Cooperative languages

The languages, libraries and systems we discuss in this section are united by two key features. First, the primary purpose of the threading or parallelism constructs they introduce is improving throughput, rather than structuring code or reducing latency. This implies that the systems must, at least in principle, be able to run on multiple processors or machines. Otherwise, it would be impossible to make use of the benefits of parallelism. Second, the languages don't require programmers to explicitly manage threads, though they may allow or require programmers to give hints as to where parallelism should be introduced.

Within this section, we divide languages into categories roughly based on the sorts of parallelism constructs they introduce. In the process of discussing these languages, we introduce the parallelism constructs which will be important in the language developed in this thesis.

Implicit Parallelism

By way of introduction, we begin, somewhat counterintuitively, with a class of languages which, by our definition, are parallel, but don't explicitly introduce any threading constructs at all! In particular, most purely functional languages can naturally be considered parallel: since evaluation doesn't have side effects, the order in which subexpressions are evaluated is immaterial, and in fact subexpressions may safely be evaluated in parallel. The prototypical example is the untyped lambda calculus, which admits a number of possible reduction strategies. Figure 2.1 shows two sets of reduction rules for applications, which result in different reduction strategies. In (a), the left subexpression is reduced completely before the right subexpression is evaluated. This "left-to-right" strategy is often shown in the literature, but it is equally possible to use a "right-to-left" strategy, or the parallel strategy shown in (b). In this rule, the $0, 1$ superscripts indicate that either or both subexpressions may take a step in a single step of the complete expression. This allows parallel evaluation (though does not require it; we could define the rule more precisely to require parallel evaluation when possible, but do not do so here for the sake of brevity).

This property of naturally facilitating parallel evaluation was noted by Burton and Sleep [26] as a benefit of functional languages: while they may not be as fast as their imperative counterparts, their evaluation can easily be sped up through parallel evaluation. Parallelism achieved through parallel evaluation of subexpressions rather than explicit programmer effort is often called *implicit* parallelism. Implicit parallelism is exemplified in practice by Id [7], or Irvine Dataflow, a language that dates back to 1975. Its user manual states that "for most subexpressions that have multiple sub-expressions, all sub-expressions are evaluated, and they

are evaluated in parallel." [92] (Id also introduces non-functional features for which evaluation order is important, but we will not discuss these here.) The ideas of Id were carried on into the 1990s by its spiritual successor, pH (for parallel Haskell, though this is not to be confused with other parallel dialects of Haskell which is discussed later). The pH language combines Id's parallel-by-default execution with Haskell syntax.

Sisal [46], first specified in 1983, is somewhat more explicit in the way parallelism is specified, in that parallelism is introduced by specific language constructs (e.g., for loops, streams). While the syntax of Sisal appears imperative, all variables are single-assignment and must be fully defined before use. These restrictions ensure that programs are deterministic and that the compiler can easily determine data dependencies in order to compile the program into a dataflow graph that can be executed in parallel.

Implicit parallelism is attractive in many ways: it requires no additional programmer effort beyond writing a functional program, and it exposes a great deal of parallelism. However, this style also has notable drawbacks. First, it is very difficult to know or control what is evaluated when. This is not a problem if one stays within a functional language (or the functional subset of a language), but in practice many applications use state or other side effects. Side effects become essential, in particular, when we consider interactive applications. Second, the amount of parallelism generated is enormous and impossible to control (without adding more features to the language, which defeats to some extent the idea of implicit parallelism). As we will soon see, large numbers of parallel threads can overwhelm a scheduler and much work has been done on controlling the amount of parallelism generated by a program.

Data Parallelism

Another early parallel programming paradigm was data parallelism, in which an operation is applied in parallel to all elements of a collection. This paradigm mapped nicely to parallel hardware of the time (and is reflected today in GPU programming, which we will not discuss in detail). The data parallel (or "collection-oriented") paradigm dates back as early as the 1960s with APL [70] and continued with languages like C*, CM-Lisp and Paralation Lisp. A wide range of applications can make effective use of data parallelism, but for more general purposes, data parallelism exposes a trade-off. Some languages, such as C*, disallowed nested collections. This restriction matched the hardware, which could not efficiently support nesting, but restricts expressiveness. For example, one might naturally wish to parallelize a recursive algorithm such as Quicksort by simply making the two (independent) recursive calls in parallel, but such an approach is naturally nested. Other languages, such as CM-Lisp and Paralation Lisp, supported nested collections, but could not implement them efficiently in parallel [14]. To remedy this situation, Blelloch and Sabot [13] developed the technique of *flattening* nested parallel collections into flat collections that could be efficiently operated on in parallel by contemporary hardware.

NESL [14, 18] was designed in the mid-1990s with nested data parallelism in mind. The core data type of NESL is the sequence. Sequences can contain elements of any type, including other sequences (hence *nested* parallelism), and support a large number of primitive operations which can be performed in parallel. For example, one can encode the parallel recursive Quicksort example in NESL by building three sequences consisting of elements less than, equal to, and greater than the chosen pivot, nesting the "less than" and "greater than" sequences in another

sequence, and applying Quicksort recursively to this nested sequence.

Later work addressed remaining flaws in data parallel languages. First, work on Nepal extended the ideas of flattening to support richer types than were allowed in NESL, such as algebraic data types [30, 31]. This project continued as Data Parallel Haskell [32], which was made available as an extension of GHC. Separate work developed the technique of *data-only* flattening, which is more suitable to modern multiprocessors, and implemented it in the Manticore system [11].

Task Parallelism

Data parallelism is useful for applications whose parallelism is derived from applying operations uniformly to collections of data. In many applications, however, some or all of the parallelism is derived from performing different operations in parallel. For example, many of the applications which interest us in this thesis perform a large computation (which may or may not fit the data parallel paradigm) in parallel with one or more threads performing interaction with the user. This form of parallelism is permitted by a paradigm known variously as *task* or *control* parallelism, which is the main focus of this thesis.

Many abstractions for task parallelism have been studied in prior languages and systems. We derive the threading mechanism of this thesis from the abstraction of *futures*. A future is a way of deferring a computation: the expression `future` e, for example, immediately returns a handle to a future which will eventually compute the value of e. In this way, futures are closely related to lazy evaluation. The primary difference is that, if there are available processors, a future will evaluate in parallel with continued evaluation of the program (as in implicit parallelism, if the program is purely functional, it doesn't matter when evaluation is performed). The first parallel implementation of futures was in Multilisp [60], a parallel dialect of Scheme. Multilisp inherits from Scheme the ability to perform operations on shared mutable state. The inclusion of state was a conscious decision in the design of Multilisp, and motivated the use of explicit, rather than implicit, parallelism to aid reasoning about when evaluation occurs.

In Multilisp, and many early implementations of futures, the *use* of future values is implicit. If a program, for example, binds `future` e to a variable x and later uses x in a way that requires a concrete value, e.g., integer addition, program evaluation blocks at the use site until the future has completed evaluation. A use of a future value which only requires partial information, for example a pattern match which only needs to know which pattern matches the value, only needs a future to be partially evaluated. This allows futures to be used effectively for a form of pipelining in which a parallel thread continues evaluating the future, producing intermediate values which are consumed asynchronously by the client of the future [17, 60]. To make the types of expressions, and the cost of evaluating them, easier to reason about, later languages introduced an explicit *touch* or *force* operation which blocks evaluation until the value of a future is available. Futures have become a popular and widely-used abstraction for parallel programming. They are supported by a number of modern languages including .NET [27], Scala, Java and C++, and are in use at companies such as Twitter [116] and Facebook [55]

Futures are quite general and powerful, but this power comes at a cost: the general dependency graphs they can induce defeat many of the optimizations on which efficient schedulers for task parallelism rely [1]. One way of taming the power of futures is an abstraction known

as *async-finish* or *spawn-sync*. This type of parallelism is exemplified by Cilk-5 [54], another influential parallel language (the original Cilk release [23] used a different model in which continuations of threads had to be specified explicitly). In async-finish style, threads are spawned like futures, but must be joined with their parent thread by an explicit *finish* or *sync* operation that waits for the completion of all threads spawned in a certain scope. In Cilk, the `sync` operation syncs on all threads spawned by the current Cilk procedure (analogous to a C function). X10 [35] explicitly denotes the scope of threads using `finish {}` blocks. All threads spawned within the scope of a `finish` must complete before evaluation proceeds out of the block. Async-finish parallelism results in a more regular structure of dependencies but still provides substantial power.

More structured still is *fork-join* or *nested* parallelism which enforces a strict nesting of subcomputations. A *fork* operation, e.g., `fork(e1, e2)`, runs `e1` and `e2` in parallel and waits until both complete before returning both values and proceedings. Each of the subcomputations may themselves fork nested parallel computations, but parallel threads may not escape the scope of their parent thread. Fork-join parallelism is strictly less expressive than futures and async-finish, in that both of these styles can be used to encode fork-join in a straightforward way, but is still useful for many applications (including most of the applications explored in this thesis), and generates a form of highly structured parallelism that results in efficient scheduling.

The styles above are not mutually exclusive: many modern parallel languages and libraries provide facilities for multiple of them, allowing programmers to use a more structured construct like fork-join most of the time for efficiency and fall back to a more expressive construct, such as futures, when this is necessary or convenient for the application. Habanero Java, a parallel language based on X10 which exists in both a stand-alone form [28] and as a Java library [68], includes the async-finish style of X10, as well as futures, loop-based parallelism and several other constructs. Other flexible task-parallel libraries have been developed for Java [78], as well as .NET [79] and Standard ML [112]. The Parallel ML language of the Manticore system [49, 50] supports task parallelism as well as other, more competitive, parallelism features. We discuss the Manticore project later in this chapter.

Throughout the above, and in the introduction, we have alluded to both cost analyses for cooperatively threaded languages and schedulers for these languages. As outlined in the thesis statement, a substantial part of the contribution of this thesis focuses on extending existing cost models and scheduling algorithms to account for competitive threading constructs. We therefore introduce existing approaches to cost analyses and scheduling extensively in Sections 2.1.2 and 2.1.3, respectively. We begin each section by briefly expanding on why each topic is an important component of cooperative parallelism, and then describe existing approaches.

2.1.2 Cost Models

Cost models serve two major purposes. First, they allow one to statically reason about the resource usage (time, space or other resources) of programs. Second, they serve as evaluation metrics for an implementation. For example, in the previous section, we discussed various features that hindered or allowed efficient scheduling algorithms. However, without prior knowledge of how the scheduling algorithm can or should perform, there is little we can say about whether a scheduling algorithm is efficient. We can implement it and see that it performs better

than or comparably to another implementation, but without the theoretical knowledge that a cost model provides, we have no way of knowing if both implementations suffer from asymptotic performance issues. Thus, before we discuss the scheduling algorithms that allow for efficient execution of cooperatively threaded programs, we discuss the cost models that allow us to reason about whether an execution is efficient.

Traditionally, cost models for parallel programs are based around the concept of a directed acyclic graph (DAG) showing the dependencies in a program. From this DAG, we can derive cost metrics that predict the performance of the program. We discuss these models and the results that use them to predict performance, and then describe techniques for analyzing a source program in a parallel language to produce an appropriate dependency DAG.

DAG-based Models

The idea of representing a parallel program as a directed graph goes back decades (e.g., [21, 22, 58]). In DAG models of parallel programs, each vertex represents a unit of sequential computation. The units themselves are not important, but we generally make the simplifying assumption that all vertices take the same amount of time to execute (a vertex may, for example, correspond to a single processor cycle). An edge between two vertices u_1 and u_2 indicates that the operation corresponding to u_1 must occur before the operation corresponding to u_2. If two vertices u_3 and u_4 don't have a path between them, they may execute in parallel.

For example, consider the pseudocode below which calculates the 3^{rd} Fibonacci number in parallel:

```
1 fib(n):
2   if n <= 1:
3     return 1
4   else:
5     (a, b) := fork (fib (n - 2), fib (n - 1))
6     return a + b
7
8 main():
9   return fib(3)
```

The DAG for this code is shown in Figure 2.2. As will be our practice in drawing DAGs in this document, we relax the requirement that each drawn vertex consist of a single unit of work. To make it tractable to present the figures, we collapse long sequential chains of vertices into single vertices shown in the figure. Each vertex is labeled with the function call or operation to which it corresponds.

Two cost metrics of a parallel program may easily be read off of the corresponding DAG. The work, which we notate W (it is also sometimes written T_1 in the literature), is the total number of vertices in the DAG. This corresponds to the number of time units it would take to execute the program on one processor. The span, which we notate S (it is sometimes known as the depth, d, and also sometimes written T_∞), is the length of the longest path in the DAG. Since it corresponds to the critical path of the computation, it is the number of time units that an unbounded number of processors would take to run the program.

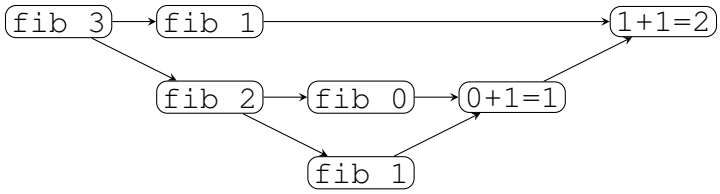


Figure 2.2: The DAG corresponding to the simple Fibonacci program.

A *schedule* is an assignment of vertices to processors at each time step. In a valid schedule, a vertex may only be assigned to a processor if all of its ancestors in the DAG have been assigned at earlier steps. Such a vertex is called *ready*. A schedule corresponds to a strategy for executing a parallel program on a certain number of processors, notated P. The length of the schedule corresponds to the execution time of the program using this strategy on P processors. The problem of finding the shortest schedule for a given DAG and given P is known as the *offline scheduling problem*.

Finding a precise solution to the offline scheduling problem is known to be NP-complete [117], but several simple strategies can yield constant-factor approximations of the shortest schedule. Brent [25] showed that a "level-by-level" strategy of scheduling all of the vertices at a distance n from the root in batches of size P, followed by all the vertices at distance $n+1$, and so on, results in a schedule of length $\frac{W}{P} + S\frac{P-1}{P}$. This is a two-approximation of the optimal schedule since $\frac{W}{P}$ and S are each independently lower bounds on the length of a valid schedule. This result was extended in later work by Eager et al. [42] to all *greedy* schedules. A greedy schedule executes as many vertices as possible at each time step bounded by P and the number of ready vertices.

Language-based Models

The DAG scheduling results above are only useful if we have a dependency graph for a program. A separate line of work focuses on producing such a graph, and therefore the relevant cost metrics and offline scheduling bounds, for a source program written in a parallel language. These approaches build on the ideas of language-based cost semantics [103, 105], static techniques for determining or approximating the resource usage of a program.

Blelloch and Greiner [16] developed a cost semantics for NESL which abstractly evaluates a program to produce not just a value but also a cost DAG of the form described above. This approach, while designed for the data-parallel constructs of NESL, can be adapted in a straightforward way to fork-join parallelism (e.g., [113]) and futures [112, 114].

2.1.3 Scheduling

The results of Brent and Eager et al. described above (which we refer to as "Brent bounds" or "Brent-type theorems") give us an intuition that it should be possible to execute a parallel program efficiently on multiple processors, but do not actually give an algorithm for doing so in an online situation where the DAG unfolds dynamically at runtime. Blelloch et al. [19] present several online scheduling algorithms that provably meet the Brent bound (in terms of number of time steps) in certain cases, but which are based on maintaining global task queues. Global

queues lead to simple algorithms that can be shown to be efficient in theory (the main difficulty is working on tasks in an appropriate order so as to control the number of tasks in the queue at a time and thereby control space usage), but result in a great deal of overhead when considering actual execution time, due to contention and the cost of distributing fine-grained tasks to all P processors.

Work stealing [26, 60, 75] arose in early parallel languages as a distributed alternative to global task queues. In work stealing, each processor maintains a queue of tasks locally. A processor works on tasks from its own queue and pushes newly generated tasks onto its own queue. When a processor finds its queue empty, it becomes a "thief" and "steals" a task from another processor's queue. Blumofe and Leiserson [22] showed that a randomized work stealing scheduler, in which thieves randomly select their victims from the remaining processors, can execute a "fully strict" parallel computation (a computation with the neatly nested structure of, e.g., a fork-join program) with work W and span S in expected time $O(\frac{W}{P} + S)$. The fully strict restriction enforces a certain kind of well-structured nesting on DAGs. Fork-join computations, for example, are fully strict. Later work [6] extended this analysis to general DAGs and multiprogrammed environments, in which the operating system might assign the work stealing scheduler only some of the available processors at any given point in time. The key insight in these analyses is that steals should be rare, so that processors spend most of their time doing useful work. Most of the overhead of scheduling can be associated with steals, so sequential work (which makes up the bulk of the time) can be fast.

Several lines of research (e.g., [2, 106, 124] have extended or adapted work stealing to improve performance in certain contexts. Still other work uses somewhat different approaches to control the potential explosion of parallelism and the performance degradation that comes from prematurely parallelizing. For example, in Workcrews [118], processors request help from other processors; if there are no other processors available to help, the requesting processor is able to continue performing fast sequential work.

2.2 Toward Competitive Multithreading

In the previous section, we described how existing work on competitive threading allows programmers to write parallel programs using high-level abstractions, reason about their time and space usage using elegant cost models, and run them efficiently using provable scheduling algorithms. However, all of this research assumes a fairly limited computation model. In particular, none of this work is designed for situations in which threads might block or have resource requirements other than throughput (or total completion time, as bounded by Brent's Theorem). The goal of this thesis is to extend the abstractions and models discussed in the previous section to handle these sorts of competitive features. In this section, we discuss prior research that also works toward achieving this goal, and compare and contrast these prior approaches with the approach taken in this thesis.

Latency-incurring Operations. Many programs perform operations, such as I/O, that incur latency, i.e., a period of time in which the calling thread is unable to run but is not performing useful computational work. Competitive threading systems usually perform *latency hiding* in

such cases by scheduling another thread that is able to run in place of the thread that is blocked on I/O. Two obstacles prevent cooperatively threaded systems, as described in the previous section, from gracefully performing latency hiding: first, work stealing algorithms are generally not set up to do so. Without special attention, if a thread in a work stealing scheduler performs a blocking I/O operation, the entire processor on which that thread was running will be blocked until the operation completes. Second, the DAG-based cost models of Section 2.1.2 have no way of expressing an operation that takes time but is not computational work.

In prior work [86], we addressed both of these issues. That work extended the DAG model with a notion of latency which counts toward the span, but not the work. It also gives a latency-hiding work stealing algorithm that provably executes in time $O(\frac{W}{P} + SU(1 + \lg U))$, where U is the maximum number of simultaneous latency-incurring operations.

Concurrent Cilk [123] uses a work stealing scheduler that distinguishes between tasks, which are the units of work stealing, and lightweight threads. Lightweight threads have their own work stealing state and can easily be switched on and off of processors, but still result in less overhead than system threads. When a task executes a blocking operation, it is automatically promoted to a lightweight thread. The Concurrent Cilk work does not model or prove bounds on execution time.

BATCHER [3] uses a lighter-weight approach to work stealing and gives proven runtime bounds, but for a particular type of latency-incurring operation: batched concurrent accesses to a shared data structure. The system performs operations on the data structure in batches to avoid synchronization, but this causes data structure operations to block until the next batch of operations is performed. In BATCHER, threads that block on data structure operations are moved to a separate deque to allow other work to proceed.

Inter-thread communication. A number of systems blur the line between cooperative and competitive threading by providing lightweight abstractions that can be used for expressing fine-grained parallelism, but which also allow inter-thread communication, a feature not accounted for in traditional cooperative threading. Concurrent ML (CML) [100, 102], originally designed as a library for Standard ML but adapted into other settings, exposes abstractions of threads and channels over which threads communicate through message passing. For almost two decades, CML had only been implemented for uniprocessors, making it not usable for the main purpose of cooperative parallelism, that is, increasing throughput. Multicore implementations of CML were developed in 2009 [101] allowing CML to be used for more traditional cooperative threading workloads, but the implementations yielded only modest parallel speedup on eight processors over the sequential performance.

The Manticore project [49, 50] has developed a heterogeneous parallel language that combines several threading paradigms, including NESL-style data parallelism, several task-parallel constructs and CML-inspired explicit concurrent threads. Manticore uses a flexible hierarchical scheduling framework [51] that allows for a range of scheduling policies, including different scheduling policies at different levels of the hierarchy. This, for example, allows an application to use different scheduling policies for different types of threads, which is key to supporting interaction. Our goals in this thesis diverge somewhat from Manticore's in that we wish to support a broad range of applications using a homogeneous set of threading primitives, as opposed

to the large heterogeneous set used in Manticore. We note, however, that Manticore compiles to a lower-level set of primitives consisting essentially of cancellable futures [52], quite similar to the primitives provided by our language PriML. This similarity gives us confidence that our primitives are powerful enough to support a wide range of applications, and allows us to build on ideas developed for compiling and scheduling in Manticore.

Priorities. When some threads in a fine-grained parallel program need to be responsive, e.g., because they are performing user interaction, it is important for the scheduler to have some notion of *priority*. The threads with responsiveness requirements should be given higher priority by the scheduler in order to maintain the user experience. Priorities are a substantial departure from typical cooperative threading because in a very real sense, threads are now distinct and are competing for the resource of the processor. Still, we believe it is possible to add thread priorities to a language and its runtime scheduler while still maintaining the other features of a cooperative language: provable cost bounds, low scheduling overhead and high-level abstractions.

Our recent work [87] extends traditional cooperative scheduling to maximize responsiveness of interactive threads, but considers only two priorities and has no notion of fairness. That paper extends the traditional DAG-based cost model with a way of highlighting high-priority threads with responsiveness requirements, and proposes a scheduling principle analogous to the greedy scheduling principle for ensuring both high throughput and responsiveness. We explore this *prompt* scheduling principle in more detail in Chapter 3. The prior work also does not address the *online* problem of efficiently determining what threads to schedule to maintain responsiveness.

We are aware of no other cooperative languages that provide priorities for the purposes of supporting responsive interaction. Several lines of work, however, have added priorities to a work stealing scheduler for the purpose of improving throughput [69, 120, 121]. Applications for these extensions include search problems where branches may be explored in parallel and some branches appear more promising than others based on some heuristic. Adding the more promising branches to the task queue with a higher priority will cause those branches to be explored first, possibly saving work overall by eliminating the need to explore the remaining branches.

The problem of handling computational threads and interactive threads in the same system arises in Concurrent ML. CML does not introduce priorities explicitly, but treats computational and interactive threads separately in the scheduler, implicitly assigning them different priorities. If a thread blocked on communication in the last round of scheduling, it is deemed interactive and scheduled with higher priority in the next round [100]. This heuristic prioritizes threads that frequently perform interaction without requiring explicit annotations, but it can only distinguish between two classes of threads (computational and interactive), and can't classify threads perfectly.

Distributed and PGAS languages. Partitioned Global Address Space (PGAS) languages such as X10, Chapel [33] and Titanium [122] allow inter-thread communication through direct writes to memory but partition memory into segments (e.g., *places* in X10, *locales* in Chapel and *regions* in Titanium) and distinguish between local and non-local memory access. This makes such languages quite suitable for execution in distributed environments where non-local memory ac-

cess can be substantially more expensive than local memory access. On the other hand, making this distinction requires a slightly lower-level view of parallelism since the programmer must be aware of how work is divided and where each computation will be running, in order to access memory accordingly. Other than this awareness of locality, the three PGAS languages mentioned above still have high-level parallelism models—X10 and Chapel have task and data parallelism similar to other languages mentioned earlier, and Titanium has data parallelism.

Other Contemporary Languages. Many modern languages have facilities for lightweight threads, to various ends. In most cases, the implementation is uniprocessor and so, even if scheduling is cooperative, the purpose of the threading constructs is more like that of competitive threading: to improve responsiveness, allow for asynchrony or structure a program more logically. For example, OCaml 4.06 [81, Chapter 31] (the latest version as of this writing) has two implementations of its threads library, one using lightweight virtual threads (which are scheduled in user space) and one using system threads. In both implementations, scheduling is achieved by time-sharing on one processor.

Python 3.2 [98, Chapter 17.4] added support for futures, with two implementations. One implementation uses a pool of lightweight threads. As with OCaml threads, this implementation is subject to the "global interpreter lock" which prevents multiple threads from concurrently executing Python code. Python also supplies an implementation based on processes. This implementation does allow for usable parallelism, but only very coarse-grained parallelism: the high overhead of spawning and maintaining processes makes it impossible to use such a mechanism for the fine-grained parallelism of most of the examples we consider in this thesis.

Go [56] supports lightweight concurrent threads called *goroutines*. Unlike either of the languages described above, Go has a multiprocessor runtime with a work stealing scheduler [84] that allows Go code using goroutines to leverage parallel hardware to improve throughput. Goroutines do not return values and so must communicate using shared global state, but can otherwise be used for either cooperative-style parallelism or shared-memory concurrency.

2.3 Other Related Work

Up to this point, we have been discussing work based on the ideas of cooperative threading. Of course, there is a great deal of related work in other areas as well. In the remainder of this chapter, we briefly describe the most important areas of related work and provide pointers to other sources of information about this work. This exposition serves to relate this thesis to work in other fields, show where this research sits in a broad context, and briefly explain why solutions used in other fields fall short in achieving the goals of this thesis.

2.3.1 Competitive Threading

Competitive threading, the alternative to cooperative threading, has a wide range of use cases. Common uses are to reduce latency, handle interactive events asynchronously or for programs that are more logically structured as the composition of communicating processes. Because the goal of competitive threading is generally not to improve throughput, competitively threaded

systems may run on a single processor or multiple processors. Hauser et al. [65] have an excellent, though now somewhat outdated, survey of uses of threads in real-world interactive systems. They identify several thread-use paradigms in addition to the ones we mention above, and detail their relative prevalence in code.

One paradigm noted by Hauser et al. is exploiting parallelism, though at the time of that survey, competitive systems were only just beginning to run on multiprocessors. By 2000, another survey [48] showed that several classes of competitively threaded programs, like web browsers and image editing tools, were able to make moderately effective use of multithreading to improve response time. Still, the amounts of thread-level parallelism (TLP, i.e., the number of processors the program could make effective use of) were generally between 1 and 2. Some programs displayed higher (close to 2) TLP for short interactive episodes, but the average TLP for most benchmarks was lower. Ten years later, a followup study [12] showed some improvement, especially among video authoring tools, which were able to achieve TLP as high as 9.0 on some platforms. Most benchmarks, however, still had TLP close to 2.

We see little intuitive reason why competitively threaded programs should not be able to take advantage of multicore machines. As we identified in the introduction, applications from games to database systems to image authoring tools can execute parallel algorithms. The end goal of this work begun in thesis is to improve the state of the art to the point where interactive applications can make use of thread-level parallelism to improve their throughput on multicore hardware.

Priorities

Most modern competitive threading systems provide some facilities for specifying the priorities of threads. The POSIX threads (pthreads) API exposes scheduling policies which may allow as many as 99 distinct priorities. As with most most implementations of thread priorities, priorities are represented by an integer and few guidelines are given for what priority to choose for what tasks.

The Hauser et al. study gives some insights into how programmers use priorities under these conditions. The threading model explored in the study exposed 7 priorities numbered 1 through 7, with 4 being the default. In both systems observed in the study (Cedar and GVX), one of the priorities went unused. The meanings to which programmers assigned priorities differed substantially between the two systems. Cedar threads were observed to use priorities 1 through 4 fairly evenly, while most GVX threads were assigned priority 3. Cedar uses level 7 for interrupt handling while GVX uses 5. As both use level 6 for the scheduler, this distinction seems likely to result in substantially different behavior between the two systems.

The experiences with Cedar and GVX highlight the expressiveness and modularity problems inherent in representing priorities as integers in a fixed set, which have also been observed by other authors (e.g., [9, 97]). An alternative is to represent priorities as a partial order, allowing programmers to specify all and only the ordering constraints between priorities that make sense in the context of the program. Fidge [47] explores this possibility from a theoretical standpoint by extending the process algebra CSP, which provides operators for concurrency and choice, with a notion of asymmetry that prioritizes one operand over the other. Ideally, this would allow programmers to avoid the extreme requirement enforced by most thread APIs that priority

orderings be fully specified. The occam language, whose concurrency model is based on CSP, allows a form of asymmetry but, according to Fidge, goes too far to the opposite extreme, leaving many priorities unspecified and ambiguous and giving the programmer no way to resolve this ambiguity. In this thesis, we present a partially ordered model for thread priorities which allows the programmer to specify any desired thread orderings, but to leave others unspecified if desired.

Priority Inversion

Priority inversion is a significant issue in threading systems. The classic example (e.g., [76]) involves three threads: a thread at low priority acquires a lock, a thread at high priority blocks attempting to acquire the same lock, and a thread at medium priority prevents the low-priority thread from running. The medium-priority thread is therefore preventing the high-priority thread from acquiring the lock and running, contrary to the intention expressed in the priorities. While this is a frequently-cited example of a priority inversion, the actual problem is substantially more general (e.g., [9]) and can occur with just two threads and two priorities. We consider any case in which a higher-priority thread is waiting on a lower-priority thread to be a priority inversion.

Runtime techniques such as priority inheritance, in which a low-priority thread which holds a resource *inherits* the priority of a higher-priority thread that is waiting on the resource, can dynamically avoid priority inversion (e.g., [76, 108]). Babaoğlu et al. [9] describe other techniques for preventing priority inversions in some settings, and also give a formal account of priority inversions in a (partially ordered) priority system. In this thesis, we take a static approach to preventing priority inversion because we are interested not only in preventing priority inversions at runtime but also in enabling static reasoning about the responsiveness of a program. Such reasoning can be hindered if, for example, a long-running background thread can be promoted to high priority at runtime.

2.3.2 Scheduling for Responsiveness

In the Systems Community

There is a great deal of research in operating systems on scheduling threads for responsiveness, including based on priorities. This includes far too much material to be summarized here, so we refer the reader to a comprehensive text by Silberschatz et al. [110]. One issue in particular is determining the proper metric for evaluating the responsiveness of a system. Mean response time is one commonly used measurement, but for many applications it is also important to minimize variance. These same concerns come up in the theory community (see below). The main difference between the domain considered in this thesis and the domain of schedulers in the systems community is that our schedulers must contend with the very large numbers of threads spawned by fine-grained applications, while competitive schedulers typically work with far fewer threads, enabling higher-overhead scheduling techniques.

In practice, a system will have to support many different types of workloads with different and competing resource requirements, just as the scheduler we describe in this thesis has to contend with computational (throughput-bound) threads and interactive (responsiveness-bound) threads. In operating systems, responsive applications come in at least three classes, depending on the

strength of their requirements: hard real-time, soft real-time and best effort. Even within each class, applications and threads may have different priorities. Production schedulers designed to handle different types of workloads under these conditions still struggle to give proper preference to interactive workloads without starving computational workloads. For example, a study of the Unix System V Release 4 (SVR4 UNIX) scheduler ran a text editor, a video player and a batch computation simultaneously while varying the scheduling class to which each was assigned [91]. They found that granting real-time status to the continuously running video player could starve both interactive applications (represented by the text editor) and compute-bound programs, while assigning all programs to the time-sharing class (as expected) results in poor performance for the text editor and video player.

Lottery scheduling [119] attempts to fairly schedule threads by randomly selecting threads to run based on the distribution of abstract resources called "tickets". This is similar to our implementation of fairness, but operates at the level of individual jobs instead of priorities. Their approach also allows tickets to be transferred, e.g., to implement priority inversion. Weighted fair queueing [39] achieves a similar aim of fairly scheduling users with different weights reflecting their priority (though initially in the setting of network gateways rather than process schedulers) but in a deterministic fashion similar to round robin.

Goyal et al. [57] propose a framework for hierarchical scheduling in which, for example, real-time processes can get a fraction p of CPU time and best-effort processes can get a fraction $1-p$. A fair scheduler makes scheduling decisions at the internal nodes of the hierarchy (e.g., deciding whether to work on a real-time or a best-effort process). One suitable scheduler proposed for this purpose is Start-Time Fair Queueing, a variant of fair queueing which schedules process based on a virtual time computed from their actual start time and weight. Within each class (at the leaves of the hierarchy), scheduling decisions can be made using any suitable scheduler. The scheduling principle we discuss in Section 3.4 can be seen as a hierarchy with one non-leaf level consisting of a node for each priority. A randomized algorithm decides which priorities to schedule, whereas scheduling within each priority is done by work stealing.

Borrowed Virtual Time (BVT) [41] is a scheduling algorithm also based on virtual time, but which allows latency-sensitive threads to "warp" back to earlier virtual times, causing these threads to be temporarily scheduled more quickly in order to meet latency deadlines. The goal of BVT is to efficiently handle both computational and interactive threads without starvation or loss of responsiveness. Still, BVT (like all of the schedulers discussed above) is designed for a much smaller number of computational threads than the scheduler proposed in this thesis.

In the Theory Community. The field of queueing theory is also largely concerned with scheduling to maximize responsiveness. Queueing theory generally considers a model in which jobs arrive through some external process, and are queued until they can be operated on by a server. A job's response time is the time between its arrival and its completion. A system may have multiple servers, in which case the system may achieve parallelism by serving many jobs at once, but historically the field has mostly considered jobs to be themselves sequential. For a primer on queueing theory, we refer the reader to Harchol-Balter's book [62]. In the rest of this section, we consider theoretical analyses of queueing systems that allow jobs themselves to be parallel. In these analyses, jobs still arrive through an external process, in contrast to the parallel computing

setting where they arrive intrinsically through spawn operations in the running code.

A substantial body of work abstracts parallelizable jobs into speedup curves, that is, functions $s(k)$ indicating the parallel speedup a job achieves on k processors over its sequential execution. Various algorithms have been analyzed for jobs with speedup curves of the form $s(k) = k^\alpha$ [67], jobs with arbitrary (but uniform) speedup curves [10], and jobs with multiple phases of parallelism [43, 59]. The models studied do not have the same cost metrics of work and span that we use to evaluate parallel schedulers. Rather, most of these results compare a proposed scheduling technique to a theoretically optimal technique and show that the proposed technique approximates the optimal schedule to some factor (sometimes assuming a specified level of "resource augmentation", that is, a factor by which the proposed scheduler is allowed to run faster than the optimal scheduler).

Saifullah et al. [104] present a technique for scheduling parallelizable jobs with real-time deadlines, where each job is represented by a general DAG. The technique decomposes the DAG into a set of sequential tasks to which it assigns intermediate deadlines. This allows the authors to leverage existing scheduling techniques and results to schedule the resulting tasks. The limitation is that this decomposition requires knowing the structure of the DAG when beginning to schedule the job.

Finally, other lines of work are non-clairvoyant in that they can schedule arbitrary parallelizable jobs for which the DAG is not known ahead of time. In this space, various techniques have been developed to optimize for average response time [5, 66], maximum response time [4] and number of jobs missing real-time deadlines [82].

2.3.3 Modal and placed type systems

In Section 4.2.1, we present a modal type system that restricts the use of threads in order to prevent priority inversions. A number of type systems have been based on various modal logics, many of them deriving from the judgmental formulation of Pfenning and Davies [96]. While we did not strictly base our type system on a particular logic, many of our ideas and notations are inspired by S4 modal logic and prior type systems based on modal logics. Moody [83] used a type system based on S4 modal logic to model distributed computation, allowing programs to refer to results obtained elsewhere (corresponding in the logical interpretation to allowing proofs to refer to "remote hypotheses"). It is not made clear, however, what role the asymmetry of S4 plays in the logic or the computational interpretation. Later type systems for distributed computation [72, 90] used an explicit worlds formulation of S5, in which the "possible worlds" of the modal logic are made explicit in typing judgment. Worlds are interpreted as nodes in the distributed system, and an expression that is well-typed at a world is a computation that may be run on that node. Both type systems also include a "hybrid" connective A `at` w, expressing the truth of a proposition A at a world w. They interpret proofs of such a proposition as encapsulated computations that may be sent to w to be run. Our type system uses a form of both of these features; priorities are explicit, and the types τ `cmd`$[\rho]$ and τ `thread`$[\rho]$ assign priorities to computations. Unlike prior work, we give an interpretation to the asymmetry of the accessibility relations of S4 modal logic, as a partial order of thread priorities.

A different but related line of work concerns type systems for staged computation, based on linear temporal logic (LTL) (e.g., [38, 45]). In these systems, the "next" modality of LTL is

interpreted as a type of computations that may occur at the next stage of computation. In prior work [87], we adapted these ideas to a type system for prioritized computation, but that system only considers two priorities: background and foreground. In principle, a priority type system based on LTL could be generalized to more than two priorities, but (because of the "linear" nature of LTL), such systems would be limited to totally ordered priorities.

Place-based systems (e.g., [34, 35, 61, 71, 107, 122]), like the modal type systems for distributed computation, also interpret computation as located at a particular "place" and use a type system to enforce locality of resource access. These systems tend to be designed more for practical concerns rather than correspondence with a logic.

2.3.4 Information flow control

In this work, we track inter-thread communication (through synchronization) and require that high-priority threads do not wait for low priority threads. A similar problem occurs in the area of information flow control for concurrent programs (e.g., [24, 111]), in which systems must ensure that communication cannot cause data to pass from high-*security* threads to low-*security* threads. Type systems for ensuring these properties are generally finer-grained than ours, since they are concerned with the security of individual memory locations and not just overall properties of threads. Muller and Chong [85], however, use a place-based language (see above) derived from X10 to increase the granularity of this tracking by effectively reducing inter-thread communication to message passing and ensuring that this communication is secure. This approach is similar to the one we take in this thesis in the sense that both analyses only need to consider interactions between threads at explicit synchronization points.

Chapter 3

A DAG Model for Responsive Parallelism

> These trees shall be my books,
> and in their barks my thoughts I'll character
>
> *As You Like It* (III.2.5–6)

In the previous chapter, we outlined an approach to representing parallel programs as directed acyclic graphs (DAGs) and analyzing the cost of executing a DAG in parallel using Brent's Theorem and its variants. In this chapter, we extend the conventional DAG model with the ability to assign priorities to threads and to represent operations that incur latency. These two features allow us to use DAGs to represent a large class of parallel interactive programs, such as the interactive Fibonacci program of the introduction. As with the traditional DAG models, we are able to use these DAGs to reason about the total computation time of an interactive program, but in addition, interactive programs have another important cost property that we would like to bound: *response time*.

The main results of this chapter are three Brent-style bounds guaranteeing the computation and response times of parallel interactive programs under various conditions. We begin in Section 3.1 by defining the DAG model for responsive parallel programs. In Section 3.2, we formally define response time and extend the standard notions of work and span appropriately to reason about response time. Sections 3.3 and 3.4 present the Brent-type results for bounding the response time of programs. While the greedy scheduling policy of the previous chapter is sufficient to bound computation time, it does not suffice for any guarantees on response time. We therefore introduce two new scheduling policies, *prompt* scheduling (Section 3.3) and *fairly prompt* scheduling (Section 3.4), which are sufficient to prove responsiveness.

3.1 The DAG Model

In DAG models of parallel programs, vertices represent units of sequential computation and edges represent sequential dependences. Without loss of generality, it is typically assumed that each vertex represents a computation taking a single unit of time (perhaps a single processor clock cycle). These are the units in which we will measure execution time and response time.

The DAG model we develop in this thesis features *threads* in addition to vertices and edges.

A thread is a collection of vertices and their associated edges. The vertices in a thread happen in sequence, allowing us to use a more compact notation to describe the vertices and edges of a thread. We begin by defining this notation, and then proceed to discuss the remaining edges of a DAG, which are those that go between threads. Finally, we put these definitions of threads and edges together into the formal definition of a DAG.

Vertices and Threads. The core feature of the DAG model we develop in this thesis is a *thread*. A *thread* is a sequence of vertices $\vec{u} = u_1 \cdot_{\delta_1} u_2 \cdot_{\delta_2} u_3 \cdot_{\delta_3} \ldots \cdot_{\delta_{n-1}} u_n$, written $[]$ when $n = 0$. In this notation, the vertices $u_1, \ldots, u_n$ represent program instructions or operations, each of which takes one unit of time. The operations must occur in the sequence indicated. The *delays* δ_i are integers greater than or equal to 1 which indicate how long u_{i+1} must wait to run after u_i. In the common case, $\delta_i = 1$ and u_{i+1} may run immediately after u_i (i.e., one time unit later). On the other hand, suppose u_{i+1} requires some external process to occur before it can run, in addition to depending on u_i. This might be the case, for example, if it is waiting for a user input, or for a system call to return. In such a case, $\delta_i > 1$ and u_{i+1} may only run δ_i time units after u_i runs. When $\delta = 1$, we generally omit the delay subscript.

Threads are assigned priorities ρ drawn from a set R equipped with a partial order $\preceq$ (where appropriate, we also use the symbol $\prec$ to refer to the natural corresponding strict order). All of the operations of the thread run at the priority assigned to that thread. We define $Prio_g(u)$ as the priority of the thread to which u belongs in a DAG g.

DAGs as execution records. [1] Because latencies are recorded in the DAG, it may seem that, in order to construct a DAG, we must know in advance how long each latency-incurring operation will take. However, there is no paradox here as a DAG is an *a posteriori* record of a particular execution. Thus, we are simply recording how long the latency actually was in that execution.

Even in traditional DAG models of programs without latencies, there is much information about the execution of a program that is not captured in the DAG but which can determine execution time. For example, differences in inputs or nondeterminism could lead to different branches of conditionals being taken at runtime, with possibly very different costs. A DAG represents the operations that are executed, and their dependency structure, for a *particular execution*, factoring out any such difference in inputs or nondeterminism. Differences in latencies (e.g., whether, in a particular execution, the user takes 3s or 4s to respond to a question) simply add another form of nondeterminism which must be factored out when constructing a DAG (we will see another form shortly, in the timing of when threads complete). Thus, any program corresponds to a family of DAGs, one for each way in which the program could execute. When we say that there is a delay of δ between two vertices u_1 and u_2, we are implicitly referring to a family of DAGs, one DAG for each plausible value of δ (if there are two delays $\delta_1, \delta_2 > 1$, then there is a DAG for each pair of values, and so on). In Chapter 5, we describe how to analyze programs to produce DAGs corresponding to all possible executions.

[1] This set of paragraphs is a slight digression that addresses a common confusion about latencies in DAGs, and may be skipped without loss of continuity.

Figure 3.1: DAGs representing polling.

Edges. In DAG models, edges between vertices correspond to sequential dependencies between operations. We have already seen one type of dependency: between an operation and the following operation in the same thread. We refer to the edge corresponding to such a dependency as a *continuation edge*. Thus, a thread in a DAG corresponds to a series of vertices connected by continuation edges. The delays between operations are indicated by the edge weights on the corresponding edges. An edge with a weight of 1, corresponding to no delay, is called a *light* edge. Edges with weights greater than 1 are *heavy*. When describing a single continuation edge from u to u' with weight δ, we notate it (u, u', δ).

A parallel program consists of many threads, and threads have edges between them indicating inter-thread dependencies. We restrict attention to three types of inter-thread edges:

- *Spawn edges* connect a vertex which spawns a thread to the first vertex of the new thread.
- *Join edges* connect the last vertex of a thread to an operation that waits for it to finish.
- *Weak edges*, like join edges, connect the last vertex of a thread to another operation, but indicate a different sort of dependency. Weak edges will be described in more detail below.

In particular, all inter-thread edges start at the last vertex of a thread or end at the first vertex of a thread. The results of this chapter would likely extend to DAGs with edges originating and/or terminating in the middle of threads, but such edges are not necessary for the development in the remainder of this thesis. We assume all inter-thread edges are light edges (i.e., an inter-thread edge from u_1 to u_2 has weight 1 and u_2 may run the time step after u_1 if its other dependences, if any, are met). Furthermore, if a vertex has a heavy in-edge, we assume it has no other in-edges.

A weak edge from u_1 to u_2 indicates that vertex u_2 was executed after u_1 in the particular execution to which this DAG corresponds, but that u_2 need not execute after u_1 in all executions of this program. Such a situation can occur in a program execution where the operation corresponding to vertex u_2 "polls" the thread whose last vertex is u_1 to see if it completed. This program will correspond to two (sets of) DAGs: in one, u_1 has completed when u_2 executes, and so the poll operation "succeeds". If we represented this fact using a join edge, which, like edges in traditional DAGs, indicates a necessary data dependency, then our timing bounds would have to account for schedules in which u_2 waits a potentially long or unbounded time for u_1 to complete. This is not the desired behavior, however: we should not have to account for such schedules because we know they don't reflect this execution (we discuss *admissible* schedules, which respect weak edges, below). We instead use a weak edge, shown as a dotted line in Figure 3.1a. A poll operation could also lead to another execution—and therefore corresponds to another set of DAGs—in which the thread had not completed when it was polled and the poll failed, in which case there is no dependency edge (Figure 3.1b).

If there is a path from u to u' (using any combination of thread, spawn and join edges), we say

that u is an *ancestor* of u' (and u' is a *descendant* of u), and write $u \sqsupseteq u'$. We define shorthands for a graph with the (proper) ancestors and descendants of a vertex u removed:

$$\nuparrow u \triangleq g \setminus \{u' \neq u \mid u' \sqsupseteq u\}$$
$$\ndownarrow u \triangleq g \setminus \{u' \neq u \mid u \sqsupseteq u'\}$$

If there is no path between u and u' (i.e., $u \not\sqsupseteq u'$ and $u' \not\sqsupseteq u$), the two computations may run in parallel.

It is sometimes necessary to distinguish between *weak* and *strong* paths and ancestors. A path is *weak* if it contains a weak edge. If all paths from u to u', where $u \sqsupseteq u'$, are weak, we say that u is a *weak* ancestor of u' and write $u \sqsupseteq^w u'$. Otherwise, u is a *strong* ancestor of u' and we write $u \sqsupseteq^s u'$. When it is not important whether a vertex is a weak or strong ancestor, we continue to simply use the term "ancestor" and the un-superscripted notation.

Formal definition of a DAG. We now combine the above definitions into the formal definition of a DAG. A DAG is a collection of threads, each identified by a *thread symbol* or *thread name*, for which we use the metavariables a, b and variants. Formally, we model DAGs as quadruples $g = (\mathcal{T}, E_s, E_j, E_w)$, in which $\mathcal{T}$ is a mapping from thread names to a triple consisting of that thread's priority, its sequence of operations, and the delay, if any, before it is ready. We write an element of the mapping as $a \underset{\rho}{\hookrightarrow} (\delta, \vec{u})$. This represents a thread a operating at priority ρ consisting of the operations in $\vec{u}$. If $\delta = 0$, then the thread is ready to run. If $\delta > 0$, the first operation of $\vec{u}$ is delayed and must wait δ more time units before it can run. The next three components of a DAG are the set E_s of spawn edges, the set E_j of join edges and the set E_w of weak edges. We write spawn edges as (u, a), indicating that a vertex u spawned a thread a. This may be considered an edge from vertex u to the first vertex of a. We write join and weak edges as (a, u) to indicate that vertex u joined with or polled thread a. This may be considered an edge from the last vertex of a to vertex u.

Example. Finally, we conclude this section with an example that illustrates the features of the DAG model. We begin with a non-interactive example, and then extend it with interactive features. Consider the following code, which computes the 3^{rd} Fibonacci number using a standard parallel, recursive algorithm. All threads involved are at priority `low`. The syntax is that of PriML, which we describe in the next chapter, but the ideas of the DAG model are language-independent. At each call to `fib n`, a new thread is spawned to run `fib (n - 1)` while `fib (n - 2)` is computed in the current thread. When both results are available, the two threads join to sum the two results.

```
1  fun fib (n: int) =
2    if n <= 1 then
3      cmd[low] {ret n}
4    else
5      cmd[low]
6      {
7        t <- spawn[low] { do(fib (n - 1)) };
```

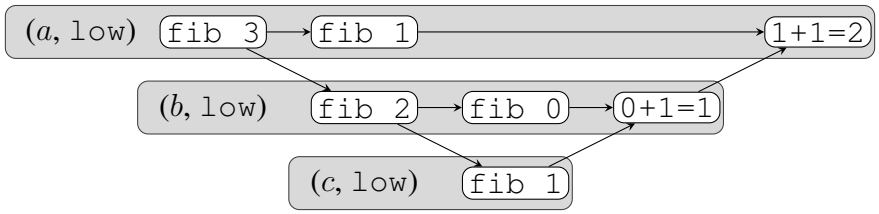


Figure 3.2: The DAG corresponding to the simple Fibonacci program.

```
8         a <- do(fib (n - 2));
9         b <- sync t;
10        ret (a + b)
11      }
12  main {do (fib 3)}
```

The DAG corresponding to this code is shown in Figure 3.2. As before, we collapse long sequential chains of vertices into single vertices shown in the figure. Vertices that are part of the same thread are outlined in gray boxes. Each thread is labeled with its name and priority. Edges to the beginning of a thread, such as the edge from `fib 3` to `fib 2`, are spawn edges, edges from the end of a thread (e.g., `fib 1` to `0+1=1`) are join edges, and edges within a thread are continuation edges.

We now compose the Fibonacci program above with an interactive thread that asks the user two questions and returns the answers to the top level. In the DAG corresponding to this program (Figure 3.3), the interactive thread at priority `high` is shown in a darker box to indicate the higher priority. The heavy edges after the input operations are drawn as thick lines, and annotated with the latencies δ_1 and δ_2. The remaining edges are light; the implicit edge weight of 1 is omitted from the figure.

```
1   fun quest () =
2     let val _ = output "What is your name?"
3         val name = input ()
4         val _ = output "What is your quest?"
5         val quest = input ()
6     in
7       cmd[high] { ret (name, quest) }
8     end
9
10  main
11  {
12    quest_t <- spawn[high] { do (quest ()) };
13    do (fib 3);
14    sync quest_t
15  }
```

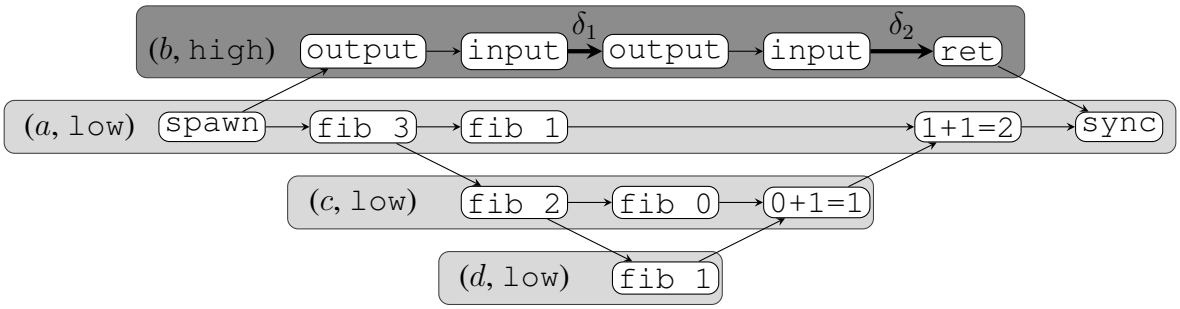


Figure 3.3: The DAG corresponding to the interactive Fibonacci program.

3.2 Response time and well-formedness

This section defines most of the remaining concepts necessary to discuss the response time of threads in a schedule of a DAG. We begin by formally defining terms related to schedules and response time, and then extend the standard notions of work and span into cost metrics that allow us, in the following sections, to prove bounds on response time.

A *schedule* of a DAG simulates the execution of a parallel program on a given number of processors. In determining the schedule, we assume that the entire DAG is given in advance; thus, the results in the remainder of this chapter are *offline* scheduling results (see Chapter 2 for a discussion of the distinction between the offline and online problems). In Chapter 5, we apply this result to executions of the λ^4 dynamic semantics as well.

A schedule determines which processors execute which vertices of the DAG during which time steps. A vertex is *enabled* once all of its parents in the DAG have executed. An enabled vertex may still not be able to be executed, because it may have incurred a delay. A vertex whose parents have all executed and whose delay, if any, has elapsed, is called *ready* and may be executed. Because we assume that a vertex with a heavy in-edge has no other edges, vertices with in-edges fall into two categories. If a vertex has one or more light in-edges and no heavy in-edges, it is ready as soon as it is enabled. If a vertex has a heavy in-edge of weight δ, it is ready $\delta - 1$ steps after it is enabled.

We formally define a schedule below.

Definition 1. A *schedule* is a mapping from time steps (indexed by integers starting at 0) to a set of pairs (p, u) indicating that processor p executes vertex u at this step. A schedule must obey certain rules:

- Each processor may execute at most one vertex per time step.
- A vertex may be executed only if it is ready.

One may think of a schedule as a form of pebbling: if P processors are available, at each time step, place up to P pebbles on ready vertices until all vertices have been pebbled.

Response time and DAG Cost Metrics. Our goal is to show a bound on the response time of threads in schedules. In a given schedule, the *response time* of a thread a, written $T(a)$, is defined as the number of steps from when the first vertex of a is ready (exclusive) to when the last vertex of a is executed (inclusive). Many, though not all, programs start and end with a single thread

(call it a). In terms of the DAG, this means that the DAG has a unique source (the first vertex of a) and a unique sink (the last vertex of a). All fork-join programs, for example, obey this property. For such programs, then the total execution time of the program is the same as $T(a)$.

Intuitively, if our definitions of priority are set up correctly, the response time of a thread running at priority ρ should depend only on the amount of work at priorities not less than ρ: any other dependence would represent a priority inversion. Because the set of priorities "not less than ρ" will be important in several places, we define a special notation for it:

$$\not\prec \rho \triangleq \{\rho' \in R \mid \rho' \not\prec \rho\}$$

The *priority work* $W_{R'}(g)$ of a graph g is defined as the number of vertices in the graph at priorities contained in the set R'.

$$W_{R'}(g) \triangleq |\{u \in g \mid Prio_g(u) \in R'\}|$$

Furthermore, a thread's response time should depend only on the amount of work available in the system while it is running (i.e., not vertices completed before the thread is ready or spawned after it completes). The *competitor work*, $\nleftrightarrow a$, of thread a is the subgraph formed by the vertices that may be executed in a valid schedule while a is active. More precisely, if

$$g = (a \underset{\rho}{\hookrightarrow} (\delta, s \cdot \ldots \cdot t) \uplus \mathcal{T}, E_s, E_j, E_w)$$

then

$$\nleftrightarrow a \triangleq g \setminus (\{u \neq s \mid u \sqsupseteq s\} \cup \{u \neq t \mid t \sqsupseteq u\}) = \uparrow s \cap \nleftarrow t$$

In this notation, the underlying graph, g, is left implicit because it will generally be clear from context.

Combining these two notations, we can say that the response time of a thread a at priority ρ should depend on $W_{\not\prec\rho}(\nleftrightarrow a)$.

For example, in the DAG shown in Figure 3.4, the competitor work of b includes vertices a2, a3, a4, c1, d1 and d2, because all of these vertices may be scheduled in parallel with b1 or b2. However, the response time of b will only be affected by c1, d1 and d2 because thread a is at lower priority. The competitor work of b does not include, for example, a1, c3 or e1 because these vertices must be scheduled strictly before or after the vertices of b.

It may seem counterintuitive that, in the above example, the response time of thread b depends on the work of d and vice versa. However, this must be the case: there exists a schedule in which all of thread d is completed between b1 and b2, and there exists a schedule in which all of thread b is completed between d1 and d2. A worst-case analysis must allow for both of these schedules.

Just as Brent's Theorem relates computation time to both the work and the span of a parallel computation, we will need a notion corresponding to the span, which will capture the property that the response time $T(a)$ cannot be less than the time required to execute the critical path of a thread. The *a-span* $S_a(g)$ of a graph $g \ni a \underset{\rho}{\hookrightarrow} (\delta, s \cdot \ldots \cdot t)$ is the length of the longest strong path in $\uparrow s$ ending at t. We are interested in paths ending at t because these paths must be fully executed before thread a completes, but we do not count ancestors of s because such vertices will have already been executed before a begins and are therefore not counted in $T(a)$. If a program

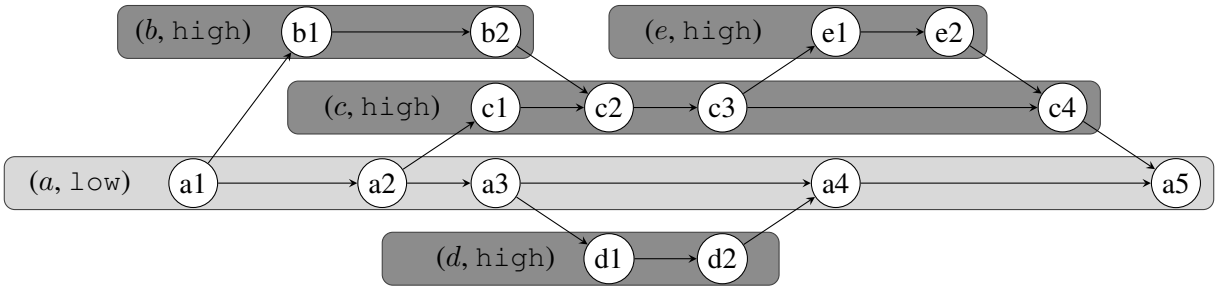


Figure 3.4: An example DAG.

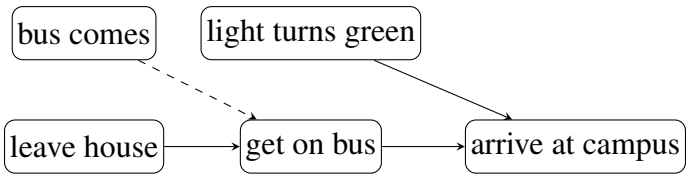


Figure 3.5: An analogy for thinking about weak edges.

starts and ends with a single thread a, then $S_a(g)$ reduces to the standard notion of the span of the graph, i.e., the longest path.

In Figure 3.4, the c-span corresponds to the path b1, b2, c2, c3, e1, e2, c4. The path does not include a1 because this is an ancestor of c1 and must therefore be scheduled strictly before thread c.

The theorems later in this chapter place a bound on the response time of a thread involving the above quantities, under certain conditions. We now describe in detail the two most important conditions required for the bounds to hold: *admissibility* of the schedule and *well-formedness* of the DAG.

Admissibility. The scheduling results of the remainder of the chapter only concern *admissible* schedules.

Definition 2. A schedule of a DAG with a weak edge (u, u') is *inadmissible* if, at any step, u has not executed and u' has no incoming strong edge. Schedules in which such a situation never arises are *admissible*.

Admissibility is important because, in keeping with the intuition behind weak edges, it ensures that we consider only executions in which u' does not wait for u to complete.

To understand weak edges and admissibility, suppose I am an impatient graduate student who does not like waiting for buses. Figure 3.5 shows a DAG corresponding to part of my morning routine, which I might use to calculate how long it will take to arrive on campus from when I leave the house[2]. If all edges in the DAG were strong, the critical path of my morning routine would include both waiting for the bus to come and waiting for the traffic light to turn green. Both events would factor into the maximum amount of time I would have to schedule. However, if I don't have to be on campus today, I might simply decide that if I leave the house and the bus isn't there, I will give up and not go to campus that day. This condition is indicated with a

[2]This DAG incorrectly assumes that the bus won't leave without me.

weak edge from "bus comes" to "get on bus". If I wish to calculate the maximum (or average, etc.) time my morning routine will take *on days I go to campus*, I need not consider the time I might have to spend waiting for the bus, because if I have to wait for the bus, the morning is inadmissible and not counted toward "days I go to campus".

Well-formed DAGs To prove a bound on response time that depends only on work at low priority, we need to place an additional restriction on DAGs. Consider a DAG with two threads, $a \underset{\rho_a}{\hookrightarrow} (\delta_a, u_1 \cdot \ldots \cdot u \cdot \ldots \cdot u_n)$ and $b \underset{\rho_b}{\hookrightarrow} (\delta_b, \vec{u}_b)$, where $\rho_b \prec \rho_a$. Suppose there is a join edge (b, u) from b to a. Then thread b is counted toward the a-span of the DAG (put another way, thread a will need to wait for b to complete), so the response time of a depends on the length of thread b. It is not possible to construct a schedule of such a DAG in which the response time of thread a depends only on lower-priority work, and so we rule out such DAGs in our theorems.

Definition 3. A DAG $g = (\mathcal{T}, E_s, E_j, E_w)$ is *well-formed* if for all threads

$$a \underset{\rho}{\hookrightarrow} (\delta, u_1 \cdot_{\delta_1} \ldots \cdot_{\delta_{n-1}} u_n) \in \mathcal{T}$$

if $u \sqsupseteq^s u_n$ and $u \not\sqsupseteq u_1$ then $\rho \preceq Prio_g(u)$.

That is, no strong ancestor of u_n that isn't also an ancestor of u_1 may have a priority less than that of a. In more intuitive phrasing, this means that no thread depends on lower-priority work along its critical path. Threads may depend on lower-priority work through weak edges, but this will not cause any lack of responsiveness in admissible schedules because admissibility requires that weak edges do not delay the computation (i.e., do not end up being on the critical path).

3.3 Bounding response time of prompt schedules

Bounding the length of a schedule generally requires restricting attention to schedules that obey some *scheduling principle* (for an example of why this is necessary, the schedule that never executes any vertices is valid according to our definition but not useful for proving any bounds). Possibly the best-known of these is the *greedy* scheduling principle. A greedy schedule is one in which as many vertices as possible are executed in each time step, bounded by P and the number of ready vertices. Greedy schedules obey provable bounds on computation time [42], but greediness is insufficient to place bounds on response time. To provide such bounds, a schedule must take into account the thread priorities. Our prior work [87] develops the idea of a *prompt* schedule, which is a greedy schedule that prioritizes vertices according to their priority, with high priorities preferred over low. This prior work only considered languages with two priorities, so more care is required to apply the bounds to an arbitrary partial order. At each step, we assign at most P vertices to processors and then execute all of the assigned vertices in parallel. To begin, assign any ready vertex such that no unassigned ready vertex has a higher priority,[3] and continue until P vertices are assigned or no ready vertices remain. According to this definition, a prompt schedule is necessarily greedy.

[3]Simply saying "pick a vertex of the highest available priority" would be correct in a totally ordered setting, but might be ambiguous in our partially ordered setting.

Theorem 1 bounds the response time of a thread based on quantities which depend only on the work and span of high-priority threads.

Theorem 1. *Let $g \ni a \underset{\rho}{\hookrightarrow} (\delta, \vec{u})$ be a well-formed DAG. For any admissible prompt schedule on P processors,*

$$T(a) \leq \frac{W_{\not\prec\rho}(\nleftrightarrow a)}{P} + S_a(\nleftrightarrow a)$$

Proof intuition. This proof and several others in this chapter use a "bucket-and-token" analogy, which is fairly common in proofs of this form (e.g., [6]), to account for the actions of processors at each time step. We visualize a processor at a time step by a token that each processor spends at each time step. We divide processor-timesteps, and therefore tokens, into two categories or "buckets", depending on whether or not the processor is busy with high-priority work at that time step. We can bound the number of tokens in each category individually, and then add the quantities together to find the total number of tokens spent. Because P tokens are spent per time step, this in turn gives the number of time steps.

Proof. Let s and t be the first and last vertices of a, respectively. Consider the portion of the schedule from the step in which s becomes ready (exclusive) to the step in which t is executed (inclusive). For each processor at each step, place a token in one of two buckets. If the processor is working on a vertex of a priority not less than ρ, place a token in the "high" bucket; otherwise, place a token in the "low" bucket. Because P tokens are placed per step, we have $T(a) = \frac{1}{P}(B_l + B_h)$, where B_l and B_h are the number of tokens in the low and high buckets, respectively, after t is executed.

Each token in B_h corresponds to work done at priority not less than ρ, and thus $B_h \leq W_{\not\prec\rho}(\nleftrightarrow a)$, so

$$T(a) \leq \frac{W_{\not\prec\rho}(g)}{P} + \frac{B_l}{P}$$

We now need only bound B_l by $P \cdot S_a(\nleftrightarrow a)$.

Let step 0 be the first step after s is ready, and let $Exec(j)$ be the set of vertices that have been executed at the start of step j. Consider a step j in which a token is added to B_l. For any strong path ending at t consisting of vertices of $g \setminus Exec(j)$, the path starts at a vertex that is enabled at the beginning of step j. If the vertex is not ready, its delay decreases by one in step j. If the vertex is ready, then by the definition of well-formedness, it must have priority greater than ρ and is therefore executed in step j by the prompt principle. In either case, the length of the path decreases by 1 and so $S_a(g \setminus Exec(j+1)) = S_a(g \setminus Exec(j)) - 1$. If $S_a(g \setminus Exec(j)) = 1$ (i.e., t has no incoming strong edges), then t must be ready, otherwise this schedule is inadmissible. Therefore, the maximum number of steps in which B_l increases is $S_a(g \setminus Exec(0))$, and so $B_l \leq P \cdot S_a(g \setminus Exec(0))$. Because $\nleftrightarrow s \supset g \setminus Exec(0)$, any path excluding vertices in $Exec(0)$ is contained in $\nleftrightarrow s$, and $S_a(g \setminus Exec(0)) \leq S_a(\nleftrightarrow s)$, so $B_l \leq P \cdot S_a(\nleftrightarrow s) = P \cdot S_a(\nleftrightarrow a)$. □

For programs that start and end with a single "main" thread, the above theorem not only bounds response time, but computation time as well. Let a be the main thread, which is always at the bottommost priority. The response time of the main thread is equal to the computation time of the entire program. Because prompt schedules are greedy, we expect to be able to bound this time by $\frac{W}{P} + S$, where W is the total number of operations in the program and S is the length

of the longest path in the DAG [42]. Indeed, the priority work and $a-$span reduce to the overall work and span, respectively, so the bound given by Theorem 1 coincides with the expected bound on computation time.

3.4 Fairness

The prompt scheduling principle of the previous section, and the accompanying results, assume that higher-priority threads should always be scheduled preferentially over lower-priority ones. This is not always the desired behavior. In applications where higher-priority (e.g., interactive) threads constitute a large fraction of the work, prompt scheduling could cause lower-priority computation to be starved. Such applications benefit from a schedule that observes some notion of fairness, devoting a certain fraction of processor cycles to lower-priority work.

To avoid starvation, we introduce a fair scheduling principle that takes a parameter we call the *fairness criterion*. A fairness criterion C is a discrete probability distribution over priorities: a mapping from priorities to real numbers in the range $[0, 1]$, summing to 1. When threads of *all priorities* are present in the system, the scheduler should devote, on average, the fraction $C(\rho)$ of processor cycles to threads at priority ρ. When a particular priority is unavailable (i.e., has no threads available in the system), it "donates" its cycles to the highest available priority. This policy is flexible enough to encode many application-specific scheduling policies. For example, if $\top$ is the highest priority, we can encode a form of prompt scheduling in which as many processors as possible are devoted to the highest-priority work available, followed by the next-highest, and so on. The fairness criterion for such a policy would be $C(\top) = 1$ and $C(\rho) = 0$ for all $\rho \neq \top$.

A *fairly prompt schedule*, parametrized by a fairness criterion C, is a schedule that adheres to the principle described intuitively above. At each time step, processors are assigned to priorities probabilistically according to the distribution C. Processors attempt to execute a ready vertex at their assigned priority. Processors that are unable to execute a vertex at their assigned priority default to the "prompt" policy and execute the highest-priority ready vertex.

The use of probability in the definitions of fairness and fairly prompt schedules merits further discussion. Formal definitions of fairness, for example in the model checking community (e.g., [53]), are often stated as properties of infinite executions (e.g., that every process executes infinitely many times over an infinite interval). The notion of fairness described here diverges from such definitions in two important ways. First, the number of processes and (in general) the lengths of schedules are finite. Second, our notion of fairness places requirements on the relative frequency at which threads execute, as opposed to simply requiring that they execute eventually. Consider the following fairness criterion over a set of priorities $R = \{H, M, L\}$.

$$C(\rho) = \begin{cases} 0.5 & \rho = H \\ 0.3 & \rho = M \\ 0.2 & \rho = L \end{cases}$$

For a program that runs for 12 time steps on 2 processors, it is impossible to exactly meet such a fairness criterion because 0.2×24 is not an integral number of processor-steps. Given a finite, discrete number of processors and time steps, we could guarantee that the fairness criterion is met *to within some approximation*, but we instead choose to introduce probabilistic reasoning and to

guarantee that the fairness criterion is met exactly *in expectation*. As the number of processors or time steps approaches ∞, the distribution of work should converge to the fairness criterion.

In the pebbling analogy, executing a program using a fairly prompt schedule may be seen as pebbling a graph by drawing up to P pebbles at a time at random from an infinite bag of pebbles. Each pebble is colored, and each color is associated with a priority. The colored pebbles in the bag are distributed according to the fairness criterion C. When pebbles are drawn at a time step, we attempt to place each one on a vertex of the appropriate priority. Any pebbles that are left are placed on the highest-priority vertices, then the next-highest, and so on.

3.4.1 Bounding Response Time

We now bound the response time of threads in fairly prompt schedules by extending the results for prompt schedules in the previous section. We need not develop new cost metrics because, intuitively, fairness doesn't cause high-priority threads to depend on any work they otherwise wouldn't. Fairness only "inflates" the response time bounds to account for the fraction of cycles devoted to lower-priority work according to the fairness criterion. The priority work and a-span are defined as before. We also introduce a convenient shorthand for summing the fairness criterion over a set of priorities:

$$C(R') \triangleq \sum_{\rho \in R'} C(\rho)$$

Theorem 2 bounds the expected response time (because fairly prompt schedules are defined probabilistically, the response time can only be bounded in expectation) of a thread based on quantities which depend only on the fairness criterion and the work and span of high-priority vertices. Before stating the theorem, we first consider intuitively what the bound should be in several special cases. The theorem will give a general bound that specializes correctly in each of these cases.

1. **Prompt scheduling.** As stated above, a prompt schedule may be defined as a fairly prompt schedule for the fairness criterion C_p where $C_p(\top) = 1$ and $C_p(\rho) = 0$ if $\rho \neq \top$. So, for any schedule using C_p, the bound should match that of Theorem 1:

   $$E[T(a)] \leq \frac{W_{\not\prec \rho}(\nleftrightarrow a)}{P} + S_a(\nleftrightarrow a)$$

   (We express the bound as an expectation for consistency although this bound is actually exact because the fairness criterion is degenerate.)

2. **A thread of priority** ρ. Suppose the DAG has a thread a of priority ρ. Consider the processor cycles which are intended to be devoted to priorities not less than ρ. These cycles, in expectation, make up a fraction $C(\not\prec \rho)$. For each of these processor cycles, the schedule is greedy with respect to ready work at priority not less than ρ: by the fairly prompt principle, if such work exists, it will be executed on this cycle. (If a cycle is intended for priority ρ and no work at ρ is ready on this cycle, but a vertex of priority ρ' greater than ρ is ready, then the cycle will be donated to that vertex rather than being used on lower-priority work.) So, we would expect the response time bound to match that of

Theorem 1, inflated to account for the fact that only $\frac{1}{C(\not\prec\rho)}$ of cycles are being spent on this work in expectation.

$$E[T(a)] \leq \frac{1}{C(\not\prec \rho)} \left(\frac{W_{\not\prec\rho}(\ddagger a)}{P} + S_a(\ddagger a) \right)$$

3. **A bottom-heavy fairness criterion.** Consider a fairness criterion C_b where $C_b(\bot) = 1$ and $C_b(\rho) = 0$ for $\rho \neq \bot$. Since such a schedule is, in some sense, the opposite of prompt, one might expect it to not be useful. However, some of our experimental results have shown that, when high-priority interactive work is very rare, it can be beneficial to assign very few cycles to it and devote most cycles to computation. The criterion C_b is the extreme limit of this idea. If C_b is plugged in to the bound of the previous case, the bound diverges for any $\rho \neq \bot$. In this case, a more useful analysis is to simply observe that we are always being greedy with respect to the work available at any priority. This would give us the bound

$$E[T(a)] \leq \frac{W_R(\ddagger a)}{P} + S_a(\ddagger a)$$

Cases 2 and 3 above can be viewed as two extremes along a spectrum. In both cases, some set R' of priorities is under consideration, along with the cycles devoted to priorities in R' and the competitor work at priorities in R'. In case 2, we let $R' = \not\prec \rho$, thus focusing on a smaller fraction of cycles but also a smaller amount of competitor work. In case 3, we let $R' = R$, focusing on all of the cycles but a larger amount of competitor work. The general bound interpolates between the two, allowing R' to be $\not\prec \rho'$ for any $\bot \preceq \rho' \preceq \rho$. Case 2 is obtained by letting $\rho' = \rho$ and case 3 is obtained by letting $\rho' = \bot$. Case 1 is obtained by letting $\rho' = \rho$ and $C = C_p$.

Theorem 2. *Let $g \ni a \underset{\rho}{\hookrightarrow} (\delta, \vec{u})$ be a well-formed DAG. For any fairly prompt schedule on P processors and any $\rho' \preceq \rho$,*

$$E[T(a)] \leq \frac{1}{C(\not\prec \rho')} \left(\frac{W_{\not\prec\rho'}(\ddagger a)}{P} + S_a(\ddagger a) \right)$$

Proof. Let s and t be the first and last vertices of a, respectively. Consider the portion of the schedule from the step in which s becomes ready (exclusive) to the step in which t is executed (inclusive). At each step, let P_C be the number of processors attempting to work on vertices of priorities in $\not\prec \rho'$. For each processor at each step, place a token in one of three buckets. If the processor is attempting to work at a priority not less than ρ', but is unable to, place a token in the "low" bucket. If it attempting to work at a priority not less than ρ' and succeeds, place a token in the "high" bucket. If it is attempting to work at a priority less than ρ', place a token in the "fair" bucket B_f. Because P tokens are placed per step, we have $T(a) = \frac{1}{P}(B_l + B_h + B_f)$, where B_l, B_h and B_f are the number of tokens in the low, high and fair buckets, respectively, after t is executed.

Let $c = C(\not\prec \rho')$. By fairness, we have $E[B_l + B_h] = c(B_l + B_h + B_f)$. Thus,

$$E[T(a)] = \frac{1}{Pc}(B_l + B_h)$$

Each token in B_h corresponds to work done at priority not less than ρ', and thus $B_h \leq W_{\not\prec\rho'}(\nleftrightarrow a)$, so

$$T(a) \leq \frac{1}{c}\left(\frac{W_{\not\prec\rho'}(g)}{P} + \frac{B_l}{P}\right)$$

We now need only bound B_l by $P \cdot S_a(\nleftrightarrow a)$. The proof of this bound is identical to the analogous portion of the proof of Theorem 1. □

Theorem 2 adapts the bound of Theorem 1 to fairly prompt schedules. Both theorems are primarily concerned with the responsiveness of high-priority threads, since Theorem 1 was formulated in a setting where starvation is possible and no guarantees can be made about the performance of low-priority threads. Theorem 2 bounds how much high-priority threads can be penalized by fairness, but not how much low-priority threads can benefit from fairness. Since the entire point of introducing fairness was to guarantee decent performance for lower-priority threads, we give such a bound as well.

The intuition behind this result is that, from the perspective of a thread (or collection of threads) at one priority ρ, a fairly prompt schedule on P processors appears the same as a greedy schedule on $P \cdot C(\rho)$ processors. In this way, the result is very much like the offline scheduling bound of Arora et al. [6], which concerns *multiprogrammed environments* in which only a fraction of the total processors may be available to a parallel computation at any given step. Our bound must include the thread's competitor work at priority ρ, but otherwise the bound and proof are quite similar to those of Arora et al.

Theorem 3. *Let g be a well-formed DAG. Let $a \underset{\rho}{\hookrightarrow} (\delta, s \cdot \ldots \cdot t) \in g$ be such that for all $u \in g$ where $u \sqsupseteq t$ and $u \not\sqsupseteq s$, it is the case that $Prio_u(g) = \rho$, and that all edges incident on u are strong and light.*

For any fairly prompt schedule on P processors and any $\rho' \preceq \rho$,

$$E[T(a)] \leq \frac{1}{P \cdot C(\rho)}\left(W_{\{\rho\}}(\nleftrightarrow a) + S_a(\nleftrightarrow a) \cdot (P-1)\right)$$

Proof. Consider the portion of the schedule from the step in which s becomes ready (exclusive) to the step in which t is executed (inclusive). At each step, collect a token from each processor *assigned to priority* ρ at that step. If the processor successfully works on a vertex of priority ρ at that step, place a token in the "work" bucket; otherwise, place a token in the "idle" bucket. Suppose the number of tokens collected at a step i is P_i. Since $E[P_i] = P \cdot C(\rho)$, for any T, we have

$$E[\sum_{i=1}^{T} P_i] = TP \cdot C(\rho)$$

Thus,

$$E[T(a)] = \frac{1}{P \cdot C(\rho)}(B_w + B_l)$$

where B_w and B_l are the numbers of tokens in the work and idle buckets, respectively, at the end of the computation. Each token in B_w corresponds to work done at priority ρ, and thus $B_w = W_{\{\rho\}}(\nleftrightarrow a)$, so

$$E[T(a)] \leq \frac{1}{P \cdot C(\rho)}\left(W_{\{\rho\}}(\nleftrightarrow a) + +B_l\right)$$

We now need only bound B_l by $S_a(\nleftrightarrow a) \cdot (P-1)$.

Let step 0 be the step after s is ready, and let $Exec(j)$ be the set of vertices that have been executed at the start of step j. Consider a step j in which a token is added to B_l. For any strong path ending at t consisting of vertices of $g \setminus Exec(j)$, the path starts at a vertex of priority ρ that is ready at the beginning of step j (it must be ready since, by assumption, all edges are light). By the fair principle, this vertex must be executed in step j, so the length of the path decreases by 1 and so $S_a(g \setminus Exec(j+1)) = S_a(g \setminus Exec(j)) - 1$. Therefore, the maximum number of steps in which B_l increases is $S_a(g \setminus Exec(0))$, and so $B_l \leq S_a(g \setminus Exec(0)) \cdot (P-1)$ since at most $P-1$ processors can be idle while thread a is active. Because $\nmid s \supset g \setminus Exec(0)$, any path excluding vertices in $Exec(0)$ is contained in $\nmid s$, and $S_a(g \setminus Exec(0)) \leq S_a(\nmid s)$, so $B_l \leq S_a(\nmid s) = P \cdot S_a(\nleftrightarrow a) \cdot (P-1)$. □

Chapter 4

A Language for Responsive Parallel Programs

> They have been at a great feast
> of languages, and stolen the scraps.
>
> *Love's Labour's Lost* (V.1.38–39)

This chapter introduces PriML, a language for writing responsive parallel programs, which includes a high-level thread model inspired by cooperative languages, facilities for specifying the priorities of threads using partially ordered priorities, and a type system for preventing priority inversions. In Section 4.1, we present the language at a high level. This section is not intended to be an exhaustive or formal language definition, but rather to highlight the main ideas of the language and how it accomplishes the goals of this thesis. The presentation is therefore high-level and sometimes informal. The rest of this chapter formalizes these ideas by presenting a core calculus λ^4 (Section 4.2) that captures the essence of PriML, and showing how programs written in a sizable, formally defined subset of PriML may be elaborated into programs in the calculus (Section 4.3). The remainder of the thesis expands on the ideas of this chapter to place performance bounds on λ^4 programs (Chapter 5) and describes how they may be realized in practice (Chapters 6 and7).

4.1 The PriML language

This section presents an overview of an ML-like language for multithreaded programming with priorities. As a running example, we consider an email client which interacts with a user while performing other necessary tasks in the background. The PriML language is built on top of Standard ML, and incorporates most of the features of SML, in addition to facilities for creating and manipulating threads, and for defining priorities. The language also has a type system that extends SML's type system to ensure the proper use of priorities. We will not describe the existing SML language in detail in this thesis.

Priorities. PriML enables the programmer to define priorities as needed and specify the relationships between them. For example, in designing an email client, we may wish to alert the user to certain situations (such as an incoming email) and we also wish to compress old emails in the background when the system is idle. To express this in PriML, we define two priorities `alert` and `background` and order them accordingly as follows.

```
priority alert
priority background
order background < alert
```

The ordering constraint specifies that `background` is lower priority than `alert`. Programmers are free to specify as many, or as few, ordering constraints between priorities as desired. PriML therefore provides support for a set of partially ordered priorities. Partially ordered priorities suffice to capture the intuitive notion of priorities, and to give the programmer flexibility to express any desired priority behavior, but without the burden of having to reason about a total order over all priorities. Consider two priorities `p` and `q`. If they are ordered, e.g., `p < q`, then the system is instructed to run threads with priority `q` over threads with priority `p`[1]. If no ordering is specified (i.e., `p` and `q` are incomparable in the partial order), then the system is free to choose arbitrarily between a thread with priority `p` and another with priority `q`.

The structure of a program. To ensure responsive use of priorities, PriML provides a modal type system that tracks priorities. To support the tracking of priorities, the syntax and type system of PriML distinguish between commands and expressions.

Commands provide the constructs for spawning and synchronizing threads. Because commands may interact with other threads, the priority at which a command runs is significant and must be specified (we will later introduce a way to introduce commands that are polymorphic in the priority at which they run). Commands return values, and are considered to have the type of the value they return. For example, the command `ret e` evaluates `e` to a value and returns the value. Thus,

```
ret (20 * 2 + 2)
```

is a command of type `int`.

Expressions consist of an ML-style functional language, with some extensions. Expressions cannot directly execute commands or interact with threads, and can thus be evaluated without regard to priority. Expressions can, however, pass around encapsulated commands. The syntax

```
cmd[p] { m }
```

encapsulates the command `m`, running at priority `p` (where `m` is a command of type `'a`). The expression above is of type `'a cmd[p]`. The encapsulated command is suspended and does not run immediately (because it may spawn or interact with other threads, which expressions are

[1]As the syntax implies, the above ordering gives `p` a *strictly lower* priority than `q`. Thus, it would be more precise to say that the programmer annotations define a *strict* order on priorities. However, for the purposes of type checking, it will be more useful to consider the partial order <=, where `p <= q` if `p` and `q` are syntactically equal or if `p < q` can be derived from the transitive closure of the programmer annotations. Because the partial order can be easily derived from the strict order and vice versa, we use the term "partial order" for consistency with prior literature on priorities.

not permitted to do), but it may be executed by the command `do e`, which evaluates `e` to an encapsulated command and then executes it. As an example, the code

```
do
{
  let val m = cmd[p] { ret (20 * 2 + 2) }
      val _ = do_a_lot_of_stuff ()
  in
    m
  end
}
```

will execute the command `ret (20 * 2 + 2)`, thus returning 42, but only after evaluating `do_a_lot_of_stuff ()`.

Commands may be sequenced using the notation `{ x <- m1; m2}`, which runs `m1`, binds its return value to `x` and then runs `m2`. If the return value of `m1` is not needed, the shorthand `{ m1; m2}` is available. A full PriML program consists of a sequence of declarations, followed by a single command which is run as the "main" thread of the program.

Threads. Commands may spawn new threads which execute asynchronously with the rest of the program. Threads are annotated with the priorities at which they run. For example, in response to a request from the user, the mail client can spawn a thread to sort emails for background compression, and spawn another thread to alert the user about an incoming email.

```
spawn[background] { ret (sort ...) };
spawn[alert] { ret (display ``Incoming mail!'') }
```

The `spawn` command takes a command to run in the new thread and returns a handle to the spawned thread. Thread handles to threads at priority `p` returning values of type `t` are first-class values of type `t thread[p]`. In the above code, the returned thread handle is ignored, but it can also be bound to a variable and used later to synchronize with the thread (wait for it to complete).

```
spawn[background] { ret (sort ...) };
alert_thread <- spawn[alert] { ret (display ``New mail received'') };
sync alert_thread
```

Priority polymorphism. Previously, we have seen that commands must be annotated with the priority at which they run. This can be quite restrictive in practice because it is often the case that code is reused in many places throughout a program, possibly at different priorities. For example, several parts of our email client might wish to use a library function `sort` for sorting (e.g., the background thread sorts emails by date to decide which ones to compress and a higher-priority thread sorts emails by subject when the user clicks a column header.) To support computations that can operate at multiple priorities, the type system supports *priority polymorphism* through a polymorphic type of the form `forall p: C. t`, where `p` is a newly bound priority variable, and `C` is a set of constraints of the form `p1 <= p2` (where `p1` and `p2` are priority constants or variables, one of which will in general be `p`), which bounds the allowable instantiations of `p`.

```
fun[p] qsort (compare: 'a * 'a -> bool) (s: 'a seq) : 'a seq cmd[p] =
  if Seq.isEmpty s then
    cmd[p] {ret Seq.empty}
  else
    let val pivot = Seq.sub(s, (Seq.length s) / 2)
        val (s_l, s_e, s_g) = Seq.partition (compare pivot) s
    in
      cmd[p]
      {
        quicksort_l <- spawn[p] {do ([p]qsort compare s_l)};
        quicksort_g <- spawn[p] {do ([p]qsort compare s_g)};
        ss_l <- sync quicksort_l;
        ss_g <- sync quicksort_g;
        ret (Seq.append [ss_l, s_e, ss_g])
      }
    end
```

Figure 4.1: Code for multithreaded quicksort, which is priority polymorphic.

Example: priority-polymorphic multithreaded quicksort. We can implement a parallel sorting routine, using a parallel version of the Quicksort algorithm. Quicksort is easily parallelized, and so the library code spawns threads to perform recursive calls in parallel. The use of threads, however, means that the code must involve priorities and cannot be purely an expression. Because sorting is a basic function and may be used at many priorities, we would want the code for `qsort` to be polymorphic over priorities. This is possible in PriML by defining `qsort` to operate at a priority defined by an unrestricted priority variable.

Figure 4.1 illustrates the code for a multithreaded implementation of Quicksort in PriML. The code operates over sequences, an abstract data type of parallel ordered collections, and uses a module called `Seq` which implements some basic operations on sequences. In addition to a comparison function on the elements of the sequence that will be sorted and the sequence to sort, `qsort` takes as an argument a priority `p`, to which the body of the function may refer (e.g., to spawn threads at that priority)[2]. The implementation of `qsort` follows a standard implementation of the algorithm but is structured according to the type system of PriML. This can be seen in the return type of the function, which is an encapsulated command at priority `p`.

The function starts by checking if the sequence is empty. If so, it returns a command that returns an empty sequence. If the sequence is not empty, it partitions the sequence into subsequences consisting of elements less than, equal to and greater than, a pivot, chosen to be the middle element of the sequence. It then returns a command that sorts the sub-sequences in parallel, and concatenates the sorted sequences to produce the result. To perform the two recursive calls in parallel, the function `spawn`s two threads, specifying that the threads operate

[2]Note that, unlike type-level parametric polymorphism in languages such as ML, which can be left implicit and inferred during type checking, priority parameters in PriML must be specified in the function declaration.

```
priority loop_p
priority sort_p
order sort_p < loop_p

fun loop (emails: email list) (bgt: email list thread[sort_p])
  : unit cmd[loop_p] =
  cmd[loop_p]
  {
    r <- poll bgt;
    x <- ret (case r of
                SOME l => display_ordered l
              | NONE => ());
    (* Handle events... *)
    do (loop emails bgt)
  }

fun startloop (emails: email list) : unit cmd[loop_p] =
  cmd[loop_p]
  {
    t <- spawn[sort_p] {do ([sort_p]qsort date emails)};
    do (loop emails t)
  }
```

Figure 4.2: Querying background threads with polling.

at priority `p`.

Polling. The `sync` command blocks on a thread until it is complete. Sometimes, we would rather check if a thread is complete in a non-blocking fashion. For this purpose, PriML also supports *polling* of threads. The command `poll e`, where `e : 'a thread[p]`, evaluates `e` to a thread handle and returns a value of type `'a option`. If the thread has completed and returned a value `v`, the command returns `SOME v`. Otherwise, it returns `NONE`[3]

Polling is useful for checking the status of a background computation without blocking a foreground thread. For example, the email client might spawn a background sorting thread from a high-priority event loop and check its status at each iteration of the loop in order to display the sorted emails when complete. Code for this example is shown in Figure 4.2.

Cancellation. By default, a thread runs to completion even if it is never synchronized on or polled. This is necessary to preserve the semantics of a program because, in general, threads

[3]We do not specify that polling must return completely up-to-date results, because this may be difficult or costly to implement in practice on very large systems, depending on memory models and implementation details. Our implementation, however, makes thread results available for polling very quickly.

```
1  {
2    t1 <- spawn[p] {do ([sort_p]qsort date emails)};
3    t2 <- spawn[p] {do ([sort_p]mergesort date emails)};
4    do (let fun choose () =
5                  cmd[p]
6                  {
7                    v1 <- poll t1;
8                    v2 <- poll t2;
9                    do (case (v1, v2) of
10                         (SOME l, _) => cmd[p] { cancel t2; ret l }
11                       | (_, SOME l) => cmd[p] { cancel t1; ret l }
12                       | (NONE, NONE) => choose ())
13                 }
14       in
15         choose ()
16       end)
17 }
```

Figure 4.3: Choosing the first thread to complete with polling and cancellation.

may have side effects. We provide a `cancel` command to explicitly stop execution of running threads. Cancelling a thread indicates to the runtime that the thread is no longer needed and can be discarded. The runtime may delay cancellation of the thread until it is safe and/or convenient. Programmers should therefore not depend on cancellation happening immediately.

Combined with polling, cancellation allows us to spawn two threads, perhaps performing the same computation by different methods, use the answer of the first thread to complete, and cancel the other to avoid unnecessary computation. This is similar to Manticore's nondeterministic choice [49] or CML's select, though it is not the most efficient implementation of such a construct because of the busy-waiting. Code for this example is shown in Figure 4.3. This example shows that, even in the absence of mutable state, the presence of polling makes PriML programs nondeterministic. Cancellation also leads to nondeterminism in the presence of any side effects, because there is no guarantee of which side effects will occur before a thread is cancelled.

Priority Inversions. The purpose of the modal type system is to prevent priority inversions, that is, situations in which a thread synchronizes with a thread of a lower priority. As we saw in Chapter 3, such inversions prevent us from showing reasonable bounds on the response times of high-priority threads, and so we ruled them out by considering only *well-formed* DAGs without priority inversions. The requirements placed on programs by the type system are analogous, and indeed we show in Section 5.2 that well-typed programs give rise to well-formed cost DAGs in a formal sense. An example of a priority inversion appears in Figure 4.4a. This code shows a portion of the main event loop of the email client, which processes and responds to input from the user. The event loop runs at a high priority. If the user sorts the emails by date, the loop spawns a new thread, which calls the priority-polymorphic sorting function. The code

```
1  priority loop_p
2  priority sort_p
3  order sort_p < loop_p
4
5  fun loop emails : unit cmd[loop_p]
6    =
7    case next_event () of
8    SORT_BY_DATE =>
9      cmd[loop_p]
10     {
11       t <- spawn[sort_p]
12            {do ([sort_p]qsort
13                   date emails)};
14       l <- sync t;
15       ret (display_ordered l)
16     }
17     | ...
```

(a) Ill-typed event loop code

```
1  priority loop_p
2  priority sort_p
3  order sort_p < loop_p
4
5  fun loop emails : unit cmd[loop_p]
6    =
7    case next_event () of
8    SORT_BY_DATE =>
9      cmd[loop_p]
10     {
11       spawn[sort_p]
12       { l <- do ([sort_p]qsort
13                    date emails);
14         ret (display_ordered l)
15       }
16     }
17     | ...
```

(b) Well-typed event loop code

Figure 4.4: Two implementations of the event loop, one of which displays a priority inversion.

instantiates this function at a lower priority `sort_p`, reflecting the programmer's intention that the sorting, which might take a significant fraction of a second for a large number of emails, should not delay the handling of new events. Because syncing with that thread immediately afterward (line 14) causes the remainder of the event loop (high-priority) to wait on the sorting thread (lower priority), this code is correctly rejected by the type system. The programmer could instead write the code as shown in Figure 4.4b, which displays the sorted list in the new thread, allowing the event loop to continue processing events. This code does not have a priority inversion and is accepted by the type system.

Although the priority inversion of Figure 4.4a could easily be noticed by a programmer, the type system also rules out more subtle priority inversions. Consider the ill-typed code in Figure 4.5, which shows another way in which a programmer might choose to implement the event loop. In this implementation, the event loop spawns two threads. The first (at priority `sort_p`) sorts the emails, and the second (at priority `display_p`) calls a priority-polymorphic function `[p]disp`, which takes a sorting thread at priority `p`, waits for it to complete, and displays the result. This type of "chaining" is a common idiom in programming with futures, but this attempt has gone awry because the thread at priority `display_p` is waiting on the lower-priority sorting thread. Because of priority polymorphism, it may not be immediately clear where exactly the priority inversion occurs, and yet this code will still be correctly rejected by the type system. The type error is on line 10. This `sync` operation is passed a thread of priority `p`, and there is no guarantee that `p` is higher-priority than `display_p` (and, in fact, the instantiation on line 20 would violate this constraint). We may correct the type error in the `disp` function by adding this

```
1  priority loop_p
2  priority display_p
3  priority sort_p
4  order sort_p < loop_p
5  order sort_p < display_p
6
7  fun[p] disp (t : email seq thread[p]) : unit cmd[display_p] =
8    cmd[display_p]
9    {
10     l <- sync t;
11     ret (display_ordered l)
12   }
13
14 fun loop emails : unit cmd[loop_p] =
15   case next_event () of
16   SORT_BY_DATE =>
17     cmd[loop_p]
18     {
19       t <- spawn[sort_p] { do ([sort_p]qsort date emails) };
20       spawn[display_p] { do ([sort_p]disp t) }
21     }
22     | ...
```

Figure 4.5: An ill-typed attempt at chaining threads together.

constraint to the signature:

```
fun[p : display_p <= p] disp (t: email seq thread[p])
  : unit cmd[display_p] =
```

With this change, the instantiation on line 20 would become ill-typed, as it should because this way of structuring the code inherently has a priority inversion. The event loop code should be written as in Figure 4.4b to avoid a priority inversion. However, the revised `disp` function could still be called on a higher-priority thread (e.g., one that checks for new mail).

Note that the programmer could also fix the type error in both versions of the code by spawning the sorting thread at `loop_p`. This change, however, betrays the programmer's intention (clearly stated in the priority annotations) that the sorting should be lower priority. The purpose of the type system, as with all such programming language mechanisms, is not to relieve programmers entirely of the burden of thinking about the desired behavior of their code, but rather to ensure that the code adheres to this behavior if it is properly specified.

Fairness. PriML allows programmers to specify fairness criteria, as described in Section 3.4, by assigning integers to priorities indicating how processor resources should be allocated in situations where work at all priorities is available. After declaring a priority `p`, the fairness value

can be indicated using a new form of declaration:

```
fairness p 100
```

The scaling on the numbers is arbitrary and is normalized at compile time. The relative amount of processor time given to each priority is determined by the ratio of the fairness values. The fairness criterion is orthogonal to the ordering on the priorities. For example, if it is very important to us that the sorting thread not be starved by the interaction loop, we might assign it most of the processor time while still indicating that the interaction loop is higher priority:

```
1 priority loop_p
2 priority display_p
3 priority sort_p
4 order sort_p < loop_p
5 order sort_p < display_p
6 fairness sort_p 50
7 fairness display_p 30
8 fairness loop_p 20
```

If a fairness value is not given for a particular priority, it defaults to zero. Work at that priority will still be done if work at higher priorities is unavailable. For example, the fairness declarations

```
fairness display_p 30
fairness loop_p 20
```

indicate that sorting should be done during any time in which display and interaction loop events are unavailable. If no fairness criterion is specified, the default is a "winner-take-all" strategy in which the highest-priority work available is always run.

4.2 A Core Calculus for Prioritized Threads

In this section, we define a core calculus λ^4 which captures the key ideas of the PriML language. Figure 4.6 presents the abstract syntax of λ^4. In addition to the unit type, a type of natural numbers, functions, product types and sum types, λ^4 has three special types. The type $\tau\ \texttt{thread}[\rho]$ is used for a handle to an asynchronous thread running at priority ρ and returning a value of type τ. The type $\tau\ \texttt{cmd}[\rho]$ is used for an encapsulated command. The calculus also has a type $\forall\pi : C.\tau$ of priority-polymorphic expressions. These types are annotated with a constraint C which restricts the instantiation of the bound priority variable. For example, the abstraction $\Lambda\pi : \pi \preceq \overline{\rho}.e$ can only be instantiated with priorities $\overline{\rho}'$ for which $\overline{\rho}' \preceq \overline{\rho}$.

A priority ρ can be either a priority constant, written $\overline{\rho}$, or a priority variable π. Priority constants are drawn from a pre-defined set, in much the same way that numerals $\overline{n}$ are drawn from the set of natural numbers. The set of priority constants (and the partial order over them) will be determined statically and is a parameter to the static and dynamic semantics. This is a key difference between the calculus λ^4 and PriML, in which the program can define new priority constants. We discuss in Section 4.3 how the priority definitions of PriML may be hoisted out of the program to produce λ^4 programs.

Types	τ	$::=$	$\texttt{unit}$	Unit
		$\mid$	$\texttt{nat}$	Natural number
		$\mid$	$\tau \to \tau$	Function
		$\mid$	$\tau \times \tau$	Product
		$\mid$	$\tau + \tau$	Sum
		$\mid$	$\tau\,\texttt{thread}[\rho]$	Thread
		$\mid$	$\tau\,\texttt{cmd}[\rho]$	Encapsulated command
		$\mid$	$\forall \pi : C.\tau$	Priority polymorphism
Priorities	ρ	$::=$	$\overline{\rho}$	Priority constant
		$\mid$	π	Priority variable
Constraints	C	$::=$	$\rho \preceq \rho$	Less-or-equal
		$\mid$	$C \wedge C$	Conjunction
Values	v	$::=$	x	Variable
		$\mid$	$\langle\rangle$	Unit
		$\mid$	$\overline{n}$	Numeral
		$\mid$	$\lambda x.e$	Function abstraction
		$\mid$	$\langle v, v\rangle$	Tuple value
		$\mid$	$\texttt{l} \cdot v$	Left-tagged value
		$\mid$	$\texttt{r} \cdot v$	Right-tagged value
		$\mid$	$\texttt{tid}[a]$	Thread handle
		$\mid$	$\texttt{cmd}[\rho]\ \{m\}$	Encapsulated command
		$\mid$	$\Lambda \pi : C.e$	Priority abstraction
Expressions	e	$::=$	v	Value
		$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	Let binding
		$\mid$	$\texttt{ifz}\ v\ \{e; x.e\}$	Zero test
		$\mid$	$v\ v$	Application
		$\mid$	(v, v)	Pair creation
		$\mid$	$\texttt{fst}\ v$	Left projection
		$\mid$	$\texttt{snd}\ v$	Right projection
		$\mid$	$\texttt{inl}\ v$	Left injection
		$\mid$	$\texttt{inr}\ v$	Right injection
		$\mid$	$\texttt{case}\ v\ \{x.e; y.e\}$	Case analysis
		$\mid$	$\texttt{output}\ v$	(Natural number) output
		$\mid$	$\texttt{input}_d$	(Natural number) input
		$\mid$	$v[\rho]$	Priority application
		$\mid$	$\texttt{fix}\ x{:}\tau\ \texttt{is}\ e$	Fixed point
Commands	m	$::=$	$x \leftarrow e; m$	Monadic bind
		$\mid$	$\texttt{spawn}[\rho; \tau]\ \{m\}$	Thread spawn
		$\mid$	$\texttt{sync}\ e$	Thread join
		$\mid$	$\texttt{poll}\ e$	Poll
		$\mid$	$\texttt{ret}\ e$	Return

Figure 4.6: Syntax of λ^4

As in PriML, the syntax is separated into expressions, which do not involve priorities, and commands, which do. For simplicity, the expression language is in "2/3-cps" form: we distinguish between expressions and values, and expressions take only values as arguments when this would not interfere with evaluation order. An expression with unevaluated subexpressions such as (e_1, e_2) can be expressed using let bindings as $\texttt{let}\ x = e_1\ \texttt{in}\ \texttt{let}\ y = e_2\ \texttt{in}\ (x, y)$. Values consist of the unit value $\langle\rangle$, numerals $\overline{n}$, anonymous functions $\lambda x.e$, pairs of values, left- and right-injection of values, thread identifiers, encapsulated commands $\texttt{cmd}[\rho]\ \{m\}$ and priority-level abstractions $\Lambda\pi : C.e$.

Expressions include values, let binding, the if-zero conditional $\texttt{ifz}\ e\ \{e_1; x.e_2\}$ and function application. There are also additional expression forms for pair introduction and left- and right-injection. These are (v_1, v_2), $\texttt{inl}\ v$ and $\texttt{inr}\ v$, respectively. One may think of these forms as the source-level instructions to allocate the pair or tag, and the corresponding value forms as the actual runtime representation of the pair or tagged value. Making this distinction in the syntax allows us to account for the cost of performing the allocation. Finally, expressions include the case construct $\texttt{case}\ e\ \{x.e_1; y.e_2\}$, output, input, priority instantiation $e[\rho]$ and fixed points. The expression $\texttt{output}\ v$ models an output operation providing information to the user. The expression $\texttt{input}_d$ models receiving information from the user. The exact form of the output and input (e.g., console interaction, network interaction, arbitrary system calls) are left abstract. We only provide output and input operations for natural numbers, since these are the only nontrivial base type in λ^4 . Adding more base types, and extending the output and input constructs, would be straightforward. Input statements are annotated with unique identifiers d. These identifiers will be used in the cost semantics to associate each input operation with the latency it may incur. They may safely be ignored for now.

Commands are combined using the binding construct $x \leftarrow e; m$, which evaluates e to an encapsulated command, which it executes, binding its return value to x, before continuing with command m. There are also commands to spawn, synchronize with, and poll, other threads. We do not include cancellation in λ^4 because it is essentially orthogonal to thread priorities, the type system, and runtime cost, which are the three important features of the calculus[4]. The spawn command $\texttt{spawn}[\rho; \tau]\ \{m\}$ is parametrized by both a priority ρ and the type τ of the return value of m for convenience in defining the dynamic semantics.

4.2.1 Static Semantics

The type system of λ^4 carefully tracks the priorities of threads as they wait for each other and enforces that a program is free of priority inversions. This static guarantee ensures that we can derive cost guarantees from well-typed programs.

As with the syntax, the static semantics are separated into the expression layer and the command layer. Because expressions do not depend on priorities, the static semantics for expressions is fairly standard. The main unusual feature is that the typing judgment is parametrized by a signature Σ containing the types and priorities of running threads. A signature has entries of the

[4] It is quite counterintuitive that cancellation would be orthogonal to the cost semantics, because a main goal of cancellation is to reduce unnecessary work. However, in PriML, cancellation is not guaranteed to occur, and so in the worst-case cost analysis we perform later in the thesis, it is effectively a no-op.

$$\textsc{var}\ \frac{}{\Gamma, x:\tau \vdash^R_\Sigma x : \tau} \qquad \textsc{unit}\mathrm{I}\ \frac{}{\Gamma \vdash^R_\Sigma \langle\rangle : \mathtt{unit}} \qquad \textsc{Tid}\ \frac{}{\Gamma \vdash^R_{\Sigma, a\sim\tau@\rho'} \mathtt{tid}[a] : \tau\ \mathtt{thread}[\rho']}$$

$$\textsc{nat}\mathrm{I}\ \frac{}{\Gamma \vdash^R_\Sigma \overline{n} : \mathtt{nat}} \qquad \textsc{nat}\mathrm{E}\ \frac{\Gamma \vdash^R_\Sigma v : \mathtt{nat} \qquad \Gamma \vdash^R_\Sigma e_1 : \tau \qquad \Gamma, x:\mathtt{nat} \vdash^R_\Sigma e_2 : \tau}{\Gamma \vdash^R_\Sigma \mathtt{ifz}\ v\ \{e_1; x.e_2\} : \tau}$$

$$\to\mathrm{I}\ \frac{\Gamma, x:\tau_1 \vdash^R_\Sigma e : \tau_2}{\Gamma \vdash^R_\Sigma \lambda x.e : \tau_1 \to \tau_2} \qquad \to\mathrm{E}\ \frac{\Gamma \vdash^R_\Sigma v_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash^R_\Sigma v_2 : \tau_1}{\Gamma \vdash^R_\Sigma v_1\ v_2 : \tau_2}$$

$$\times I_1\ \frac{\Gamma \vdash^R_\Sigma v_1 : \tau_1 \qquad \Gamma \vdash^R_\Sigma v_2 : \tau_2}{\Gamma \vdash^R_\Sigma (v_1, v_2) : \tau_1 \times \tau_2} \qquad \times I_2\ \frac{\Gamma \vdash^R_\Sigma v_1 : \tau_1 \qquad \Gamma \vdash^R_\Sigma v_2 : \tau_2}{\Gamma \vdash^R_\Sigma \langle v_1, v_2\rangle : \tau_1 \times \tau_2}$$

$$\times E_1\ \frac{\Gamma \vdash^R_\Sigma v : \tau_1 \times \tau_2}{\Gamma \vdash^R_\Sigma \mathtt{fst}\ v : \tau_1} \qquad \times E_2\ \frac{\Gamma \vdash^R_\Sigma v : \tau_1 \times \tau_2}{\Gamma \vdash^R_\Sigma \mathtt{snd}\ v : \tau_2} \qquad +I_1\ \frac{\Gamma \vdash^R_\Sigma v : \tau_1}{\Gamma \vdash^R_\Sigma \mathtt{inl}\ v : \tau_1 + \tau_2}$$

$$+I_2\ \frac{\Gamma \vdash^R_\Sigma v : \tau_2}{\Gamma \vdash^R_\Sigma \mathtt{inr}\ v : \tau_1 + \tau_2} \qquad +I_3\ \frac{\Gamma \vdash^R_\Sigma v : \tau_1}{\Gamma \vdash^R_\Sigma \mathtt{l} \cdot v : \tau_1 + \tau_2} \qquad +I_4\ \frac{\Gamma \vdash^R_\Sigma v : \tau_2}{\Gamma \vdash^R_\Sigma \mathtt{r} \cdot v : \tau_1 + \tau_2}$$

$$+\mathrm{E}\ \frac{\Gamma \vdash^R_\Sigma v : \tau_1 + \tau_2 \qquad \Gamma, x:\tau_1 \vdash^R_\Sigma e_1 : \tau' \qquad \Gamma, y:\tau_2 \vdash^R_\Sigma e_2 : \tau'}{\Gamma \vdash^R_\Sigma \mathtt{case}\ v\ \{x.e_1; y.e_2\} : \tau'}$$

$$\textsc{Output}\ \frac{\Gamma \vdash^R_\Sigma v : \mathtt{nat}}{\Gamma \vdash^R_\Sigma \mathtt{output}\ v : \mathtt{unit}} \qquad \textsc{Input}\ \frac{}{\Gamma \vdash^R_\Sigma \mathtt{input}_d : \mathtt{nat}} \qquad \textsc{cmd}\mathrm{I}\ \frac{\Gamma \vdash^R_\Sigma m \precsim \tau\ @\ \rho}{\Gamma \vdash^R_\Sigma \mathtt{cmd}[\rho]\ \{m\} : \tau\ \mathtt{cmd}[\rho]}$$

$$\forall\mathrm{I}\ \frac{\Gamma, \pi\ \mathtt{prio}, C \vdash^R_\Sigma e : \tau}{\Gamma \vdash^R_\Sigma \Lambda\pi : C.e : \forall\pi : C.\tau} \qquad \forall\mathrm{E}\ \frac{\Gamma \vdash^R_\Sigma v : \forall\pi : C.\tau \qquad \Gamma \vdash^R [\rho'/\pi]C}{\Gamma \vdash^R_\Sigma v[\rho'] : [\rho'/\pi]\tau}$$

$$\textsc{fix}\ \frac{\Gamma, x:\tau \vdash^R_\Sigma e : \tau}{\Gamma \vdash^R_\Sigma \mathtt{fix}\, x{:}\tau\ \mathtt{is}\ e : \tau} \qquad \textsc{let}\ \frac{\Gamma \vdash^R_\Sigma e_1 : \tau_1 \qquad \Gamma, x:\tau_1 \vdash^R_\Sigma e_2 : \tau_2}{\Gamma \vdash^R_\Sigma \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

Figure 4.7: Expression typing rules.

$$\textsc{Bind}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{cmd}[\rho] \qquad \Gamma, x : \tau \vdash^R_\Sigma m \precsim \tau' \mathbin{@} \rho}{\Gamma \vdash^R_\Sigma x \leftarrow e; m \precsim \tau' \mathbin{@} \rho}$$

$$\textsc{Spawn}\quad \frac{\Gamma \vdash^R_\Sigma m \precsim \tau \mathbin{@} \rho'}{\Gamma \vdash^R_\Sigma \mathtt{spawn}[\rho'; \tau]\ \{m\} \precsim \tau\ \mathtt{thread}[\rho'] \mathbin{@} \rho} \qquad \textsc{Sync}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{thread}[\rho'] \qquad \Gamma \vdash^R \rho \preceq \rho'}{\Gamma \vdash^R_\Sigma \mathtt{sync}\ e \precsim \tau \mathbin{@} \rho}$$

$$\textsc{Poll}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{thread}[\rho']}{\Gamma \vdash^R_\Sigma \mathtt{poll}\ e \precsim \tau + \mathtt{unit} \mathbin{@} \rho} \qquad \textsc{Ret}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau}{\Gamma \vdash^R_\Sigma \mathtt{ret}\ e \precsim \tau \mathbin{@} \rho}$$

Figure 4.8: Command typing rules.

$$\textsc{hyp}\quad \frac{}{\Gamma, \rho_1 \preceq \rho_2 \vdash^R \rho_1 \preceq \rho_2} \qquad \textsc{assume}\quad \frac{(\overline{\rho}_1, \overline{\rho}_2) \in R}{\Gamma \vdash^R \overline{\rho}_1 \preceq \overline{\rho}_2} \qquad \textsc{refl}\quad \frac{}{\Gamma \vdash^R \rho \preceq \rho} \qquad \textsc{trans}\quad \frac{\Gamma \vdash^R \rho_1 \preceq \rho_2 \quad \Gamma \vdash^R \rho_2 \preceq \rho_3}{\Gamma \vdash^R \rho_1 \preceq \rho_3} \qquad \textsc{conj}\quad \frac{\Gamma \vdash^R C_1 \quad \Gamma \vdash^R C_2}{\Gamma \vdash^R C_1 \wedge C_2}$$

Figure 4.9: Constraint entailment

form $a{\sim}\tau@\rho$ indicating that thread a is running at priority ρ and will return a value of type τ. The signature is needed to check the types of thread handles.

The expression typing judgment is $\Gamma \vdash^R_\Sigma e : \tau$, indicating that under signature Σ, a partial order R of priority constants and context Γ, expression e has type τ. As usual, the variable context Γ maps variables to their types. The rules for this judgment are shown in Figure 4.7. The variable rule VAR, the rule for fixed points and the introduction and elimination rules for unit, natural numbers, functions, products and sums, are straightforward. The rule for thread handles $\texttt{tid}[a]$ looks up the thread a in the signature. The rule for encapsulated commands $\texttt{cmd}[\rho]\ \{m\}$ requires that the command m be well-typed and runnable at priority ρ, using the typing judgment for commands, which is defined below. Rule $\forall$I extends the context with both the priority variable π and the constraint C. Rule $\forall$E handles priority instantiation. When instantiating the variable π with priority ρ', the rule requires that the constraints hold with ρ' substituted for π (the constraint entailment judgment $\Gamma \vdash^R C$ is discussed below). The rule also performs the corresponding substitution in the return type.

The command typing judgment is $\Gamma \vdash^R_\Sigma m \mathrel{\dot{\sim}} \tau \mathbin{@} \rho$ and includes both the return type τ and the priority ρ at which m is runnable. The rules are shown in Figure 4.8. The rule for bind requires that e return a command of the current priority and return type τ, and then extends the context with a variable x of type τ in order to type the remaining command. The rule for $\texttt{spawn}[\rho';\tau]\ \{m\}$ requires that m be runnable at priority ρ' and return a value of type τ. The spawn command returns a thread handle of type $\tau\ \texttt{thread}[\rho']$, and may do so at any priority. The $\texttt{sync}\ e$ command requires that e have the type of a thread handle of type τ, and returns a value of type τ. The rule also checks the priority annotation on the thread's type and requires that this priority be at least the current priority. This is the condition that rules out sync commands that would cause priority inversions. The rule for $\texttt{poll}\ e$ is similar but without this requirement, since polling threads of arbitrary priority is permitted. Finally, if e has type τ, then the command $\texttt{ret}\ e$ returns a value of type τ, at any priority.

The constraint checking judgment is defined in Figure 4.9. We can conclude that a constraint holds if it appears directly in the context (rule HYP) or the partial order (rule ASSUME) or if it can be concluded from reflexivity or transitivity (rules REFL and TRANS, respectively). Finally, the conjunction $C_1 \wedge C_2$ requires that both conjuncts hold.

We use several forms of substitution in both the static and dynamic semantics. All use the standard definition of capture-avoiding substitution. We can substitute expressions for variables in expressions ($[e_2/x]e_1$) or in commands ($[e/x]m$), and we can substitute priorities for priority variables in expressions ($[\rho/\pi]e$), commands ($[\rho/\pi]m$), constraints ($[\rho/\pi]C$), contexts ($[\rho/\pi]\Gamma$), types ($[\rho/\pi]\tau$) and priorities ($[\rho'/\pi]\rho$). For each of these substitutions, we prove the principle that substitution preserves typing. These substitution principles are collected in Lemma 1.

Lemma 1 (Substitution)**.**

1. *If* $\Gamma, x:\tau \vdash^R_\Sigma e_1 : \tau'$ *and* $\Gamma \vdash^R_\Sigma e_2 : \tau$*, then* $\Gamma \vdash^R_\Sigma [e_2/x]e_1 : \tau'$.
2. *If* $\Gamma, x:\tau \vdash^R_\Sigma m \mathrel{\dot{\sim}} \tau' \mathbin{@} \rho$ *and* $\Gamma \vdash^R_\Sigma e : \tau$*, then* $\Gamma \vdash^R_\Sigma [e/x]m \mathrel{\dot{\sim}} \tau' \mathbin{@} \rho$.
3. *If* $\Gamma, \pi\ \mathit{prio} \vdash^R_\Sigma e : \tau$*, then* $[\rho/\pi]\Gamma \vdash^R_\Sigma [\rho/\pi]e : [\rho/\pi]\tau$.
4. *If* $\Gamma, \pi\ \mathit{prio} \vdash^R_\Sigma m \mathrel{\dot{\sim}} \tau \mathbin{@} \rho$*, then* $[\rho'/\pi]\Gamma \vdash^R_\Sigma [\rho'/\pi]m \mathrel{\dot{\sim}} [\rho'/\pi]\tau \mathbin{@} [\rho'/\pi]\rho$.
5. *If* $\Gamma, \pi\ \mathit{prio} \vdash^R C$*, then* $[\rho/\pi]\Gamma \vdash^R [\rho/\pi]C$.

Proof. 1. By induction on the derivation of $\Gamma, x : \tau \vdash^R_\Sigma e_1 : \tau'$. All cases are straightforward.

2. By induction on the derivation of $\Gamma, x : \tau \vdash^R_\Sigma m \precsim \tau' \mathbin{@} \rho$. All cases are straightforward.

3. By induction on the derivation of $\Gamma, \pi\ \mathtt{prio} \vdash^R_\Sigma e : \tau$. Consider one representative case.

Case

$$\dfrac{\Gamma \vdash^R_\Sigma v : \forall \pi' : C.\tau \qquad \Gamma \vdash^R [\rho'/\pi']C}{\Gamma \vdash^R_\Sigma v[\rho'] : [\rho'/\pi']\tau}\ \forall\text{E}$$

(1) $[\rho/\pi]\Gamma \vdash^R_\Sigma [\rho/\pi]v : [\rho/\pi](\forall \pi' : C.\tau)$ (induction)
(2) $[\rho/\pi](\forall \pi' : C.\tau) = \forall \pi' : [\rho/\pi]C.[\rho/\pi]\tau$ (definition)
(3) $[\rho/\pi]\Gamma \vdash^R [\rho/\pi][\rho'/\pi']C$ (induction)
(4) $[\rho/\pi][\rho'/\pi']C = [([\rho/\pi])\rho'/\pi']([\rho/\pi]C)$
(5) $[\rho/\pi]\Gamma \vdash^R_\Sigma [\rho/\pi]v[[\rho/\pi]\rho'] : [([\rho/\pi]\rho')/\pi'][\rho/\pi]\tau$ ($\forall$E)
(6) $[([\rho/\pi]\rho')/\pi'][\rho/\pi]\tau = [\rho/\pi][\rho'/\pi']\tau$

4. By induction on the derivation of $\Gamma, \pi\ \mathtt{prio} \vdash^R_\Sigma m \precsim \tau \mathbin{@} \rho$.

Case

$$\dfrac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{cmd}[\rho] \qquad \Gamma, x : \tau \vdash^R_\Sigma m \precsim \tau' \mathbin{@} \rho}{\Gamma \vdash^R_\Sigma x \leftarrow e; m \precsim \tau' \mathbin{@} \rho}\ \textsc{Bind}$$

(1) $[\rho'/\pi]\Gamma \vdash^R_\Sigma [\rho'/\pi]e : [\rho'/\pi](\tau\ \mathtt{cmd}[\rho])$ (induction)
(2) $[\rho'/\pi]\Gamma, x : [\rho'/\pi]\tau \vdash^R_\Sigma [\rho'/\pi]m \precsim [\rho'/\pi]\tau' \mathbin{@} [\rho'/\pi]\rho$ (induction)
(3) $[\rho'/\pi]\Gamma \vdash^R_\Sigma x \leftarrow [\rho'/\pi]e; [\rho'/\pi]m \precsim [\rho'/\pi]\tau' \mathbin{@} [\rho'/\pi]\rho$ (Bind)

Case

$$\dfrac{\Gamma \vdash^R_\Sigma m \precsim \tau \mathbin{@} \rho''}{\Gamma \vdash^R_\Sigma \mathtt{spawn}[\rho''; \tau]\ \{m\} \precsim \tau\ \mathtt{thread}[\rho''] \mathbin{@} \rho}\ \textsc{Spawn}$$

(1) $[\rho'/\pi]\Gamma \vdash^R_\Sigma [\rho'/\pi]m \precsim [\rho'/\pi]\tau \mathbin{@} [\rho'/\pi]\rho''$ (induction)
(2) $[\rho'/\pi]\Gamma \vdash^R_\Sigma \mathtt{spawn}[[\rho'/\pi]\rho''; [\rho'/\pi]\tau]\ \{[\rho'/\pi]m\} \precsim$
$[\rho'/\pi]\tau\ \mathtt{thread}[[\rho'/\pi]\rho''] \mathbin{@} [\rho'/\pi]\rho$ (Spawn)

Case

$$\dfrac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{thread}[\rho''] \qquad \Gamma \vdash^R \rho \preceq \rho''}{\Gamma \vdash^R_\Sigma \mathtt{sync}\ e \precsim \tau \mathbin{@} \rho}\ \textsc{Sync}$$

(1) $[\rho'/\pi]\Gamma \vdash^R_\Sigma [\rho'/\pi]e : [\rho'/\pi]\,\mathtt{thread}[[\rho'/\pi]\rho'']$ (induction)
(2) $[\rho'/\pi]\Gamma \vdash^R [\rho'/\pi]\rho \preceq [\rho'/\pi]\rho''$ (induction)
(3) $[\rho'/\pi]\Gamma \vdash^R_\Sigma \mathtt{sync}\ ([\rho'/\pi]e) \precsim [\rho'/\pi]\tau \mathbin{@} [\rho'/\pi]\rho$ (Sync)

Case

$$\dfrac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{thread}[\rho'']}{\Gamma \vdash^R_\Sigma \mathtt{poll}\ e \precsim \tau + \mathtt{unit} \mathbin{@} \rho}\ \textsc{Poll}$$

(1) $[\rho'/\pi]\Gamma \vdash^R_\Sigma [\rho'/\pi]e : [\rho'/\pi]\,\mathtt{thread}[[\rho'/\pi]\rho'']$ (induction)
(2) $[\rho'/\pi]\Gamma \vdash^R_\Sigma \mathtt{poll}\ ([\rho'/\pi]e) \precsim [\rho'/\pi]\tau \mathbin{@} [\rho'/\pi]\rho$ (Poll)

Case

$$\textsc{Ret}\ \frac{\Gamma \vdash^R_\Sigma e : \tau}{\Gamma \vdash^R_\Sigma \texttt{ret}\ e \precsim \tau\ @\ \rho}$$

(1) $[\rho'/\pi]\Gamma \vdash^R_\Sigma [\rho'/\pi]e : [\rho'/\pi]\tau$ (induction)
(2) $[\rho'/\pi]\Gamma \vdash^R_\Sigma \texttt{ret}\ [\rho'/\pi]e \precsim [\rho'/\pi]\tau\ @\ [\rho'/\pi]\rho$ (RET)

5. By induction on the derivation of $\Gamma, \pi\ \texttt{prio} \vdash^R C$. We consider the non-trivial cases.
Case

$$\textsc{trans}\ \frac{\Gamma \vdash^R \rho_1 \preceq \rho_2 \qquad \Gamma \vdash^R \rho_2 \preceq \rho_3}{\Gamma \vdash^R \rho_1 \preceq \rho_3}$$

(1) $[\rho/\pi]\Gamma \vdash^R [\rho/\pi]\rho_1 \preceq [\rho/\pi]\rho_2$ (induction)
(2) $[\rho/\pi]\Gamma \vdash^R [\rho/\pi]\rho_2 \preceq [\rho/\pi]\rho_3$ (induction)
(3) $[\rho/\pi]\Gamma \vdash^R [\rho/\pi]\rho_1 \preceq [\rho/\pi]\rho_3$ (TRANS)

Case

$$\textsc{conj}\ \frac{\Gamma \vdash^R C_1 \qquad \Gamma \vdash^R C_2}{\Gamma \vdash^R C_1 \wedge C_2}$$

(1) $[\rho/\pi]\Gamma \vdash^R [\rho/\pi]C_1$ (induction)
(2) $[\rho/\pi]\Gamma \vdash^R [\rho/\pi]C_2$ (induction)
(3) $[\rho/\pi]\Gamma \vdash^R [\rho/\pi]C_1 \wedge [\rho/\pi]C_2$ (CONJ)

□

4.2.2 Dynamic Semantics

We define a transition semantics for λ^4 . Because the operational behavior (as distinct from run-time or responsiveness, which is the focus of Chapter 5) of expressions does not depend on the priority at which they run or what other threads are running, their semantics can be defined without regard to other running threads. The semantics for commands is more complex, because it must include other threads. We will also define a syntax and dynamic semantics for *thread pools*, which are collections of all of the currently running threads.

The dynamic semantics for expressions, shown in Figure 4.10, consists of two judgments. The judgment $e\ \texttt{val}_\Sigma$ states that e is irreducible and refers only to thread names in the signature Σ.

The transition relation for expressions, $e \rightarrow^\Delta_\Sigma (\delta, e')$, indicates that expression e steps to e' after a delay δ. This delay will typically be zero, but input instructions may result in non-zero delays. The *delay assignment* Δ maps input identifiers d to a set of possible delays δ [87]. It is included in the dynamic semantics to factor out the nondeterminism inherent in the latency incurred by input operations, since this nondeterminism is neither under the control of the programmer nor the scheduler. For example, suppose d represents a user input to which the user may take between one second and five minutes to respond. Then, we would evaluate the cost semantics using a Δ such that $\Delta(d) = [1s, 5m]$.

$$\frac{}{\langle\rangle\ \mathtt{val}_\Sigma} \qquad \frac{}{\mathtt{tid}[a]\ \mathtt{val}_{\Sigma,a\sim\tau@\rho}} \qquad \frac{}{\overline{n}\ \mathtt{val}_\Sigma} \qquad \frac{}{\lambda x.e\ \mathtt{val}_\Sigma} \qquad \frac{v_1\ \mathtt{val}_\Sigma \qquad v_2\ \mathtt{val}_\Sigma}{\langle v_1, v_2\rangle\ \mathtt{val}_\Sigma}$$

$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{l}\cdot v\ \mathtt{val}_\Sigma} \qquad \frac{v\ \mathtt{val}_\Sigma}{\mathtt{r}\cdot v\ \mathtt{val}_\Sigma} \qquad \frac{}{\mathtt{cmd}[\rho]\ \{m\}\ \mathtt{val}_\Sigma} \qquad \frac{}{\Lambda\rho : C.e\ \mathtt{val}_\Sigma}$$

D-LET-STEP
$$\frac{e_1 \to_\Sigma^\Delta (\delta, e_1')}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \to_\Sigma^\Delta (\delta, \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2)}$$

D-LET
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{let}\ x = v\ \mathtt{in}\ e \to_\Sigma^\Delta (0, [v/x]e)}$$

D-IFZ-Z
$$\frac{}{\mathtt{ifz}\ 0\ \{e_1; x.e_2\} \to_\Sigma^\Delta (0, e_1)}$$

D-IFZ-NZ
$$\frac{}{\mathtt{ifz}\ \overline{n+1}\ \{e_1; x.e_2\} \to_\Sigma^\Delta (0, [\overline{n}/x]e_2)}$$

D-APP
$$\frac{v\ \mathtt{val}_\Sigma}{(\lambda x.e)\ v \to_\Sigma^\Delta (0, [v/x]e)}$$

D-PAIR
$$\frac{v_1\ \mathtt{val}_\Sigma \qquad v_2\ \mathtt{val}_\Sigma}{(v_1, v_2) \to_\Sigma^\Delta (0, \langle v_1, v_2\rangle)}$$

D-FST
$$\frac{v_1\ \mathtt{val}_\Sigma \qquad v_2\ \mathtt{val}_\Sigma}{\mathtt{fst}\ \langle v_1, v_2\rangle \to_\Sigma^\Delta (0, v_1)}$$

D-SND
$$\frac{v_1\ \mathtt{val}_\Sigma \qquad v_2\ \mathtt{val}_\Sigma}{\mathtt{snd}\ \langle v_1, v_2\rangle \to_\Sigma^\Delta (0, v_2)}$$

D-INL
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{inl}\ v \to_\Sigma^\Delta (0, \mathtt{l}\cdot v)}$$

D-INR
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{inr}\ v \to_\Sigma^\Delta (0, \mathtt{r}\cdot v)}$$

D-CASE-L
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{case}\ \mathtt{l}\cdot v\ \{x.e_1; y.e_2\} \to_\Sigma^\Delta (0, [v/x]e_1)}$$

D-CASE-R
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{case}\ \mathtt{r}\cdot v\ \{x.e_1; y.e_2\} \to_\Sigma^\Delta (0, [v/y]e_2)}$$

D-OUTPUT
$$\frac{}{\mathtt{output}\ \overline{n} \to_\Sigma^\Delta (0, \langle\rangle)}$$

D-INPUT
$$\frac{\delta \in \Delta(d)}{\mathtt{input}_d \to_\Sigma^\Delta (\delta - 1, \mathtt{in})}$$

D-IN
$$\frac{n \in \mathbb{N}}{\mathtt{in} \to_\Sigma^\Delta (0, \overline{n})}$$

D-PRAPP
$$\frac{}{(\Lambda\pi : C.e)[\rho'] \to_\Sigma^\Delta (0, [\rho'/\pi]e)}$$

D-FIX
$$\frac{}{\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e \to_\Sigma^\Delta (0, [\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e/x]e)}$$

Figure 4.10: Dynamic semantics for expressions.

$$\overline{\nu\Sigma\{\mu_1\} \uplus \mu_2 \equiv \nu\Sigma\{\mu_1 \uplus \mu_2\}} \qquad \overline{\nu\Sigma\{\nu\Sigma'\{\mu\}\} \equiv \nu\Sigma, \Sigma'\{\mu\}} \qquad \overline{\nu \cdot \{\mu\} \equiv \mu}$$

Figure 4.11: Congruence rules for thread pools.

$$\textsc{Empty}\ \frac{}{\vdash^R_\Sigma \emptyset : \cdot} \qquad \textsc{OneThread}\ \frac{\cdot \vdash^R_\Sigma m \mathrel{\dot\sim} \tau\ @\ \rho}{\vdash^R_\Sigma a \underset{\rho}{\hookrightarrow} (\delta, m) : a{\sim}\tau@\rho} \qquad \textsc{Concat}\ \frac{\vdash^R_{\Sigma,\Sigma_2} \mu_1 : \Sigma_1 \qquad \vdash^R_{\Sigma,\Sigma_1} \mu_2 : \Sigma_2}{\vdash^R_\Sigma \mu_1 \uplus \mu_2 : \Sigma_1, \Sigma_2} \qquad \textsc{Extend}\ \frac{\vdash^R_\Sigma \mu : \Sigma', \Sigma''}{\vdash^R_\Sigma \nu\Sigma'\{\mu\} : \Sigma''}$$

Figure 4.12: Typing rules for thread pools

The signature Σ does not change during expression evaluation and is used solely to determine whether thread IDs are well-formed values. Because of the structure of 2/3-cps form, the only expressions which require evaluation of subexpressions are let bindings. In the expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, subexpression e_1 is evaluated to a value and then substituted into e_2. The `ifz` construct conditions on the value of the numeral $\overline{n}$. If $n = 0$, it steps to e_1. If not, it steps to e_2, substituting $n - 1$ for x. The `case` construct conditions on whether e is a left or right injection, and steps to e_1 (resp. e_2), substituting the injected value for x (resp. y). Function applications and priority instantiations simply perform the appropriate substitution. Pair creation and injection evaluate to the equivalent value forms. One may think of these transitions (D-PAIR, D-INL and D-INR) as performing the allocation of the value on the heap, though we do not explicitly model heap allocation. Input operations are split into two steps. The rule D-INPUT nondeterministically picks a possible delay δ from Δ for the input. After $\delta - 1$ additional steps, the instruction steps to the auxiliary expression `in`. By rule D-IN, this `in` instruction nondeterministically picks an integer for the input (representing the nondeterminism of user input) and steps to it.

Evaluation of commands may interact with other running threads (e.g., by spawning new threads or synchronizing with existing ones), and so in order to define the dynamic semantics of commands, we must develop a way of talking about collections of running threads. We use *thread pools* to refer to some or all of the currently running threads. Much of the notation for thread pools is inspired by Harper [63]. Formally, a thread pool μ is a mapping of thread symbols to threads: $a \underset{\rho}{\hookrightarrow} (\delta, m)$ indicates a thread a at priority ρ running m. The thread may be ready to run immediately (if $\delta = 0$) or may be waiting on input for δ more steps. The concatenation of two thread pools is written $\mu_1 \uplus \mu_2$. Thread pools can also introduce new thread names: the thread pool $\nu\Sigma\{\mu\}$ allows the thread pool μ to use thread names bound in the signature Σ. Thread pools are not ordered; we identify thread pools up to commutativity and associativity of $\uplus$[5]. We also introduce the additional congruence rules of Figure 4.11, which allow for thread name bindings to freely change scope within a thread pool.

Figure 4.12 gives the typing rules for thread pools. The typing judgment $\vdash^R_\Sigma \mu : \Sigma'$ indicates

[5]Because threads cannot refer to threads that (transitively) spawned them, we could order the thread pool, which would allow us to prove that deadlock is not possible in λ^4 . This is outside the scope of this thesis.

D-BIND1

$$\frac{e \to_\Sigma^\Delta (\delta, e')}{x \leftarrow e; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, x \leftarrow e'; m, \emptyset)}$$

D-BIND2

$$\frac{m_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \Sigma', m_1', \mu')}{x \leftarrow \mathtt{cmd}[\rho]\ \{m_1\}; m_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \Sigma', x \leftarrow \mathtt{cmd}[\rho]\ \{m_1'\}; m_2, \mu')}$$

D-BIND3

$$\frac{e\ \mathtt{val}_\Sigma}{x \leftarrow \mathtt{cmd}[\rho]\ \{\mathtt{ret}\ e\}; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (0, \cdot, [e/x]m, \emptyset)}$$

D-SPAWN

$$\frac{b\ \mathsf{fresh}}{\mathtt{spawn}[\rho; \tau]\ \{m\} \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (0, b{\sim}\tau@\rho, \mathtt{ret}\ \mathtt{tid}[b], b \underset{\rho}{\hookrightarrow} (0, m))}$$

D-SYNC1

$$\frac{e \to_\Sigma^\Delta (\delta, e')}{\mathtt{sync}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, \mathtt{sync}\ e', \emptyset)}$$

D-SYNC2

$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{sync}\ (\mathtt{tid}[b]) \overset{v \triangleleft b}{\underset{\Sigma}{\longmapsto}}^{\Delta} (0, \cdot, \mathtt{ret}\ v, \emptyset)}$$

D-POLL1

$$\frac{e \to_\Sigma^\Delta (\delta, e')}{\mathtt{poll}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, \mathtt{poll}\ e', \emptyset)}$$

D-POLL2A

$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{poll}\ (\mathtt{tid}[b]) \overset{v ? b}{\underset{\Sigma}{\longmapsto}}^{\Delta} (0, \cdot, \mathtt{ret}\ \mathtt{inl}\ v, \emptyset)}$$

D-POLL2B

$$\frac{}{\mathtt{poll}\ (\mathtt{tid}[b]) \overset{\not{v} b}{\underset{\Sigma}{\longmapsto}}^{\Delta} (0, \cdot, \mathtt{ret}\ \mathtt{inr}\ \langle\rangle, \emptyset)}$$

D-RET

$$\frac{e \to_\Sigma^\Delta (\delta, e')}{\mathtt{ret}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, \mathtt{ret}\ e', \emptyset)}$$

Figure 4.13: Dynamic rules for commands.

that all threads of μ are well-typed assuming an ambient environment that includes the threads mentioned in Σ, and that Σ' includes the threads introduced in μ, minus any bound in a $\nu\Sigma''\{\mu''\}$ form. The rules are straightforward: the empty thread pool $\emptyset$ is always well-typed and introduces no threads, individual threads are well-typed if their commands are, and concatenations are well-typed if their components are. In a concatenation $\mu_1 \uplus \mu_2$, if μ_1 introduces the threads Σ_1 and μ_2 introduces the threads Σ_2, then μ_1 may refer to threads in Σ_2 and vice versa. If a thread pool μ is well-typed and introduces the threads in Σ', Σ'', then $\nu\Sigma'\{\mu\}$ introduces the threads in Σ'' (subtracting off the threads explicitly introduced by the binding).

With the notation for thread pools, we can now define the dynamics of commands. The transition judgment $m \overset{\alpha}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \Sigma', m', \mu')$, indicates that under signature Σ and delay assignment Δ, command m steps to m' after delay δ. The transition relation carries a label α, indicating the "action" taken by this step. At this point, actions can be the silent action ϵ, the sync action $v \triangleleft b$,

indicating that the transition receives a value v by synchronizing on thread b, the successful poll action $v \mathbin{?} b$ indicating that the transition polls b and receives value v, or the unsuccessful poll action $\not?b$ indicating that the transition polls thread b and learns it has not completed. This step may also spawn new threads, and so the judgment includes extensions to the thread pool (μ') and the signature (Σ'). Both extensions may be empty.

The rules for the transition judgment are shown in Figure 4.13. The rules for the bind construct $x \leftarrow e; m_2$ evaluate e to an encapsulated command $\mathtt{cmd}[\rho]\ \{m_1\}$, then evaluate this command to a return value $\mathtt{ret}\ v$ before substituting v for x in m_2. The spawn command $\mathtt{spawn}[\rho;\tau]\ \{m\}$ does *not* evaluate m, but simply spawns a fresh thread b to execute it, and returns a thread handle $\mathtt{tid}[b]$. The sync command $\mathtt{sync}\ e$ evaluates e to a thread handle $\mathtt{tid}[b]$, and then takes a step to $\mathtt{ret}\ v$ labeled with the action $v \triangleleft b$. Note that, because the thread b is not available to the rule, the return value v is "guessed". It will be the job of the thread pool semantics to connect this thread to the thread b and provide the appropriate return value. There are two rules for polling a thread handle. Rule D-POLL2A behaves analogously to the rule for sync and allows a thread to receive the returned value of the polled thread. The returned value is left-injected, since polling returns an option type $\tau + \mathtt{unit}$. Rule D-POLL2B silently steps to $\mathtt{r} \cdot \langle\rangle$. Finally, $\mathtt{ret}\ e$ evaluates e to a value.

We define an additional transition judgment for thread pools, which nondeterministically allows a thread to step. The judgment $\mu \overset{a/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu'$ is again annotated with an action. In this judgment, because it is not clear what thread is taking the step, the action is labeled with the thread a. Actions now also include the "return" action $v\triangleright$, indicating that the thread returns the value v. Rule DT-SYNC matches this with a corresponding sync or poll action and perform the synchronization. If a thread in μ_1 wishes to sync with b and a thread b in μ_2 wishes to return its value, then the thread pool $\mu_1 \uplus \mu_2$ can step silently, performing the synchronization. Without loss of generality, μ_1 can come first because thread pools are identified up to ordering. Rule DT-POLLA is similar, but matches return actions with successful poll actions. This rule also results in a special action ? indicating that this step performed a poll (regardless of whether it was successful). Steps involving polls will be treated separately when we introduce parallelism. Rule DT-POLLB allows thread a to take a step with an unsuccessful poll as long as the polled thread is not returning a value. The last two rules allow threads to step when concatenated with other threads and under bindings.

We will show as part of the type safety theorem that any thread pool may be, through the congruence rules, placed in a normal form $\nu\Sigma\{a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, m_1) \uplus \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, m_n)\}$ and that stepping one of these threads does not affect the rest of the thread pool other than by spawning new threads. This property, that transitions of separate threads do not impact each other, is key to parallel functional programs and allows us to cleanly talk about taking multiple steps of separate threads in parallel. Parallelism is expressed by the judgment $\mu \overset{A}{\underset{P}{\Rightarrow}}{}^{\Delta} \mu'$, which allows all of the threads in the set A to step silently in parallel. The only rule for this judgment is DT-PAR, which nondeterministically steps any number of threads that can take silent transitions. Following this, the rule takes zero or more poll steps. This forces $\mathtt{poll}$ operations to eagerly "commit" to whether they will succeed or fail based on whether their target threads have currently completed. This "early commitment" will be important in the next chapter for producing usable

DT-THREAD

$$\frac{m \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \Sigma', m', \mu')}{a \underset{\rho}{\hookrightarrow} (0, m) \overset{a/\alpha}{\underset{a\sim\tau@\rho,\Sigma}{\Longrightarrow}}{}^{\Delta} \nu\Sigma'\{a \underset{\rho}{\hookrightarrow} (\delta, m') \uplus \mu'\}}$$

DT-RET

$$\frac{v\ \mathtt{val}_\Sigma}{a \underset{\rho}{\hookrightarrow} (0, \mathtt{ret}\ v) \overset{a/v\triangleright}{\underset{a\sim\tau@\rho,\Sigma}{\Longrightarrow}}{}^{\Delta} a \underset{\rho}{\hookrightarrow} (0, \mathtt{ret}\ v)}$$

DT-SYNC

$$\frac{\Sigma = \Sigma', a{\sim}\tau_a@\rho_a, b{\sim}\tau_b@\rho_b \qquad \mu_1 \overset{a/v\triangleleft b}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_1' \qquad \mu_2 \overset{b/v\triangleright}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_2}{\mu_1 \uplus \mu_2 \overset{a/\epsilon}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_1' \uplus \mu_2}$$

DT-POLLA

$$\frac{\Sigma = \Sigma', a{\sim}\tau_a@\rho_a, b{\sim}\tau_b@\rho_b \qquad \mu_1 \overset{a/v?b}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_1' \qquad \mu_2 \overset{b/v\triangleright}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_2}{\mu_1 \uplus \mu_2 \overset{a/?}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_1' \uplus \mu_2}$$

DT-POLLB

$$\frac{\Sigma = \Sigma', a{\sim}\tau_a@\rho_a, b{\sim}\tau_b@\rho_b \qquad \mu_1 \overset{a/?b}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_1' \qquad \mu_2 \overset{b/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_2' \qquad \alpha \neq v\triangleright}{\mu_1 \uplus \mu_2 \overset{a/?}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_1' \uplus \mu_2}$$

DT-CONCAT

$$\frac{\mu_1 \overset{a/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_1'}{\mu_1 \uplus \mu_2 \overset{a/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu_1' \uplus \mu_2}$$

DT-EXTEND

$$\frac{\mu \overset{b/\alpha}{\underset{\Sigma, a\sim\tau@\rho}{\Longrightarrow}}{}^{\Delta} \mu'}{\nu a{\sim}\tau@\rho\{\mu\} \overset{b/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \nu a{\sim}\tau@\rho\{\mu'\}}$$

Figure 4.14: Dynamic rules for thread pools.

DT-PAR

$$\frac{\begin{array}{c}\mu \equiv \nu\Sigma\{\mu_0 \uplus a_1 \underset{\rho_1}{\hookrightarrow} (0, m_1) \uplus \ldots a_n \underset{\rho_n}{\hookrightarrow} (0, m_n)\} \\ (\forall 1 \leq i \leq n)\mu \overset{a_i/\epsilon}{\underset{\cdot}{\Longrightarrow}}{}^{\Delta} \nu\Sigma, \Sigma_i'\{\mu_0 \uplus a_1 \underset{\rho_1}{\hookrightarrow} (0, m_1) \uplus \ldots \uplus a_i \underset{\rho_i}{\hookrightarrow} (\delta_i, m_i') \uplus \mu_i' \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} (0, m_n)\} \\ \nu\Sigma, \Sigma_1', \ldots, \Sigma_n'\{\mu_0 \uplus a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, m_1') \uplus \mu_1' \uplus \ldots a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, m_n') \uplus \mu_n'\} \overset{b_1/?}{\underset{\cdot}{\Longrightarrow}}{}^{\Delta} \mu_1 \overset{b_2/?}{\underset{\cdot}{\Longrightarrow}}{}^{\Delta} \ldots \overset{b_k/?}{\underset{\cdot}{\Longrightarrow}}{}^{\Delta} \mu_k \\ \mu_k \text{ polled}\end{array}}{\mu \overset{\{a_1,\ldots,a_n\}}{\underset{P}{\Longrightarrow}}{}^{\Delta} Decr(\mu_n)}$$

$$\begin{array}{rcl} Decr(a \underset{\rho}{\hookrightarrow} (0, m)) & \triangleq & a \underset{\rho}{\hookrightarrow} (0, m) \\ Decr(a \underset{\rho}{\hookrightarrow} (\delta + 1, m)) & \triangleq & a \underset{\rho}{\hookrightarrow} (\delta, m) \\ Decr(\mu_1 \uplus \mu_2) & \triangleq & Decr(\mu_1) \uplus Decr(\mu_2) \\ Decr(\nu\Sigma\{\mu\}) & \triangleq & \nu\Sigma\{Decr(\mu)\} \end{array}$$

Figure 4.15: Parallel step judgment.

$$\frac{}{x \text{ polled}} \quad \frac{}{\langle\rangle \text{ polled}} \quad \frac{n \in \mathbb{N}}{\overline{n} \text{ polled}} \quad \frac{}{\lambda x.e \text{ polled}} \quad \frac{}{\langle v_1, v_2\rangle \text{ polled}}$$

$$\frac{}{\mathtt{l} \cdot v \text{ polled}} \quad \frac{}{\mathtt{r} \cdot v \text{ polled}} \quad \frac{m \text{ polled}}{\mathtt{cmd}[\rho]\ \{m\} \text{ polled}} \quad \frac{}{\Lambda\pi : C.e \text{ polled}}$$

$$\frac{}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \text{ polled}} \quad \frac{}{\mathtt{ifz}\ v\ \{e_1; x.e_2\} \text{ polled}} \quad \frac{}{v_1\ v_2 \text{ polled}} \quad \frac{}{\langle v_1, v_2\rangle \text{ polled}}$$

$$\frac{}{\mathtt{fst}\ v \text{ polled}} \quad \frac{}{\mathtt{snd}\ v \text{ polled}} \quad \frac{}{\mathtt{inl}\ v \text{ polled}} \quad \frac{}{\mathtt{inr}\ v \text{ polled}} \quad \frac{}{\mathtt{case}\ v\ \{x.e_1; y.e_2\} \text{ polled}}$$

$$\frac{}{\mathtt{output}\ v \text{ polled}} \quad \frac{}{\mathtt{input}_d \text{ polled}} \quad \frac{}{v[\rho] \text{ polled}} \quad \frac{}{\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e \text{ polled}}$$

$$\frac{e \text{ polled}}{x \leftarrow e; m \text{ polled}} \quad \frac{}{\mathtt{spawn}[\rho; \tau]\ \{m\} \text{ polled}} \quad \frac{}{\mathtt{sync}\ e \text{ polled}} \quad \frac{e \text{ polled}}{\mathtt{poll}\ e \text{ polled}} \quad \frac{}{\mathtt{ret}\ e \text{ polled}}$$

$$\frac{m \text{ polled}}{a \underset{\rho}{\hookrightarrow} (\delta, m) \text{ polled}} \quad \frac{\mu_1 \text{ polled} \qquad \mu_2 \text{ polled}}{\mu_1 \uplus \mu_2 \text{ polled}} \quad \frac{\mu \text{ polled}}{\nu\Sigma\{\mu\} \text{ polled}}$$

Figure 4.16: Rules for the polled judgment.

$$\frac{}{\vdash^R_\Sigma \epsilon\ \mathsf{action}} \qquad \frac{\cdot \vdash^R_{\Sigma,b\sim\tau@\rho} v : \tau}{\vdash^R_{\Sigma,b\sim\tau@\rho} v \triangleleft b\ \mathsf{action}} \qquad \frac{\cdot \vdash^R_{\Sigma,b\sim\tau@\rho} v : \tau}{\vdash^R_{\Sigma,b\sim\tau@\rho} v \triangleright\ \mathsf{action}}$$

$$\frac{\cdot \vdash^R_{\Sigma,b\sim\tau@\rho} v : \tau}{\vdash^R_{\Sigma,b\sim\tau@\rho} v\ ?\ b\ \mathsf{action}} \qquad \frac{}{\vdash^R_{\Sigma,b\sim\tau@\rho} \not?b\ \mathsf{action}} \qquad \frac{}{\vdash^R_\Sigma ?\ \mathsf{action}}$$

Figure 4.17: Static semantics for actions.

cost DAGs from programs. The thread pool takes poll steps until all `poll` operations have committed, which is captured in the `polled` judgment, defined on expressions, commands and thread pools in Figure 4.16. The judgment simply requires that the thread pool not contain expressions of the form `poll tid`$[b]$. Finally, any delays on threads are decremented using the auxiliary form $Decr(\cdot)$. We do not impose any sort of scheduling algorithm in the semantics, nor even a maximum number of threads. When discussing cost bounds, we will quantify over executions which choose threads in certain ways.

We prove a version of the standard progress theorem for each syntactic class. Progress for expressions is standard: a well-typed expression is either a value or can take a step. The progress statement for commands is similar, because commands can step (with a sync action) even if they are waiting for other threads. The statement for thread pools is somewhat counter-intuitive. One might expect it to state that if a thread pool is well-typed, then either all threads are complete or the thread pool can take a step. This statement is true but too weak to be useful; because of the non-determinism in our semantics, such a theorem would allow for one thread to enter a "stuck" state as long as any other thread is still able to make progress (for example, if it is in an infinite loop). Instead, we state that, in a well-typed thread pool, *every* thread is either complete, is delayed, or is *active*, that is, able to take a step.

The progress theorems for commands and thread pools also state that, if the command or thread pool can take a step, the action performed by that step is well-typed. The typing rules for actions are shown in Figure 4.17 and require that the value returned or received match the type of the thread.

Theorem 4 (Progress)**.** *1. If* $\cdot \vdash^R_\Sigma e : \tau$*, then either* $e\ \mathtt{val}_\Sigma$ *or* $e \rightarrow^\Delta_\Sigma (\delta, e')$*.*

2. If $\cdot \vdash^R_\Sigma m \precsim \tau\ @\ \rho$*, then either* $m = \mathtt{ret}\ e$ *where* $e\ \mathtt{val}_\Sigma$ *or* $m \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta, \Sigma', m', \mu)$ *where* $\vdash^R_\Sigma \alpha\ \mathsf{action}$*.*

3. If $\vdash^R_\Sigma \mu : \Sigma'$ *and*

$$\Sigma', \Sigma'' = a_1 \sim \tau_1 @ \rho_1, \ldots, a_n \sim \tau_n @ \rho_n$$

then

$$\mu \equiv \nu\Sigma''\{a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, m_1) \uplus \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, m_n)\}$$

and for all $i \in [1, n]$*, if* $\delta_n > 0$*, then* $\mu \overset{a_i/\alpha}{\underset{\Sigma,\Sigma'}{\Longrightarrow}}{}^\Delta \mu'$ *and* $\vdash^R_{\Sigma,\Sigma'} \alpha\ \mathsf{action}$*.*

Proof. 1. By induction on the derivation of $\cdot \vdash^R_\Sigma e : \tau$. Consider two representative cases.

Case

$$\frac{\Gamma \vdash^R_\Sigma e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash^R_\Sigma e_2 : \tau_2}{\Gamma \vdash^R_\Sigma \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \textsc{let}$$

(1) $e_1\ \texttt{val}_\Sigma$ or $e_1 \to^\Delta_\Sigma (\delta, e_1')$ (induction)

Subcase: $e_1\ \texttt{val}_\Sigma$

(a) $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to^\Delta_\Sigma (0, [e_1/x]e_2)$ (D-LET)

Subcase: $e_1 \to^\Delta_\Sigma (\delta, e_1')$

(a) $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to^\Delta_\Sigma (\delta, \texttt{let}\ x = e_1'\ \texttt{in}\ e_2)$ (D-LET-STEP)

Case

$$\frac{\Gamma \vdash^R_\Sigma v : \texttt{nat} \qquad \Gamma \vdash^R_\Sigma e_1 : \tau \qquad \Gamma, x : \texttt{nat} \vdash^R_\Sigma e_2 : \tau}{\Gamma \vdash^R_\Sigma \texttt{ifz}\ v\ \{e_1; x.e_2\} : \tau}\ \textsc{nat}\mathbf{E}$$

(1) $v = \overline{n}$ (canonical forms)

Subcase: $n = 0$

(a) $e \to^\Delta_\Sigma e_1$ (D-IFZ-Z)

Subcase: $n > 0$

(a) $e \to^\Delta_\Sigma [\overline{n-1}/x]e_2$ (D-IFZ-NZ)

2. By induction on the derivation of $\cdot \vdash^R_\Sigma m \precsim \tau\ @\ \rho$.

Case

$$\frac{\Gamma \vdash^R_\Sigma e : \tau\ \texttt{cmd}[\rho] \qquad \Gamma, x : \tau \vdash^R_\Sigma m \precsim \tau'\ @\ \rho}{\Gamma \vdash^R_\Sigma x \leftarrow e; m \precsim \tau'\ @\ \rho}\ \textsc{Bind}$$

(1) $e\ \texttt{val}_\Sigma$ or $e \to^\Delta_\Sigma (\delta, e')$ (induction)

Subcase: $e \to^\Delta_\Sigma (\delta, e')$

(a) $x \leftarrow e; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta, \cdot, x \leftarrow e'; m, \emptyset)$ (D-BIND1)

Subcase: $e\ \texttt{val}_\Sigma$

(a) $e = \texttt{cmd}[\rho]\ \{m_1\}$ (canonical forms)

(b) $m_1 = \texttt{ret}\ v$ or $m_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta, \Sigma', m_1', \mu')$ (induction)

Subcase: $m_1 = \texttt{ret}\ v$

(a) $x \leftarrow e; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^\Delta (0, \cdot, [v/x]m, \emptyset)$ (D-BIND3)

Subcase: $m_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta, \Sigma', m_1', \mu')$

(a) $x \leftarrow e; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta, \Sigma', x \leftarrow \texttt{cmd}[\rho]\ \{m_1'\}; m, \mu')$ (D-BIND2)

Case

$$\textsc{Spawn}\ \frac{\Gamma \vdash^R_\Sigma m \precsim \tau \mathbin{@} \rho'}{\Gamma \vdash^R_\Sigma \mathtt{spawn}[\rho';\tau]\ \{m\} \precsim \tau\ \mathtt{thread}[\rho'] \mathbin{@} \rho}$$

(1) $\mathtt{spawn}[\rho';\tau]\ \{m\} \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (0, b{\sim}\tau@\rho', \mathtt{ret}\ \mathtt{tid}[b], b \underset{\rho'}{\hookrightarrow} (0, m))$ (D-SPAWN)

Case

$$\textsc{Sync}\ \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{thread}[\rho'] \qquad \Gamma \vdash^R \rho \preceq \rho'}{\Gamma \vdash^R_\Sigma \mathtt{sync}\ e \precsim \tau \mathbin{@} \rho}$$

(1) $e\ \mathtt{val}_\Sigma$ or $e \to^\Delta_\Sigma (\delta, e')$ (induction)

Subcase: $e\ \mathtt{val}_\Sigma$

(a) $e = \mathtt{tid}[b]$ (canonical forms)

(b) $\mathtt{sync}\ e \overset{v \lhd b}{\underset{\Sigma}{\longmapsto}}{}^{\Delta} (0, \cdot, \mathtt{ret}\ v, \emptyset)$ (D-SYNC2)

Subcase: $e \to^\Delta_\Sigma (\delta, e')$

(a) $\mathtt{sync}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \cdot, \mathtt{sync}\ e', \emptyset)$ (D-SYNC1)

Case

$$\textsc{Poll}\ \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathtt{thread}[\rho']}{\Gamma \vdash^R_\Sigma \mathtt{poll}\ e \precsim \tau + \mathtt{unit} \mathbin{@} \rho}$$

(1) $e\ \mathtt{val}_\Sigma$ or $e \to^\Delta_\Sigma (\delta, e')$ (induction)

Subcase: $e\ \mathtt{val}_\Sigma$

(a) $e = \mathtt{tid}[b]$ (canonical forms)

(b) $\mathtt{poll}\ e \overset{?b}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (0, \cdot, \mathtt{ret}\ \mathtt{inr}\ \langle\rangle, \emptyset)$ (D-POLL2B)

Subcase: $e \to^\Delta_\Sigma (\delta, e')$

(a) $\mathtt{poll}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \cdot, \mathtt{poll}\ e', \emptyset)$ (D-POLL1)

Case

$$\textsc{Ret}\ \frac{\Gamma \vdash^R_\Sigma e : \tau}{\Gamma \vdash^R_\Sigma \mathtt{ret}\ e \precsim \tau \mathbin{@} \rho}$$

(1) $e\ \mathtt{val}_\Sigma$ or $e \rightarrow^{\Delta}_{\Sigma} (\delta, e')$ (induction)

Subcase: $e\ \mathtt{val}_\Sigma$

(a) Conclusion holds by assumption

Subcase: $e \rightarrow^{\Delta}_{\Sigma} (\delta, e')$

(a) $\mathtt{ret}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \cdot, \mathtt{ret}\ e', \emptyset)$ (D-RET)

3. By induction on the derivation of $\vdash^R_\Sigma \mu : \Sigma'$. We consider the interesting cases.

Case

$$\textsc{Concat}\ \frac{\vdash^R_{\Sigma,\Sigma_2} \mu_1 : \Sigma_1 \qquad \vdash^R_{\Sigma,\Sigma_1} \mu_2 : \Sigma_2}{\vdash^R_\Sigma \mu_1 \uplus \mu_2 : \Sigma_1, \Sigma_2}$$

(1) $\mu_1 \equiv \nu\Sigma'_1\{a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, m_1) \uplus \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, m_n)\}$ (induction)

(2) $\mu_2 \equiv \nu\Sigma'_2\{a_{n+1} \underset{\rho_{n+1}}{\hookrightarrow} (\delta_{n+1}, m_{n+1}) \uplus \ldots \uplus a_k \underset{\rho_k}{\hookrightarrow} (\delta_k, m_k)\}$ (induction)

(3) $\Sigma_1, \Sigma'_1 = a_1 \sim \tau_1 @ \rho_1, \ldots, a_n \sim \tau_n @ \rho_n$ (induction)

(4) $\Sigma_2, \Sigma'_2 = a_{n+1} \sim \tau_{n+1} @ \rho_{n+1}, \ldots, a_k \sim \tau_k @ \rho_k$ (induction)

(5) $\forall i \in [1, n].\delta_i = 0 \Rightarrow \mu_1 \overset{a_i/\alpha_i}{\underset{\Sigma,\Sigma_2,\Sigma_1}{\Longrightarrow}}{}^{\Delta} \mu'_1$ (induction)

(6) $\forall i \in [n+1, k].\delta_i = 0 \Rightarrow \mu_2 \overset{a_i/\alpha_i}{\underset{\Sigma,\Sigma_1,\Sigma_2}{\Longrightarrow}}{}^{\Delta} \mu'_2$ (induction)

(7) $\mu_1 \uplus \mu_2 \equiv \nu\Sigma'_1, \Sigma'_2\{a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, m_1) \uplus \ldots \uplus a_k \underset{\rho_k}{\hookrightarrow} (\delta_k, m_k)\}$ (congruence rules)

(8) $\Sigma_1, \Sigma'_1, \Sigma_2, \Sigma'_2 = a_1 \sim \tau_1 @ \rho_1, \ldots, a_k \sim \tau_k @ \rho_k$

(9) $\forall i \in [1, k].\delta_i = 0 \Rightarrow \mu \overset{a_i/\alpha_i}{\underset{\Sigma,\Sigma_2,\Sigma_1}{\Longrightarrow}}{}^{\Delta} \mu'$ (D-CONCAT)

Case

$$\textsc{Extend}\ \frac{\vdash^R_\Sigma \mu : \Sigma', \Sigma''}{\vdash^R_\Sigma \nu\Sigma'\{\mu\} : \Sigma''}$$

(1) $\mu \equiv \nu\Sigma'''\{a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, m_1) \uplus \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, m_n)\}$
where $\Sigma', \Sigma'', \Sigma''' = a_1 \sim \tau_1 @ \rho_1, \ldots, a_n \sim \tau_n @ \rho_n$ (induction)

(2) $\forall i \in [1, n].\delta_i = 0 \Rightarrow \mu \overset{\alpha}{\underset{\Sigma,\Sigma',\Sigma''}{\Longrightarrow}}{}^{\Delta} \mu''$ (induction)

(3) $\nu\Sigma'\{\mu\} \equiv \nu\Sigma', \Sigma'''\{a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, m_1) \uplus \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, m_n)\}$ (congruence rules)

(4) $\forall i \in [1, n].\delta_i = 0 \Rightarrow \nu\Sigma'\{\mu\} \overset{\alpha}{\underset{\Sigma,\Sigma',\Sigma''}{\Longrightarrow}}{}^{\Delta} \nu\Sigma'\{\mu''\}$ (D-EXTEND)

□

The preservation theorem is also split into components for expressions, commands and thread pools. The theorem for commands requires that any new threads spawned (μ') meet the extension of the signature (Σ').

Theorem 5 (Preservation). *1. If* $\cdot \vdash^R_\Sigma e : \tau$ *and* $e \rightarrow^{\Delta}_{\Sigma} (\delta, e')$, *then* $\cdot \vdash^R_\Sigma e' : \tau$.

2. *If* $\cdot \vdash^R_\Sigma m \precsim \tau \mathbin{@} \rho$ *and* $m \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \Sigma', m', \mu')$ *and* $\vdash^R_\Sigma \alpha$ action *then* $\cdot \vdash^R_{\Sigma,\Sigma'} m' \precsim \tau \mathbin{@} \rho$ *and* $\vdash^R_\Sigma \mu' : \Sigma'$.
3. *If* $\vdash^R_\Sigma \mu : \Sigma'$ *and* $\mu \overset{\alpha}{\underset{\Sigma}{\Rightarrow}}{}^{\Delta} \mu'$ *then* $\vdash^R_\Sigma \mu' : \Sigma'$
4. *If* $\vdash^R_\cdot \mu : \Sigma$ *and* $\mu \overset{A}{\underset{P}{\Rightarrow}}{}^{\Delta} \mu'$ *then* $\vdash^R_\cdot \mu' : \Sigma$.

Proof. 1. By induction on the derivation of $e \rightarrow^\Delta_\Sigma (\delta, e')$.

2. By induction on the derivation of $m \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \Sigma', m', \mu')$.

Case

$$\textsc{D-Bind1}\ \frac{e \rightarrow^\Delta_\Sigma (\delta, e')}{x \leftarrow e; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \cdot, x \leftarrow e'; m, \emptyset)}$$

(1) $\cdot \vdash^R_\Sigma e : \tau'\ \mathtt{cmd}[\rho]$ (inversion on BIND)
(2) $x : \tau' \vdash^R_\Sigma m \precsim \tau \mathbin{@} \rho$ (inversion on BIND)
(3) $\cdot \vdash^R_\Sigma e' \precsim \tau'\ \mathtt{cmd}[\rho']@$ (induction)
(4) $\cdot \vdash^R_\Sigma x \leftarrow e'; m \precsim \tau \mathbin{@} \rho$ (BIND)

Case

$$\textsc{D-Bind2}\ \frac{m_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \Sigma', m_1', \mu')}{x \leftarrow \mathtt{cmd}[\rho]\ \{m_1\}; m_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \Sigma', x \leftarrow \mathtt{cmd}[\rho]\ \{m_1'\}; m_2, \mu')}$$

(1) $\cdot \vdash^R_\Sigma m_1 \precsim \tau' \mathbin{@} \rho$ (inversion on BIND)
(2) $x : \tau' \vdash^R_\Sigma m_2 \precsim \tau \mathbin{@} \rho$ (inversion on BIND)
(3) $\cdot \vdash^R_{\Sigma,\Sigma'} m_1' \precsim \tau' \mathbin{@} \rho$ (induction)
(4) $\vdash^R_\Sigma \mu' : \Sigma'$ (induction)
(5) $x : \tau' \vdash^R_{\Sigma,\Sigma'} m_2 \precsim \tau \mathbin{@} \rho$ (weakening)
(6) $\cdot \vdash^R_{\Sigma,\Sigma'} x \leftarrow \mathtt{cmd}[\rho]\ \{m_1'\}; m_2 \precsim \tau \mathbin{@} \rho$ (BIND)

Case

$$\textsc{D-Bind3}\ \frac{e\ \mathtt{val}_\Sigma}{x \leftarrow \mathtt{cmd}[\rho]\ \{\mathtt{ret}\ e\}; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (0, \cdot, [e/x]m, \emptyset)}$$

(1) $\cdot \vdash^R_\Sigma e : \tau'$ (inversion on BIND)
(2) $x : \tau' \vdash^R_\Sigma m \precsim \tau \mathbin{@} \rho$ (inversion on BIND)
(3) $\cdot \vdash^R_\Sigma [e/x]m \precsim \tau' \mathbin{@} \rho$ (Lemma 1)

Case

$$\textsc{D-Spawn}\ \frac{b\ \text{fresh}}{\mathtt{spawn}[\rho'; \tau']\ \{m\} \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (0, b \sim \tau' @ \rho', \mathtt{ret}\ \mathtt{tid}[b], b \underset{\rho'}{\hookrightarrow} (0, m))}$$

(1) $\cdot \vdash^R_{\Sigma, b\sim\tau'@\rho'} \mathtt{ret}\ \mathtt{tid}[b] \precsim \tau'\ \mathtt{thread}[\rho'] \mathbin{@} \rho$ (TID, RET)
(2) $\cdot \vdash^R_\Sigma m \precsim \tau' \mathbin{@} \rho'$ (inversion on SPAWN)
(3) $\vdash^R_\Sigma b \underset{\rho}{\hookrightarrow} (0, m) : b\sim\tau'@\rho'$ (ONETHREAD)

Case

D-SYNC1
$$\frac{e \to^\Delta_\Sigma (\delta, e')}{\mathtt{sync}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^\Delta (\delta, \cdot, \mathtt{sync}\ e', \emptyset)}$$

(1) $\cdot \vdash^R_\Sigma e : \tau\ \mathtt{thread}[\rho']$ and $\cdot \vdash^R \rho \preceq \rho'$ (inversion on SYNC)
(2) $\cdot \vdash^R_\Sigma e' : \tau\ \mathtt{thread}[\rho']$ (induction)
(3) $\cdot \vdash^R_\Sigma \mathtt{sync}\ e' \precsim \tau \mathbin{@} \rho$ (SYNC)

Case

D-SYNC2
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{sync}\ (\mathtt{tid}[b]) \overset{v \lhd b}{\underset{\Sigma}{\mapsto}}^\Delta (0, \cdot, \mathtt{ret}\ v, \emptyset)}$$

(1) $\cdot \vdash^R_\Sigma \mathtt{tid}[b] : \tau\ \mathtt{thread}[\rho']$ and $\Gamma \vdash^R \rho \preceq \rho'$ (inversion on SYNC)
(2) $b\sim\tau@\rho' \in \Sigma$ (inversion on TID)
(3) $\cdot \vdash^R_\Sigma v : \tau$ (inversion on action typing rules)
(4) $\cdot \vdash^R_\Sigma \mathtt{ret}\ v \precsim \tau \mathbin{@} \rho$ (RET)

Case

D-POLL1
$$\frac{e \to^\Delta_\Sigma (\delta, e')}{\mathtt{sync}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^\Delta (\delta, \cdot, \mathtt{poll}\ e', \emptyset)}$$

(1) $\cdot \vdash^R_\Sigma e : \tau\ \mathtt{thread}[\rho']$ (inversion on POLL)
(2) $\cdot \vdash^R_\Sigma e' : \tau\ \mathtt{thread}[\rho']$ (induction)
(3) $\cdot \vdash^R_\Sigma \mathtt{poll}\ e' \precsim \tau + \mathtt{unit} \mathbin{@} \rho$ (POLL)

Case

D-POLL2A
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{poll}\ (\mathtt{tid}[b]) \overset{v ? b}{\underset{\Sigma}{\mapsto}}^\Delta (0, \cdot, \mathtt{ret}\ \mathtt{inl}\ v, \emptyset)}$$

(1) $\cdot \vdash^R_\Sigma \mathtt{tid}[b] : \tau\ \mathtt{thread}[\rho']$ (inversion on POLL)
(2) $b\sim\tau@\rho' \in \Sigma$ (inversion on TID)
(3) $\cdot \vdash^R_\Sigma v : \tau$ (inversion on action typing rules)
(4) $\cdot \vdash^R_\Sigma \mathtt{ret}\ \mathtt{inl}\ v \precsim \tau + \mathtt{unit} \mathbin{@} \rho$ ($+I_1$, RET)

Case

D-POLL2B
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{poll}\ (\mathtt{tid}[b]) \overset{\not? b}{\underset{\Sigma}{\mapsto}}^\Delta (0, \cdot, \mathtt{ret}\ \mathtt{inr}\ \langle\rangle, \emptyset)}$$

(1) $\cdot \vdash^R_\Sigma \mathtt{ret}\ \mathtt{inr}\ \langle\rangle \precsim \tau + \mathtt{unit} \mathbin{@} \rho$ (UNITI, $+I_2$, RET)

Case

$$\textsc{D-Ret}\ \frac{e \rightarrow_{\Sigma}^{\Delta} (\delta, e')}{\texttt{ret}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, \texttt{ret}\ e', \emptyset)}$$

(1) $\cdot \vdash_{\Sigma}^{R} e : \tau$ (inversion on RET)
(2) $\cdot \vdash_{\Sigma}^{R} e' : \tau$ (induction)
(3) $\cdot \vdash_{\Sigma}^{R} \texttt{ret}\ e' \precsim \tau\ @\ \rho$ (RET)

3. By induction on the derivation of $\mu \overset{a/\alpha}{\underset{\Sigma}{\Longrightarrow}}^{\Delta} \mu'$. The non-trivial rule is the one for threads:

$$\textsc{DT-Thread}\ \frac{m \overset{\alpha}{\underset{\Sigma}{\mapsto}}^{\rho} (\delta, \Sigma'', m', \mu')}{a \underset{\rho}{\hookrightarrow} (0, m) \overset{a/\alpha}{\underset{a \sim \tau @ \rho, \Sigma}{\Longrightarrow}}^{\Delta} \nu \Sigma''\{a \underset{\rho}{\hookrightarrow} (\delta, m') \uplus \mu'\}}$$

(1) $\cdot \vdash_{\Sigma}^{R} m \precsim \tau\ @\ \rho$ (inversion on ONETHREAD)
(2) $\cdot \vdash_{\Sigma,\Sigma''}^{R} m' \precsim \tau\ @\ \rho$ (induction)
(3) $\vdash_{\Sigma}^{R} \mu' : \Sigma''$ (induction)
(4) $\vdash_{\Sigma,\Sigma''}^{R} a \underset{\rho}{\hookrightarrow} (\delta, m') \uplus \mu' : \Sigma', \Sigma''$ (Weakening, ONETHREAD, CONCAT)
(5) $\vdash_{\Sigma}^{R} \nu \Sigma''\{a \underset{\rho}{\hookrightarrow} (\delta, m') \uplus \mu'\} : \Sigma'$ (EXTEND)

4. There is one case, DT-PAR. Note that the parallel transition can be accomplished by repeated single transitions of the form $\mu \overset{a/\alpha}{\underset{\cdot}{\Longrightarrow}}^{\Delta} \mu'$, followed by a decrement. By induction, all of the transitions preserve typing. By inspection of the typing rules, typing is preserved by the decrement operation.

□

4.3 Elaboration of PriML to λ^4

In this section, we define an elaboration from PriML to λ^4 . This can be seen as giving a formal semantics to PriML programs by placing them in correspondence with λ^4 programs. We begin by formally defining the syntax of PriML, in Figure 4.18. We use the metavariables $\hat{e}$ and $\hat{m}$ (and variants) to refer to PriML expressions and commands, respectively, in order to distinguish them from their λ^4 counterparts. We also add syntactic classes for instructions i, priority annotations c, declarations d, toplevel declarations o and programs P. Expressions are more or less the same as in λ^4 , with the difference that let bindings introduce zero or more declarations into the scope of an expression (the notation $\vec{d}$ stands for a list of declarations). The command layer of PriML is split into two syntactic classes: commands and instructions, where a command is a sequence of instructions, each binding its return value. Instructions now include $\texttt{do}(e)$, which runs an encapsulated command (expressible in λ^4 as $x \leftarrow e; \texttt{ret}\ x$). Declarations d, which may be included in let bindings, introduce expressions, functions (which bind one or more variables,

Expressions	$\hat{e}$	::=	$x \mid \langle\rangle \mid \overline{n} \mid \texttt{ifz}\ \hat{e}\ \{\hat{e}; x.\hat{e}\} \mid \lambda x.\hat{e} \mid \hat{e}\ \hat{e} \mid (\hat{e}, \hat{e}) \mid \texttt{fst}\ \hat{e} \mid \texttt{snd}\ \hat{e}$
			$\mid \texttt{inl}\ \hat{e} \mid \texttt{inr}\ \hat{e} \mid \texttt{case}\ \hat{e}\ \{x.\hat{e}; y.\hat{e}\} \mid \texttt{output}\ \hat{e} \mid \texttt{input}_d$
			$\mid \texttt{cmd}[\rho]\ \{\hat{m}\} \mid \hat{e}[\rho] \mid \texttt{let}\ \vec{d}\ \texttt{in}\ \hat{e}\ \texttt{end}$
Instructions	i	::=	$\texttt{do}(\hat{e}) \mid \texttt{spawn}[\rho; \tau]\ \{\hat{m}\} \mid \texttt{sync}\ \hat{e} \mid \texttt{ret}\ \hat{e}$
Commands	$\hat{m}$	::=	$x \leftarrow i; \hat{m} \mid i$
Prio. Annotations	c	::=	$\pi : C$
Declarations	d	::=	$\texttt{val}\ x = \hat{e} \mid \texttt{fun}\ f(\vec{x}^{+}) = \hat{e} \mid \texttt{fun}[\vec{c}^{+}]\ f(\vec{x}^{+}) = \hat{e}$
Toplevel Decls.	o	::=	$d \mid \texttt{priority}\ \overline{\rho} \mid \texttt{order}\ \overline{\rho} \prec \overline{\rho}$
Programs	P	::=	$\texttt{main}\ \{\hat{m}\} \mid o\ P$

Figure 4.18: Formal syntax of PriML for elaboration.

indicated by the notation $\vec{x}^{+}$), or priority-polymorphic functions (which bind one or more priority variables and one or more expression variables). All of these declarations can appear at the top level, as can priority and order declarations. These are the separate class of toplevel declarations. Finally, a program is zero or more toplevel declarations followed by a command to run as the "main" thread.

For simplicity of the formal definition of elaboration, a number of features of PriML are omitted from the formalism. For example, we include natural numbers and unit as the only base types, while PriML has integers, booleans, strings, etc. We also omit many useful features of a language such as algebraic datatypes, mutually recursive functions and exceptions. These are present in our implementation (although we have not implemented mutually recursive priority-polymorphic functions). The elaboration of these features is standard (e.g., [64]). Because fairness declarations do not affect the static or dynamic semantics of λ^4 , they would simply be erased during elaboration and so we also omit these.

The main work of elaboration is converting to 2/3-cps and hoisting priority and order declarations out of the code so that they may be presented as a pre-defined partially-ordered set, as required by the λ^4 semantics. The overall goal of elaboration then is to convert a PriML program P into a λ^4 command m together with a partial order R of priorities. This proceeds in a number of mutually recursive stages, with one elaboration judgment for each syntactic class of PriML.

Each judgment, except the one for programs, is annotated with the type of the λ^4 expression or command that is produced by elaboration. In this sense, elaboration is typed. We do not explicitly define a static semantics for PriML. Instead, the static semantics is given by elaboration itself. If a PriML program elaborates to a λ^4 command, that command is well-typed at a distinguished priority `bot` (abstractly notated $\bot$) with the type given in the elaboration judgment (Theorem 6). If the elaboration rules do not allow a valid elaboration to be derived for a PriML program, then we will reject that program as ill-typed.

The elaboration judgments are:

- $\Gamma \vdash_R \hat{e} \leadsto_e e : \tau$ converts a PriML expression to a λ^4 expression.
- $\Gamma \vdash_R i \leadsto_i m \dot{\sim} \tau\ @\ \rho$ converts a PriML instruction to a λ^4 command. These will be

$$\frac{}{\Gamma \vdash_R \langle\rangle \rightsquigarrow_e \langle\rangle : \mathtt{unit}} \qquad \frac{}{\Gamma \vdash_R \overline{n} \rightsquigarrow_e \overline{n} : \mathtt{nat}}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \mathtt{nat} \qquad \Gamma \vdash_R \hat{e}_1 \rightsquigarrow_e e_1 : \tau \qquad \Gamma, y : \mathtt{nat} \vdash_R \hat{e}_2 \rightsquigarrow_e e_2 : \tau \qquad x \text{ fresh}}{\Gamma \vdash_R \mathtt{ifz}\ \hat{e}\ \{\hat{e}_1; y.\hat{e}_2\} \rightsquigarrow_e \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{ifz}\ x\ \{e_1; y.e_2\} : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash_R \hat{e} \rightsquigarrow_e e : \tau_2}{\Gamma \vdash_R \lambda x.\hat{e} \rightsquigarrow_e \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash_R \hat{e}_1 \rightsquigarrow_e e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_R \hat{e}_2 \rightsquigarrow_e e_2 : \tau_1 \qquad x, y \text{ fresh}}{\Gamma \vdash_R \hat{e}_1\ \hat{e}_2 \rightsquigarrow_e \mathtt{let}\ x = e_1\ \mathtt{in}\ \mathtt{let}\ y = e_2\ \mathtt{in}\ x\ y : \tau_2}$$

$$\frac{\Gamma \vdash_R \hat{e}_1 \rightsquigarrow_e e_1 : \tau_1 \qquad \Gamma \vdash_R \hat{e}_2 \rightsquigarrow_e e_2 : \tau_2 \qquad x, y \text{ fresh}}{\Gamma \vdash_R (\hat{e}_1, \hat{e}_2) \rightsquigarrow_e \mathtt{let}\ x = e_1\ \mathtt{in}\ \mathtt{let}\ y = e_2\ \mathtt{in}\ (x, y) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau_1 \times \tau_2 \qquad x \text{ fresh}}{\Gamma \vdash_R \mathtt{fst}\ \hat{e} \rightsquigarrow_e \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{fst}\ x : \tau_1} \qquad \frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau_1 \times \tau_2 \qquad x \text{ fresh}}{\Gamma \vdash_R \mathtt{snd}\ \hat{e} \rightsquigarrow_e \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{snd}\ x : \tau_2}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau_1 \qquad x \text{ fresh}}{\Gamma \vdash_R \mathtt{inl}\ \hat{e} \rightsquigarrow_e \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{inl}\ x : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau_2 \qquad x \text{ fresh}}{\Gamma \vdash_R \mathtt{inr}\ \hat{e} \rightsquigarrow_e \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{inr}\ x : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau_1 + \tau_2 \qquad \Gamma, y : \tau_1 \vdash_R \hat{e}_1 \rightsquigarrow_e e_1 : \tau' \qquad \Gamma, z : \tau_2 \vdash_R \hat{e}_2 \rightsquigarrow_e e_2 : \tau' \qquad x \text{ fresh}}{\Gamma \vdash_R \mathtt{case}\ \hat{e}\ \{y.\hat{e}_1; z.\hat{e}_2\} \rightsquigarrow_e \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{case}\ x\ \{y.e_1; z.e_2\} : \tau'}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \mathtt{nat} \qquad x \text{ fresh}}{\Gamma \vdash_R \mathtt{output}\ \hat{e} \rightsquigarrow_e \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{output}\ x : \mathtt{unit}} \qquad \frac{}{\Gamma \vdash_R \mathtt{input}_d \rightsquigarrow_e \mathtt{input}_d : \mathtt{nat}}$$

$$\frac{\Gamma \vdash_R \hat{m} \rightsquigarrow_m m \div \tau \mathbin{@} \rho}{\Gamma \vdash_R \mathtt{cmd}[\rho]\ \{\hat{m}\} \rightsquigarrow_e \mathtt{cmd}[\rho]\ \{m\} : \tau\ \mathtt{cmd}[\rho]}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \forall \pi : C.\tau \qquad \Gamma \vdash [\rho'/\pi]C \qquad x \text{ fresh}}{\Gamma \vdash_R \hat{e}[\rho'] \rightsquigarrow_e \mathtt{let}\ x = e\ \mathtt{in}\ x[\rho'] : [\rho'/\pi]\tau}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau}{\Gamma \vdash_R \mathtt{let}\ \ \mathtt{in}\ \hat{e}\ \mathtt{end} \rightsquigarrow_e e : \tau} \qquad \frac{\Gamma \vdash_R d \rightsquigarrow_d (x, e') : \tau' \qquad \Gamma, x : \tau' \vdash_R \mathtt{let}\ \vec{d}\ \mathtt{in}\ \hat{e}\ \mathtt{end} \rightsquigarrow_e e : \tau}{\Gamma \vdash_R \mathtt{let}\ d\ \vec{d}\ \mathtt{in}\ \hat{e}\ \mathtt{end} \rightsquigarrow_e \mathtt{let}\ x = e'\ \mathtt{in}\ e : \tau}$$

Figure 4.19: Elaboration of expressions.

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau\ \mathtt{cmd}[\rho] \qquad x\ \mathsf{fresh}}{\Gamma \vdash_R \mathtt{do}(\hat{e}) \rightsquigarrow_i x \leftarrow e; \mathtt{ret}\ x \precsim \tau\ @\ \rho}$$

$$\frac{\Gamma \vdash_R \hat{m} \rightsquigarrow_m m \precsim \tau\ @\ \rho'}{\Gamma \vdash_R \mathtt{spawn}[\rho'; \tau]\ \{\hat{m}\} \rightsquigarrow_i \mathtt{spawn}[\rho'; \tau]\ \{m\} \precsim \tau\ \mathtt{thread}[\rho']\ @\ \rho}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau\ \mathtt{thread}[\rho'] \qquad \Gamma \vdash^R \rho \preceq \rho'}{\Gamma \vdash_R \mathtt{sync}\ \hat{e} \rightsquigarrow_i \mathtt{sync}\ e \precsim \tau\ @\ \rho}$$

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau\ \mathtt{thread}[\rho']}{\Gamma \vdash_R \mathtt{poll}\ \hat{e} \rightsquigarrow_i \mathtt{poll}\ e \precsim \tau + \mathtt{unit}\ @\ \rho} \qquad \frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau}{\Gamma \vdash_R \mathtt{ret}\ \hat{e} \rightsquigarrow_i \mathtt{ret}\ e \precsim \tau\ @\ \rho}$$

$$\frac{\Gamma \vdash_R i \rightsquigarrow_i m \precsim \tau\ @\ \rho \qquad \Gamma, x : \tau \vdash_R \hat{m}' \rightsquigarrow_m m' \precsim \tau'\ @\ \rho}{\Gamma \vdash_R x \leftarrow i; \hat{m}' \rightsquigarrow_m x \leftarrow \mathtt{cmd}[\rho]\ \{m\}; m' \precsim \tau'\ @\ \rho} \qquad \frac{\Gamma \vdash_R i \rightsquigarrow_i m \precsim \tau\ @\ \rho}{\Gamma \vdash_R i \rightsquigarrow_m m \precsim \tau\ @\ \rho}$$

Figure 4.20: Elaboration of instructions and commands.

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau}{\Gamma \vdash_R \mathtt{val}\ x = \hat{e} \rightsquigarrow_d (x, e) : \tau}$$

$$\frac{\tau_f = \tau_1 \to \ldots \to \tau_n \to \tau \qquad \Gamma, f : \tau_f, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_R \hat{e} \rightsquigarrow_e e : \tau}{\Gamma \vdash_R \mathtt{fun}\ f(x_1 \ldots x_n) = \hat{e} \rightsquigarrow_d (f, \mathtt{fix}\ f{:}\tau_f\ \mathtt{is}\ \lambda x_1. \ldots \lambda x_n.e) : \tau_f}$$

$$\frac{\begin{array}{c}\tau_f = \forall \pi_1 : C_1. \ldots \forall \pi_n : C_n.\tau_1 \to \ldots \to \tau_m \to \tau \\ \Gamma, f : \tau_f, \pi_1\ \mathtt{prio}, C_1, \ldots, \pi_n\ \mathtt{prio}, C_n, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash_R \hat{e} \rightsquigarrow_e e : \tau \\ e_{body} = \Lambda \pi_1 : C_1. \ldots \Lambda \pi_n : C_n. \lambda x_1. \ldots \lambda x_m.e\end{array}}{\Gamma \vdash_R \mathtt{fun}[\pi_1 : C_1 \ldots \pi_n : C_n]\ f(x_1 \ldots x_m) = \hat{e} \rightsquigarrow_d (f, \mathtt{fix}\ f{:}\tau_f\ \mathtt{is}\ e_{body}) : \tau_f}$$

$$\frac{R \cup \{\overline{\rho}\}; \Gamma \vdash P \rightsquigarrow_P m; R'}{R; \Gamma \vdash \mathtt{priority}\ \overline{\rho}\ P \rightsquigarrow_P m; R' \cup \{\overline{\rho}\}}$$

$$\frac{\overline{\rho}_1, \overline{\rho}_2 \in R \qquad \overline{\rho}_2 \not\preceq \overline{\rho}_1 \qquad (R \cup \{(\overline{\rho}_1, \overline{\rho}_2)\}); \Gamma \vdash P \rightsquigarrow_P m; R'}{R; \Gamma \vdash \mathtt{order}\ \overline{\rho}_1 \prec \overline{\rho}_2\ P \rightsquigarrow_P m; R' \cup \{(\overline{\rho}_1, \overline{\rho}_2)\}}$$

$$\frac{\Gamma \vdash_R d \rightsquigarrow_d (x, e) : \tau \qquad R; \Gamma, x : \tau \vdash P \rightsquigarrow_P m; R'}{R; \Gamma \vdash d\ P \rightsquigarrow_P x \leftarrow \mathtt{cmd}[\bot]\ \{\mathtt{ret}\ e\}; m; R'} \qquad \frac{\Gamma \vdash_R \hat{m} \rightsquigarrow_m m \precsim \tau\ @\ \bot}{R; \Gamma \vdash \mathtt{main}\ \{\hat{m}\} \rightsquigarrow_P m; \emptyset}$$

Figure 4.21: Elaboration of declarations and programs.

sequenced together using the bind operator of λ^4 .

- $\Gamma \vdash_R \hat{m} \leadsto_m m \precsim \tau \mathbin{@} \rho$ converts a PriML command to a λ^4 command.
- $\Gamma \vdash_R d \leadsto_d (x, e) : \tau$ converts a PriML declaration to the variable it binds together with the λ^4 expression bound to it.
- $R; \Gamma \vdash P \leadsto_P m; R'$ converts a PriML program to a λ^4 command, together with a partially ordered set of priorities. The existing partial order R is extended by R'.

All of the judgments except the final one are parametrized by the ambient partial order R of priorities. The final judgment collects priority orderings declared in the program to produce the partial order. Recall from Section 4.1 that programmer-defined orderings induce a strict order, from which we construct the natural partial order.

The elaboration rules for expressions are defined in Figure 4.19. The most interesting feature of these rules is that they bind subexpressions using let bindings in order to convert expressions to 2/3-cps form. Otherwise, the rules closely follow the typing rules. The rules for let bindings unroll the sequence of declarations, elaborating one at a time into λ^4 let bindings. For each declaration in sequence, the declaration is elaborated into a pair (x, e'). The body of the let binding (together with the elaboration of the remaining bindings) is bound as a let.

The elaboration rules for instructions and commands are defined in Figure 4.20. The elaboration rule for `do` instructions evaluates the expression, checks that it has the type of an encapsulated command, and elaborates it to a command that uses the bind operation of λ^4 to bind the command's return value to a fresh variable. Other instructions elaborate to the corresponding command (with any sub-components recursively elaborated). This uniformly produces a command for each instruction, which can be sequenced together using binding in the first command elaboration rule. The second command elaboration rule simply elaborates the final instruction into a command. The $m_1; m_2$ form of sequencing we used in Section 4.1, which does not bind the return value of m_1, can easily be desugared to $x \leftarrow m_1; m_2$, where x does not appear free in m_2. To keep the formalism simple, this syntactic sugar is not included in the rules.

The elaboration rules for declarations and programs appear in Figure 4.21. The rule for `val` declarations simply elaborates the expression and returns a pair of the bound variable and elaborated expression. Function declarations (both expression-level and priority-level) are recursive, and so elaborating them involves finding a fixed point. The elaboration of priority-monomorphic `fun` declarations elaborates the body in a context that includes all of the arguments, as well as the function itself. This is then placed inside n nested λ-abstractions to bind the required arguments. Finally, this expression is nested inside a fixed-point operator to introduce the recursive binding of the function name. Something similar occurs for priority-polymorphic `fun` declarations. The body is nested inside m λ-abstractions to introduce the expression variables, followed by n priority-level abstractions to introduce the priority variables. Finally, this whole expression is wrapped in a fixed-point operator.

The last elaboration judgment covers both toplevel declarations and programs, and produces a command corresponding to the entire PriML program. The first three rules for this judgment elaborate each form of toplevel declaration followed by the remainder of the program, and the final rule evaluates the "main" command that terminates the program. Priority and order declarations are erased from the program; their elaboration is simply the elaboration of the remainder of the program. However, the priorities and ordering relations introduced by these declarations are

collected into the partial order that is returned at the end of elaboration. Each time a new ordering constraint $\overline{\rho}_1 \prec \overline{\rho}_2$ is introduced, we check that $\overline{\rho}_2 \not\preceq \overline{\rho}_1$. This premise checks that the new constraint will not induce a cycle in the priority relation. Ordinary declarations (`val` and `fun`) at the top level are elaborated using the declaration elaboration rules. The elaborated expression is wrapped in a command and then introduced using binding. Finally, the main command is elaborated at priority $\bot$ and returned.

Theorem 6 shows that elaboration is type-correct in that elaboration will produce a well-typed λ^4 program. The theorem has a conjunct for each elaboration judgment.

Theorem 6 (Correctness of elaboration)**.** *1. If* $\Gamma \vdash_R \hat{e} \leadsto_e e : \tau$*, then* $\Gamma \vdash^R_\cdot e : \tau$.

2. If $\Gamma \vdash_R i \leadsto_i m \precsim \tau \mathbin{@} \rho$*, then* $\Gamma \vdash^R_\cdot m \precsim \tau \mathbin{@} \rho$.

3. If $\Gamma \vdash_R \hat{m} \leadsto_m m \precsim \tau \mathbin{@} \rho$*, then* $\Gamma \vdash^R_\cdot m \precsim \tau \mathbin{@} \rho$.

4. If $\Gamma \vdash_R d \leadsto_d (x, e) : \tau$*, then* $\Gamma \vdash^R_\cdot e : \tau$.

5. If $R; \Gamma \vdash P \leadsto_P m; R'$*, then* $\Gamma \vdash^{R \cup R'}_\cdot m \precsim \tau \mathbin{@} \bot$.

Proof. 1. By induction on the derivation of $\Gamma \vdash_R \hat{e} \leadsto_e e : \tau$. Other than introducing let bindings, elaboration closely follows the typing rules, and so the cases are straightforward. We show two representative examples.

Case

$$\frac{\Gamma \vdash_R \hat{e}_1 \leadsto_e e_1 : \tau \qquad \Gamma, y : \mathtt{nat} \vdash_R \hat{e}_2 \leadsto_e e_2 : \tau \qquad x \text{ fresh} \qquad \Gamma \vdash_R \hat{e} \leadsto_e e : \mathtt{nat}}{\Gamma \vdash_R \mathtt{ifz}\ \hat{e}\ \{\hat{e}_1; y.\hat{e}_2\} \leadsto_e \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{ifz}\ x\ \{e_1; y.e_2\} : \tau}$$

(1) $\Gamma \vdash^R_\cdot e : \mathtt{nat}$ (induction)
(2) $\Gamma \vdash^R_\cdot e_1 : \tau$ (induction)
(3) $\Gamma, y : \mathtt{nat} \vdash^R_\cdot e_2 : \tau$ (induction)
(4) $\Gamma, x : \mathtt{nat} \vdash^R_\cdot \mathtt{ifz}\ x\ \{e_1; y.e_2\} : \tau$ (NATE)
(5) $\Gamma \vdash^R_\cdot \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{ifz}\ x\ \{e_1; y.e_2\} : \tau$ (LET)

Case

$$\frac{\Gamma \vdash_R d \leadsto_d (x, e') : \tau' \qquad \Gamma, x : \tau' \vdash_R \mathtt{let}\ \vec{d}\ \mathtt{in}\ \hat{e}\ \mathtt{end} \leadsto_e e : \tau}{\Gamma \vdash_R \mathtt{let}\ d\ \vec{d}\ \mathtt{in}\ \hat{e}\ \mathtt{end} \leadsto_e \mathtt{let}\ x = e'\ \mathtt{in}\ e : \tau}$$

(1) $\Gamma \vdash^R_\cdot e' : \tau'$ (induction)
(2) $\Gamma, x : \tau' \vdash^R_\cdot e : \tau$ (induction)
(3) $\Gamma \vdash^R_\cdot \mathtt{let}\ x = e'\ \mathtt{in}\ e : \tau$ (LET)

2. By induction on the derivation of $\Gamma \vdash_R i \leadsto_i m \precsim \tau \mathbin{@} \rho$.

Case

$$\frac{\Gamma \vdash_R \hat{e} \leadsto_e e : \tau\ \mathtt{cmd}[\rho] \qquad x \text{ fresh}}{\Gamma \vdash_R \mathtt{do}(\hat{e}) \leadsto_i x \leftarrow e; \mathtt{ret}\ x \precsim \tau \mathbin{@} \rho}$$

(1) $\Gamma \vdash^R_\cdot e : \tau\ \mathtt{cmd}[\rho]$ (induction)
(2) $\Gamma \vdash^R_\cdot x \leftarrow e; \mathtt{ret}\ x \precsim \tau \mathbin{@} \rho$ (VAR, RET, BIND)

Case

$$\frac{\Gamma \vdash_R \hat{m} \leadsto_m m \precsim \tau \mathbin{@} \rho'}{\Gamma \vdash_R \mathtt{spawn}[\rho'; \tau]\ \{\hat{m}\} \leadsto_i \mathtt{spawn}[\rho'; \tau]\ \{m\} \precsim \tau\ \mathtt{thread}[\rho'] \mathbin{@} \rho}$$

(1) $\Gamma \vdash^R_\cdot m \dot{\sim} \tau \mathbin{@} \rho'$ (induction)
(2) $\Gamma \vdash^R_\cdot \mathtt{spawn}[\rho'; \tau]\ \{m\} \dot{\sim} \tau\ \mathtt{thread}[\rho'] \mathbin{@} \rho$ (SPAWN)

Case

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau\ \mathtt{thread}[\rho'] \qquad \Gamma \vdash^R \rho \preceq \rho'}{\Gamma \vdash_R \mathtt{sync}\ \hat{e} \rightsquigarrow_i \mathtt{sync}\ e \dot{\sim} \tau \mathbin{@} \rho}$$

(1) $\Gamma \vdash^R_\cdot e : \tau\ \mathtt{thread}[\rho']$ (induction)
(2) $\Gamma \vdash^R_\cdot \mathtt{sync}\ e \dot{\sim} \tau \mathbin{@} \rho$ (SYNC)

Case

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau\ \mathtt{thread}[\rho']}{\Gamma \vdash_R \mathtt{poll}\ \hat{e} \rightsquigarrow_i \mathtt{poll}\ e \dot{\sim} \tau + \mathtt{unit} \mathbin{@} \rho}$$

(1) $\Gamma \vdash^R_\cdot e : \tau\ \mathtt{thread}[\rho']$ (induction)
(2) $\Gamma \vdash^R_\cdot \mathtt{poll}\ e \dot{\sim} \tau + \mathtt{unit} \mathbin{@} \rho$ (POLL)

Case

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau}{\Gamma \vdash_R \mathtt{ret}\ \hat{e} \rightsquigarrow_i \mathtt{ret}\ e \dot{\sim} \tau \mathbin{@} \rho}$$

(1) $\Gamma \vdash^R_\cdot e : \tau$ (induction)
(2) $\Gamma \vdash^R_\cdot \mathtt{ret}\ e \dot{\sim} \tau \mathbin{@} \rho$ (RET)

3. By induction on the derivation of $\Gamma \vdash_R \hat{m} \rightsquigarrow_m m \dot{\sim} \tau \mathbin{@} \rho$.

Case

$$\frac{\Gamma \vdash_R i \rightsquigarrow_i m \dot{\sim} \tau \mathbin{@} \rho \qquad \Gamma, x : \tau \vdash_R \hat{m}' \rightsquigarrow_m m' \dot{\sim} \tau' \mathbin{@} \rho}{\Gamma \vdash_R x \leftarrow i; \hat{m}' \rightsquigarrow_m x \leftarrow \mathtt{cmd}[\rho]\ \{m\}; m' \dot{\sim} \tau' \mathbin{@} \rho}$$

(1) $\Gamma \vdash^R_\cdot m \dot{\sim} \tau \mathbin{@} \rho$ (induction)
(2) $\Gamma, x : \tau \vdash^R_\cdot m' \dot{\sim} \tau' \mathbin{@} \rho$ (induction)
(3) $\Gamma \vdash^R_\cdot x \leftarrow \mathtt{cmd}[\rho]\ \{m\}; m' \dot{\sim} \tau' \mathbin{@} \rho$ (CMDI, BIND)

Case

$$\frac{\Gamma \vdash_R i \rightsquigarrow_i m \dot{\sim} \tau \mathbin{@} \rho}{\Gamma \vdash_R i \rightsquigarrow_m m \dot{\sim} \tau \mathbin{@} \rho}$$

This case follows directly from the induction hypothesis.

4. By induction on the derivation of $\Gamma \vdash_R d \rightsquigarrow_d (x, e) : \tau$

Case

$$\frac{\Gamma \vdash_R \hat{e} \rightsquigarrow_e e : \tau}{\Gamma \vdash_R \mathtt{val}\ x = \hat{e} \rightsquigarrow_d (x, e) : \tau}$$

This case follows directly from the induction hypothesis.

Case

$$\frac{\tau_f = \tau_1 \to \ldots \to \tau_n \to \tau \qquad \Gamma, f : \tau_f, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_R \hat{e} \rightsquigarrow_e e : \tau}{\Gamma \vdash_R \mathtt{fun}\ f(x_1 \ldots x_n) = \hat{e} \rightsquigarrow_d (f, \mathtt{fix}\ f{:}\tau_f\ \mathtt{is}\ \lambda x_1. \ldots \lambda x_n.e) : \tau_f}$$

(1) $\Gamma, f : \tau_f, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash^R_\cdot e : \tau$ (induction)
(2) $\Gamma, f : \tau_f \vdash^R_\cdot \lambda x_1. \ldots \lambda x_n.e : \tau_f$ ($\to$I)
(3) $\Gamma \vdash^R_\cdot \mathtt{fix}\ f{:}\tau_f\ \mathtt{is}\ \lambda x_1. \ldots \lambda x_n.e : \tau_f$ (FIX)

Case

$$\frac{\begin{array}{c}\tau_f = \forall\pi_1 : C_1 \ldots \forall\pi_n : C_n.\tau_1 \to \ldots \to \tau_m \to \tau \\ \Gamma, f : \tau_f, \pi_1\ \mathtt{prio}, C_1, \ldots, \pi_n\ \mathtt{prio}, C_n, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash_R \hat{e} \leadsto_e e : \tau \\ e_{body} = \Lambda\pi_1 : C_1 \ldots \Lambda\pi_n : C_n.\lambda x_1 \ldots \lambda x_m.e\end{array}}{\Gamma \vdash_R \mathtt{fun}[\pi_1 : C_1 \ldots \pi_n : C_n]\ f(x_1 \ldots x_m) = \hat{e} \leadsto_d (f, \mathtt{fix}\ f{:}\tau_f\ \mathtt{is}\ e_{body}) : \tau_f}$$

(1) $\Gamma, f : \tau_f, \pi_1\ \mathtt{prio}, C_1, \ldots, \pi_n\ \mathtt{prio}, C_n, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash^R_\cdot e : \tau$ (induction)

(2) $\Gamma, f : \tau_f, \pi_1\ \mathtt{prio}, C_1, \ldots, \pi_n\ \mathtt{prio}, C_n \vdash^R_\cdot \lambda x_1 \ldots \lambda x_m.e : \tau_1 \to \ldots \to \tau_m \to \tau$ ($\to$I)

(3) $\Gamma, f : \tau_f \vdash^R_\cdot \Lambda\pi_1 : C_1 \ldots \Lambda\pi_n : C_n.\lambda x_1 \ldots \lambda x_m.e : \tau_f$ ($\forall$I)

(4) $\Gamma \vdash^R_\cdot \mathtt{fix}\ f{:}\tau_f\ \mathtt{is}\ \Lambda\pi_1 : C_1 \ldots \Lambda\pi_n : C_n.\lambda x_1 \ldots \lambda x_m.e : \tau_f$ (FIX)

5. By induction on the derivation of $R; \Gamma \vdash P \leadsto_P m; R'$.

Case

$$\frac{R \cup \{\overline{\rho}\}; \Gamma \vdash P \leadsto_P m; R'}{R; \Gamma \vdash \mathtt{priority}\ \overline{\rho}\ P \leadsto_P m; R' \cup \{\overline{\rho}\}}$$

This case follows directly from the induction hypothesis because $(R \cup \{\overline{\rho}\}) \cup R' = R \cup (R' \cup \{\overline{\rho}\})$.

Case

$$\frac{\overline{\rho}_1, \overline{\rho}_2 \in R \qquad \overline{\rho}_2 \not\preceq \overline{\rho}_1 \qquad (R \cup \{(\overline{\rho}_1, \overline{\rho}_2)\}); \Gamma \vdash P \leadsto_P m; R'}{R; \Gamma \vdash \mathtt{order}\ \overline{\rho}_1 \prec \overline{\rho}_2\ P \leadsto_P m; R' \cup \{(\overline{\rho}_1, \overline{\rho}_2)\}}$$

This case follows directly from the induction hypothesis because $(R \cup \{(\overline{\rho}_1, \overline{\rho}_2)\}) \cup R' = R \cup (R' \cup \{(\overline{\rho}_1, \overline{\rho}_2)\})$.

Case

$$\frac{\Gamma \vdash_R d \leadsto_d (x, e) : \tau' \qquad R; \Gamma, x : \tau' \vdash P \leadsto_P m; R'}{R; \Gamma \vdash d\ P \leadsto_P x \leftarrow \mathtt{cmd}[\bot]\ \{\mathtt{ret}\ e\}; m; R'}$$

(1) $\Gamma \vdash^R_\cdot e : \tau'$ (induction)

(2) $\Gamma, x : \tau' \vdash^{R \cup R'}_\cdot m \precsim \tau\ @\ \bot$ (induction)

(3) $\Gamma \vdash^{R \cup R'}_\cdot e : \tau'$ (weakening)

(4) $\Gamma \vdash^{R \cup R'}_\cdot x \leftarrow \mathtt{cmd}[\bot]\ \{\mathtt{ret}\ e\}; m \precsim \tau\ @\ \bot$ (RET, CMDI, BIND)

Case

$$\frac{\Gamma \vdash_R \hat{m} \leadsto_m m \precsim \tau\ @\ \bot}{R; \Gamma \vdash \mathtt{main}\ \{\hat{m}\} \leadsto_P m; \emptyset}$$

This case follows directly from the induction hypothesis.

□

The main result is the correctness of the elaboration of an entire PriML program in an empty context with only the priority $\bot$ initially defined. This is a simple application of the last part of Theorem 6.

Corollary 1. *If* $\{\bot\}; \cdot \vdash P \leadsto_P m; R$, *then* $\cdot \vdash^{R \cup \{\bot\}}_\cdot m \precsim \tau\ @\ \bot$.

Chapter 5

A Cost Model for Responsive Parallelism

> BENVOLIO. What sadness lengthens Romeo's hours?
> ROMEO. Not having that, which, having, makes them short.
>
> *Romeo and Juliet* (I.1.168–69)

In this chapter, we develop a cost model for PriML, which can be used, together with the results of Chapter 3, to predict the throughput and responsiveness of programs. In Section 5.1, we present a cost semantics that evaluates a program, producing a value and a DAG of the form described in Chapter 3, and show that a well-typed PriML program produces a well-formed DAG, allowing the results we have shown about cost DAGs to be applied. In Section 5.2, we validate the cost semantics using a technique inspired by the *provably-efficient implementations* of Blelloch and Greiner [16]: we establish a correspondence between the cost semantics and the transition system of Section 4.2.2 and show that the evaluation of programs under the transition system is properly predicted by the cost semantics.

5.1 Cost Semantics for λ^4

This section introduces the cost semantics. Unlike the operational semantics of Section 4.2.2, this is an evaluation semantics that does not fully specify the order in which threads are evaluated. Figures 5.1-5.3 show the cost semantics for λ^4 using three judgments.

The judgment for expressions is $e \Downarrow_\Delta v; \vec{u}$, indicating that expression e evaluates to value v and produces thread $\vec{u}$. As in the operational semantics, we use a delay assignment Δ to map input identifiers to sets of possible delays. The rule C-INPUT then leads to a family of DAGs. For each element $\delta \in \Delta(d)$, the family contains a DAG where the edge corresponding to this input operation has weight δ. The two-level syntax of λ^4 ensures that expressions cannot produce spawn or join edges in the cost graph, and so the rules for the expression judgment are otherwise quite straightforward: subexpressions are evaluated to produce sequences of operations, which are then composed sequentially.

The judgment $\sigma; \Sigma; m \Downarrow_\Delta^{(a,\rho)} \sigma'; \Sigma'; v; g$ indicates that m evaluates to $\texttt{ret}\ v$ and produces the graph g. Because threads in our cost graphs are named and annotated with priorities, the current thread's name and priority are included in the judgment. The judgment also includes the

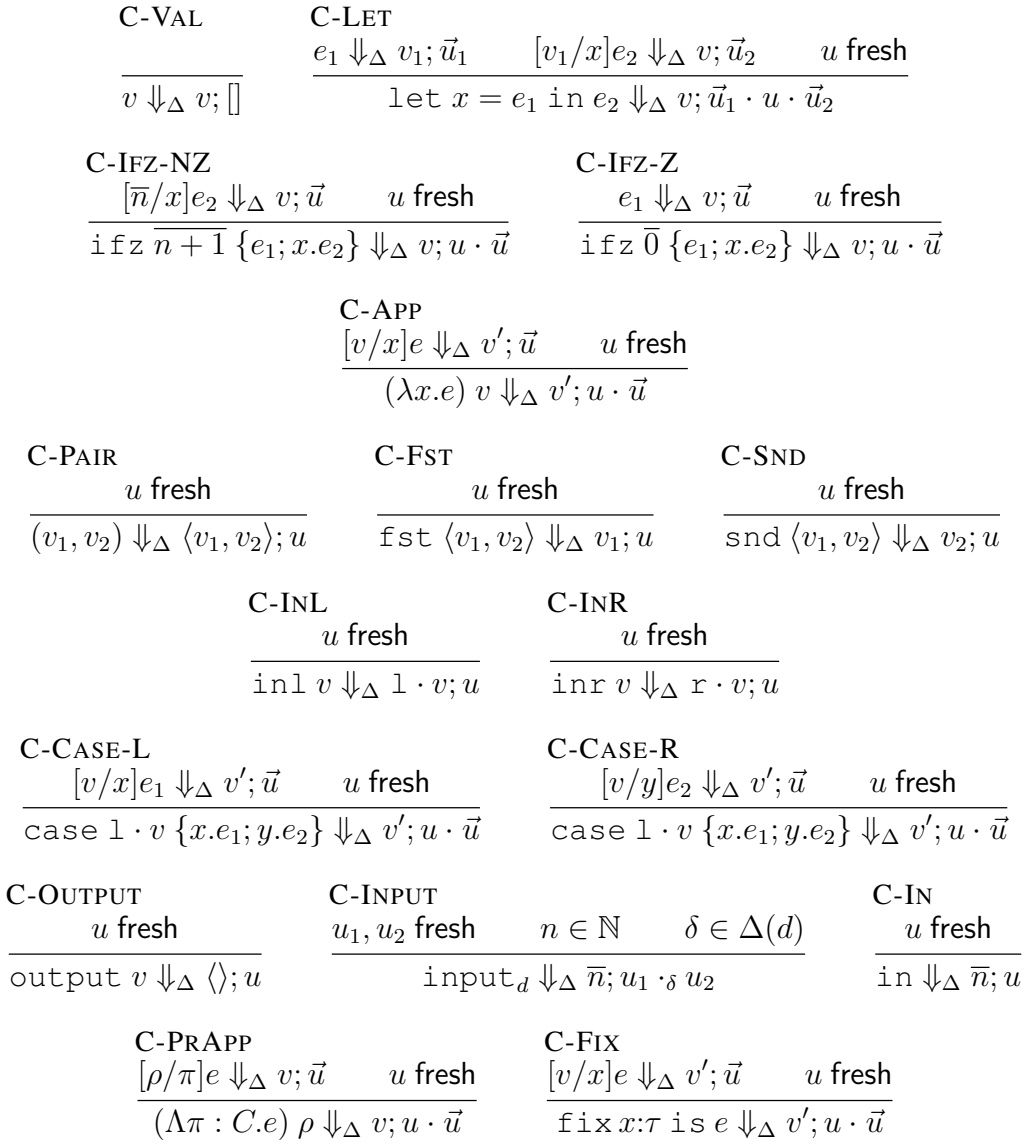


Figure 5.1: Cost semantics for expressions.

C-BIND

$$\frac{e \Downarrow_\Delta \texttt{cmd}[\rho]\ \{m_1\}; \vec{u}_1 \qquad \sigma; \Sigma; m_1 \Downarrow_\Delta^{(a,\rho)} \sigma_1; \Sigma_1; v; g_1 \qquad u \text{ fresh} \qquad \sigma_1; \Sigma_1; [v/x]m_2 \Downarrow_\Delta^{(a,\rho)} \sigma_2; \Sigma_2; v'; g_2}{\sigma; \Sigma; x \leftarrow e; m_2 \Downarrow_\Delta^{(a,\rho)} \sigma_2; \Sigma_2; v'; [\vec{u}_1] \oplus_a g_1 \oplus_a [u] \oplus_a g_2}$$

C-SPAWN

$$\frac{b \text{ fresh} \qquad \sigma; \Sigma; m \Downarrow_\Delta^{(b,\rho')} \sigma'; \Sigma, \Sigma'; v; (\mathcal{T}, E_s, E_j, E_w) \qquad u \text{ fresh}}{\sigma; \Sigma; \texttt{spawn}[\rho'; \tau]\ \{m\} \Downarrow_\Delta^{(a,\rho)} \sigma', b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \texttt{tid}[b]; (a \underset{\rho}{\hookrightarrow} (0, u) \uplus \mathcal{T}, E_s \cup \{(u, b)\}, E_j, E_w)}$$

C-SYNC

$$\frac{e \Downarrow_\Delta \texttt{tid}[b]; \vec{u} \qquad u \text{ fresh}}{\sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \texttt{sync}\ e \Downarrow_\Delta^{(a,\rho)} \sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho', \Sigma'; v; (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \{(b, u)\}, \emptyset)}$$

C-POLL-SOME

$$\frac{e \Downarrow_\Delta \texttt{tid}[b]; \vec{u} \qquad u \text{ fresh}}{\sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \texttt{poll}\ e \Downarrow_\Delta^{(a,\rho)} \sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho', \Sigma'; \texttt{l} \cdot v; (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \emptyset, \{(b, u)\})}$$

C-POLL-NONE

$$\frac{e \Downarrow_\Delta \texttt{tid}[b]; \vec{u} \qquad u \text{ fresh}}{\sigma; \Sigma, b{\sim}\tau@\rho'; \texttt{poll}\ e \Downarrow_\Delta^{(a,\rho)} \sigma; \Sigma, b{\sim}\tau@\rho'; \texttt{r} \cdot \langle\rangle; (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \emptyset, \emptyset)}$$

C-RET

$$\frac{e \Downarrow_\Delta v; \vec{u}}{\sigma; \Sigma; \texttt{ret}\ e \Downarrow_\Delta^{(a,\rho)} \sigma; \Sigma; v; (a \underset{\rho}{\hookrightarrow} (0, \vec{u}), \emptyset, \emptyset, \emptyset)}$$

Figure 5.2: Cost semantics for commands

CT-THREAD

$$\frac{\sigma; \Sigma; m \Downarrow_\Delta^{(a,\rho)} \sigma'; \Sigma'; v; g}{\sigma, a \hookrightarrow (v, \Sigma'); \Sigma; a \underset{\rho}{\hookrightarrow} (\delta, m) \Downarrow_\Delta \sigma'; g \downharpoonright_\delta^a}$$

CT-EXTEND

$$\frac{\sigma; \Sigma, \Sigma'; \mu \Downarrow_\Delta \sigma'; g}{\sigma; \Sigma; \nu\Sigma'\{\mu\} \Downarrow_\Delta \sigma'; g}$$

CT-CONCAT

$$\frac{\sigma; \Sigma; \mu \Downarrow_\Delta \sigma_1; (\mathcal{T}, E_s, E_j, E_w) \qquad \sigma; \Sigma; \mu' \Downarrow_\Delta \sigma_2; (\mathcal{T}', E'_s, E'_j, E'_w)}{\sigma; \Sigma; \mu \uplus \mu' \Downarrow_\Delta \sigma_1, \sigma_2; (\mathcal{T} \uplus \mathcal{T}', E_s \cup E'_s, E_j \cup E'_j, E_w \cup E'_w)}$$

Figure 5.3: Cost semantics for thread pools

ambient thread signature before (Σ) and after (Σ') evaluation of the command. In addition, it includes a *thread record* σ (and σ'). The thread record maps a thread name a to a pair (v_a, Σ_a) of the value to which thread a evaluates, and a signature containing threads that are (transitively) spawned by a. The thread record is used by the rules C-SYNC and C-POLL-SOME to capture the value of the target thread b, which must be returned by the sync operation (or a successful poll operation). The rules also capture the signature of threads transitively spawned by b, which they add to the signature, indicating that future operations in thread a now "know about" these threads. In showing the consistency of the cost semantics later in the chapter, we will use the judgment $\vdash^R_\Sigma \sigma$ to indicate that the values in σ are well-typed. The following rules apply to the judgment:

$$\frac{}{\vdash^R_\Sigma \cdot} \qquad \frac{\cdot \vdash^R_{\Sigma, a\sim\tau@\rho, \Sigma'} v : \tau \qquad \vdash^R_{\Sigma, a\sim\tau@\rho} \sigma}{\vdash^R_{\Sigma, a\sim\tau@\rho} \sigma, a \hookrightarrow (v, \Sigma')}$$

The other rules are more straightforward. Rule C-BIND composes the graphs generated by the subexpressions using the sequential composition operation defined as follows:

$$\begin{gathered}(a \underset{\rho}{\hookrightarrow} (0, \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w) \oplus_a (a \underset{\rho}{\hookrightarrow} (0, \vec{u}') \uplus \mathcal{T}', E'_s, E'_j, E'_w) \\ \triangleq \\ (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot \vec{u}') \uplus \mathcal{T} \uplus \mathcal{T}', E_s \cup E'_s, E_j \cup E'_j, E_w \cup E'_w)\end{gathered}$$

We use the notation $[\vec{u}]$ to indicate a graph consisting of a single thread. The name and priority of the thread will generally be evident from context, e.g., because $[\vec{u}]$ is immediately sequentially composed with another graph at thread a, so

$$[\vec{u}] \oplus_a (a \underset{\rho}{\hookrightarrow} (0, \vec{u}') \uplus \mathcal{T}, E_s, E_j, E_w) \triangleq (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot \vec{u}') \uplus \mathcal{T}, E_s, E_j, E_w)$$

Rule C-SPAWN evaluates the newly spawned thread to produce its cost graph, and then adds it to the graph along with a single vertex u which performs the spawn and the appropriate spawn edge. In contrast to C-SYNC and C-POLL-SOME, rule C-POLL-NONE does not add an edge from b: since the poll failed, there is no dependence relationship between thread b and vertex u.

Finally, the judgment $\sigma; \Sigma; \mu \Downarrow_\Delta \sigma'; g$ evaluates the thread pool μ to a graph g. The judgment includes the ambient thread record σ and signature Σ so that when evaluating one thread, we have access to the records of the other active threads. A thread pool with a single thread $a \underset{\rho}{\hookrightarrow} (\delta, m)$ evaluates to the same graph as the command m, but the graph may be "delayed" if the thread is not ready. The delay operator $g \lfloor^a_\delta$ indicates that the thread a of a graph g should be delayed by δ.

$$(a \underset{\rho}{\hookrightarrow} (\delta', \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w) \lfloor^a_\delta \triangleq (a \underset{\rho}{\hookrightarrow} (\delta + \delta', \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w)$$

Rule CT-CONCAT evaluates both parts of the thread pool and composes the graphs.

Lemma 2 shows that the evaluation judgment on expressions preserves typing. The equivalent property for commands will be shown as part of Lemma 6.

Lemma 2. *If* $\cdot \vdash^R_\Sigma e : \tau$ *and* $e \Downarrow_\Delta v; \vec{u}$*, then* $\cdot \vdash^R_\Sigma v : \tau$.

Proof. By induction on the derivation of $e \Downarrow_\Delta v; \vec{u}$. □

One more technical result we will need in Section 5.2 is that entries in the thread record for threads that don't appear in a command or thread pool are unnecessary for the purposes of the cost semantics.

Lemma 3. *1. If* $\cdot \vdash^R_\Sigma m \not\sim \tau \mathbin{@} \rho$ *and* $\sigma, c \hookrightarrow (v_c, \Sigma_c); \Sigma; m \Downarrow^{(a,\rho)}_\Delta \sigma', c \hookrightarrow (v_c, \Sigma_c); \Sigma'; v; g$ *and* $c \notin dom(\Sigma)$*, then* $\sigma; \Sigma; m \Downarrow^{(a,\rho)}_\Delta \sigma'; \Sigma'; v; g$*.*

2. If $\vdash^R_\Sigma \mu : \Sigma'$ *and* $\sigma, c \hookrightarrow (v_c, \Sigma_c); \Sigma; \mu \Downarrow_\Delta \sigma', c \hookrightarrow (v_c, \Sigma_c); g$ *and* $c \notin dom(\Sigma)$*, then* $\sigma; \Sigma; \mu \Downarrow_\Delta \sigma'; g$*.*

Proof. 1. By induction on the derivation of $\sigma, c \hookrightarrow (v', \rho'); \Sigma; m \Downarrow^{(a,\rho)}_\Delta \sigma', c \hookrightarrow (v', \rho'); \Sigma'; v; g$. The interesting cases are C-SYNC and C-POLL-SOME. We show the case for C-SYNC.

C-SYNC

$$\frac{e \Downarrow_\Delta \texttt{tid}[b]; \vec{u} \qquad u \text{ fresh}}{\begin{array}{c}\sigma, b \hookrightarrow (v, \Sigma'), c \hookrightarrow (v', \Sigma''); \Sigma, b{\sim}\tau@\rho'; \texttt{sync}\ e \\ \Downarrow^{(a,\rho)}_\Delta \sigma, b \hookrightarrow (v, \Sigma'), c \hookrightarrow (v', \Sigma''); \Sigma, b{\sim}\tau@\rho', \Sigma'; v; (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \{(b, u)\}, \emptyset)\end{array}}$$

(1) $c \neq b$ $\qquad (c \notin dom(\Sigma, b{\sim}\tau@\rho'))$

(2) $\sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \texttt{sync}\ e$
$\Downarrow^{(a,\rho)}_\Delta \sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho', \Sigma'; v; (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \{(b, u)\}, \emptyset)$ (C-SYNC)

2. By induction on the derivation of $\sigma, b \hookrightarrow (v', \rho'); \Sigma; \mu \Downarrow_\Delta \sigma', b \hookrightarrow (v', \rho'); g$. All cases follow from induction.

□

We now show that well-typed programs produce well-formed cost graphs. In fact, the type system guarantees an even stronger property which will also be more convenient to prove. Intuitively, a DAG is *strongly well-formed* if 1) all join edges go from higher-priority threads to lower-priority threads and 2) if a path from u to u' starts with a spawn edge and ends with a join edge, there exists another path from u to u' that doesn't go through the spawn edge. In terms of programs, the second condition means that thread a can't sync on thread b if it doesn't "know about" thread b. Because λ^4 is purely functional, a can only know about b by being descended from the thread that spawned b.

Definition 4. A DAG $g = (\mathcal{T}, E_s, E_j, E_w)$ is *strongly well-formed* if for all $(a, u) \in E_j$, we have that

1. $a \underset{\rho_a}{\hookrightarrow} (\delta_a, \vec{u}), b \underset{\rho_b}{\hookrightarrow} (\delta_b, \vec{u}_1 \cdot u \cdot \vec{u}_2) \in \mathcal{T}$,
2. $\rho_b \preceq \rho_a$ and
3. If $(u', a) \in E_s$, then there exists a path from u' to u where the first edge is a continuation edge.

Lemma 4. *If g is strongly well-formed, then g is well-formed.*

Proof. Let $a \underset{\rho}{\hookrightarrow} (\delta_a, u_1 \cdot \ldots \cdot u_n) \in \mathcal{T}$ and let $u \sqsupseteq u_n$. We need to show that either $u \sqsupseteq u_1$ or $u \sqsupseteq^w u_n$ or $\rho \preceq Prio_g(u)$. Since the graph is finite and acyclic, we can proceed by well-founded induction on $\sqsupseteq$. If $u = u_n$, the result is clear. Otherwise, assume that for all u' such that $u \sqsupset u' \sqsupseteq u_n$, we have $u' \sqsupseteq u_1$ or $u' \sqsupseteq^w u_n$ or $\rho \preceq Prio_g(u')$. If $u' \sqsupseteq u_1$ for any such u',

then $u \sqsupseteq u_1$, and if $u' \sqsupseteq^w u_n$ for all such u', then $u \sqsupseteq^w u_n$, so there must be at least one such u' such that $\rho \preceq Prio_g(u')$. Let E be the set of outgoing edges of u which lead to u' such that $u' \sqsupseteq u_n$ and $\rho \preceq Prio_g(u')$. If any edge in E is a continuation or join edge, then we have $\rho \preceq Prio_g(u') \preceq Prio_g(u)$. If all are weak edges, then $u \sqsupseteq^w u_n$. This means that there must be a spawn edge $(u, b) \in E$, where u' is the first vertex of thread b. If there exists a corresponding join edge (b, u'') in the path, then by assumption there exists a path from u to u'' where the first edge is a continuation edge, but this is a contradiction because there were assumed to be no continuation edges in E. If no corresponding join edge (b, u'') is in the path, then u_n must be in b, so $u' = u_1$ and $u \sqsupseteq u_1$, also a contradiction. □

We maintain the invariant that if $b \in dom(\Sigma)$ when an operation corresponding to vertex u in thread a is typed, then the vertex that spawned b must be an ancestor of u. We say that a graph for which this invariant holds is *compatible* with Σ at a.

Definition 5. We say that a graph $g = (\mathcal{T}, E_s, E_j, E_w)$ is compatible with a signature Σ at a if

1. $a \underset{\rho_a}{\hookrightarrow} (0, \vec{u}_a \cdot t_a) \in \mathcal{T}$
2. for all $b \in dom(\Sigma)$, if $(u, b) \in E_s$, then $u \sqsupseteq t_a$.

We say that a graph g is compatible with a thread record σ if for all $b \hookrightarrow (v, \Sigma') \in \sigma$, it is the case that g is compatible with Σ' at b.

We show some facts about compatibility that will be useful later:

Lemma 5. *1. If g is compatible with a signature Σ at a, then $g \oplus_a \vec{u}$ is compatible with Σ at a and $\vec{u} \oplus_a g$ is compatible with Σ at a.*

2. If g is compatible with σ, then $g \oplus_a \vec{u}$ is compatible with σ and $\vec{u} \oplus_a g$ is compatible with σ.

3. If g_1 and g_2 are compatible with Σ at a, then $g_1 \oplus_a g_2$ is compatible with Σ at a.

4. If g_1 and g_2 are compatible with σ, then $g_1 \oplus_a g_2$ is compatible with σ.

5. If g is strongly well-formed, then $g \oplus_a \vec{u}$ is strongly well-formed and $\vec{u} \oplus_a g$ is strongly well-formed.

Proof. 1. Part (1) of compatibility is immediate from the definitions, as is part (2) for $\vec{u} \oplus_a g$. To show part (2) on $g \oplus_a \vec{u}$, let $b \in dom(\Sigma)$, suppose $(u, b) \in E_s$. By definition, $u \sqsupseteq t_a$, where t_a is the last vertex of a in g. We have $t_a \sqsupseteq t'_a$ where t'_a is the last vertex of a in $\vec{u}$, completing the proof.

2. Composing at a doesn't change the structure of any other thread, so compatibility with σ is preserved.

3. Let $g_1 = (\mathcal{T}_1, E_s, E_j, E_w)$ and $g_2 = (\mathcal{T}_2, E'_s, E'_j, E'_w)$, where t_1 is the last vertex of a in g_1 and t_2 is the last vertex of a in g_2. Part (1) of compatibility is immediate from the definitions. For part (2), let $b \in dom(\Sigma)$ and suppose $(u, b) \in E_s$. Then $u \sqsupseteq t_1 \sqsupseteq t_2$. Now suppose $(u, b) \in E'_s$. Then $u \sqsupseteq t_2$ immediately.

4. Composing at a doesn't change the structure of any other thread, so compatibility with σ is preserved.

5. No join edges are added in either case, so strong well-formedness is preserved by composition.

□

Compatibility gives the final piece needed to show that a graph is strongly well-formed: if a vertex u syncs on a thread b, then b must be in the signature Σ used to type the sync operation u, and if the graph generated up to this point is compatible with Σ, the vertex that spawned b is an ancestor of u. At first glance, the phrase "the graph generated up to this point" seems terribly non-compositional. This would be worrisome, as we wish to be able to prove a large graph well-formed by breaking it into subgraphs and showing the result by induction. To do so, we strengthen the induction hypothesis by positing the existence of a graph g' which is well-formed and compatible with the current signature and thread record. This graph represents "the graph generated up to this point" and will be filled in appropriately when we compose subgraphs.

Lemma 6. *If* $\cdot \vdash^R_\Sigma m \mathrel{\dot\sim} \tau \mathbin{@} \rho$ *and* $\vdash^R_\Sigma \sigma$ *and* $\sigma; \Sigma; m \Downarrow^{(a,\rho)}_\Delta \sigma'; \Sigma'; v; g$ *and there exists some* g' *such that:*

1. g' *is strongly well-formed*
2. g' *is compatible with* Σ *at* a *and*
3. g' *is compatible with* σ

then

1. $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w)$
2. Σ' *extends* Σ
3. $g' \oplus_a g$ *is strongly well-formed*
4. $g' \oplus_a g$ *is compatible with* Σ' *at* a.
5. $g' \oplus_a g$ *is compatible with* σ'.
6. $\cdot \vdash^R_{\Sigma'} v : \tau$.

Proof. By induction on the derivation of $\sigma; \Sigma; m \Downarrow^{(a,\rho)}_\Delta \sigma'; \Sigma'; v; g$.

Case

C-BIND

$$\frac{\begin{array}{c} e \Downarrow_\Delta \mathtt{cmd}[\rho]\ \{m_1\}; \vec{u}_1 \\ \sigma; \Sigma; m_1 \Downarrow^{(a,\rho)}_\Delta \sigma_1; \Sigma_1; v; g_1 \qquad u \text{ fresh} \qquad \sigma_1; \Sigma_1; [v/x]m_2 \Downarrow^{(a,\rho)}_\Delta \sigma_2; \Sigma_2; v'; g_2 \end{array}}{\sigma; \Sigma; x \leftarrow e; m_2 \Downarrow^{(a,\rho)}_\Delta \sigma_2; \Sigma_2; v'; [\vec{u}_1] \oplus_a g_1 \oplus_a [u] \oplus_a g_2}$$

This case follows from the inductive hypothesis applied to the second subderivation if we can show that the conditions of the lemma hold for $g' \oplus_a [\vec{u}_1] \oplus_a g_1 \oplus_a [u]$. These in turn hold from Lemma 5 and the inductive hypothesis applied to the first subderivation if we can show that the conditions of the lemma hold for $g' \oplus_a [\vec{u}_1]$. This follows from Lemma 5 and the assumptions.

Case

C-SPAWN

$$\frac{b \text{ fresh} \qquad \sigma; \Sigma; m \Downarrow^{(b,\rho)}_\Delta \sigma, \sigma'; \Sigma, \Sigma'; v; (\mathcal{T}, E_s, E_j, E_w) \qquad u \text{ fresh}}{\begin{array}{c} \sigma; \Sigma; \mathtt{spawn}[\rho'; \tau]\ \{m\} \\ \Downarrow^{(a,\rho)}_\Delta \sigma, \sigma', b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \mathtt{tid}[b]; (a \underset{\rho}{\hookrightarrow} (0, u) \uplus \mathcal{T}, E_s \cup \{(u, b)\}, E_j, E_w) \end{array}}$$

Then $g = (a \underset{\rho}{\hookrightarrow} (0, u) \uplus \mathcal{T}, E_s \cup \{(u, b)\}, E_j, E_w)$. By induction, Σ, Σ' extends Σ and $g' \oplus_a (\mathcal{T}, E_s, E_j, E_w)$ is strongly well-formed and compatible with Σ, Σ' at b and is compatible

with σ, σ' and $\cdot \vdash^R_{\Sigma,\Sigma'} v : \tau$. All of these properties hold for $g' \oplus_a g$ as well, because this adds no join edges, nor does it change the structure of thread b. Because $g' \oplus_a g$ is compatible with Σ, Σ' at b and is compatible with σ, σ', we have that $g' \oplus_a g$ is compatible with $\sigma, \sigma', b \hookrightarrow (v, \Sigma')$. It remains to show that $g' \oplus_a g$ is compatible with $\Sigma, b{\sim}\tau@\rho$ at a. This is the case because $g' \oplus_a g$ is compatible with Σ at a and $(u, b) \in E_s \cup \{(u, b)\}$.

Case

C-SYNC

$$\frac{e \Downarrow_\Delta \texttt{tid}[b]; \vec{u} \qquad u \text{ fresh}}{\begin{array}{c}\sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \texttt{sync}\ e \\ \Downarrow^{(a,\rho)}_\Delta \sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho', \Sigma'; v; (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \{(b, u)\}, \emptyset)\end{array}}$$

Then $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u}{\cdot}[u]), \emptyset, \{(b, u)\}, \emptyset)$. It is clear that $\Sigma, b{\sim}\tau@\rho', \Sigma'$ extends $\Sigma, b{\sim}\tau@\rho'$. The only join edge added to form $g' \oplus_a g$ is (b, u). By inversion on the typing rule, we must have $\rho \preceq \rho'$. Because g' is compatible with $\Sigma, b{\sim}\tau@\rho'$ at a, if $(u', b) \in g'$ then $u' \sqsupseteq u$, so $g' \oplus_a g$ is strongly well-formed. In addition, it remains compatible with $\sigma, b \hookrightarrow (v, \Sigma')$. By inversion on $\vdash^R_{\Sigma, b{\sim}\tau@\rho'} \sigma, b \hookrightarrow (v, \Sigma')$, we must have $\cdot \vdash^R_{\Sigma, b{\sim}\tau@\rho', \Sigma'} v : \tau$. It remains to show that $g' \oplus_a g$ is compatible with $\Sigma, b{\sim}\tau@\rho', \Sigma'$ at a and in particular that for all $c \in dom(\Sigma')$, if $(u'', c) \in g' \oplus_a g$, then $u'' \sqsupseteq u$. Because g' is compatible with $\sigma, b \hookrightarrow (v, \Sigma')$, we have that g' is compatible with Σ' at b, so u'' is an ancestor of the last vertex of b in g' and is therefore an ancestor of u in $g' \oplus_a g$.

Case

C-POLL-SOME

$$\frac{u \text{ fresh}}{\begin{array}{c}\sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \texttt{poll}\ \texttt{tid}[b] \\ \Downarrow^{(a,\rho)}_\Delta \sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho', \Sigma'; \texttt{1} \cdot v; (a \underset{\rho}{\hookrightarrow} (0, u), \emptyset, \emptyset, \{(b, u)\})\end{array}}$$

Then $g = (a \underset{\rho}{\hookrightarrow} (0, u), \emptyset, \emptyset, \{(b, u)\})$. It is clear that $\Sigma, b{\sim}\tau@\rho', \Sigma'$ extends $\Sigma, b{\sim}\tau@\rho'$. No join edges are added to form $g' \oplus_a g$, so $g' \oplus_a g$ is strongly well-formed. In addition $g' \oplus_a g$ remains compatible with $\sigma, b \hookrightarrow (v, \Sigma')$. By inversion on $\vdash^R_{\Sigma, b{\sim}\tau@\rho'} \sigma, b \hookrightarrow (v, \Sigma')$, we must have $\cdot \vdash^R_{\Sigma, b{\sim}\tau@\rho', \Sigma'} v : \tau$. It remains to show that $g' \oplus_a g$ is compatible with $\Sigma, b{\sim}\tau@\rho', \Sigma'$ at a and in particular that for all $c \in dom(\Sigma')$, if $(u'', c) \in g' \oplus_a g$, then $u'' \sqsupseteq u$. Because g' is compatible with $\sigma, b \hookrightarrow (v, \Sigma')$, we have that g' is compatible with Σ' at b, so u'' is an ancestor of the last vertex of b in g' and is therefore an ancestor of u in $g' \oplus_a g$.

Case

C-POLL-NONE

$$\frac{u \text{ fresh}}{\sigma; \Sigma, b{\sim}\tau@\rho'; \texttt{poll}\ \texttt{tid}[b] \Downarrow^{(a,\rho)}_\Delta \sigma; \Sigma, b{\sim}\tau@\rho'; \texttt{r} \cdot \langle\rangle; (a \underset{\rho}{\hookrightarrow} (0, u), \emptyset, \emptyset, \emptyset)}$$

By rules UNITI and $+I_2$, we have $\cdot \vdash^R_{\Sigma, b{\sim}\tau@\rho'} v : \tau + \texttt{unit}$. The other conditions follow from Lemma 5.

Case

C-RET

$$\frac{e \Downarrow_\Delta v; \vec{u}}{\sigma; \Sigma; \texttt{ret}\ e \Downarrow_\Delta^{(a,\rho)} \sigma; \Sigma; v; (a \underset{\rho}{\hookrightarrow} (0, \vec{u}), \emptyset, \emptyset, \emptyset)}$$

By Lemma 2, we have $\cdot \vdash_\Sigma^R v : \tau$. The other conditions follow from Lemma 5.

□

In order to show that a full graph generated by a well-typed program is strongly well-formed, we simply observe that "the graph generated up to this point" is empty, and trivially satisfies the requirements of the lemma.

Corollary 2. *If* $\cdot \vdash_\cdot^R m \div \tau \mathbin{@} \rho$ *and* $\cdot; \cdot; m \Downarrow_\Delta^{(a,\rho)} \sigma; \Sigma; v; g$*, then* g *is well-formed.*

Proof. Because $\emptyset$ is strongly well-formed and compatible with $\cdot$, Lemma 6 shows that g is strongly well-formed, and is thus well-formed by Lemma 4. □

5.2 Response Time Bound for Operational Semantics

Thus far in this chapter, we have developed a DAG-based cost model for λ^4 programs and showed an offline scheduling bound which holds for DAGs derived from well-typed λ^4 programs. Although the DAGs are built upon our intuitions of how λ^4 programs execute, they are still abstract artifacts which must, in order to be valuable, be shown to correspond to more concrete, runtime notions.

Our goal in this section is to show that an execution of a λ^4 program using the dynamic semantics corresponds to a valid schedule of the DAG generated from that program. Because well-formed DAGs admit the cost bound of Theorem 2, we may then directly appeal to that theorem for cost bounds on programs. The argument proceeds as follows:

1. Lemmas 7 and 8 show that a thread of a DAG is ready (i.e., its first unexecuted vertex is ready) if and only if the corresponding thread in the program may take a step.
2. Lemmas 9 and 10 are used to show that the schedule corresponding to a λ^4 execution is admissible.
3. Lemma 11 shows that stepping some set of threads in the dynamic semantics corresponds to executing the first vertex of those threads in a schedule of the DAG.
4. Lemma 12 combines the above results to establish a correspondence between an execution of a λ^4 program and a schedule of a cost graph that can be produced by the program.
5. Finally, we use Theorem 2 to bound the length of the schedule and therefore the length of the execution in the dynamic semantics.

The correspondence between ready DAG threads and active thread pool threads requires intermediate results about expressions and commands. Part (1) of Lemma 7 states that an expression produces an empty thread if and only if it is a value. Part (2) states that a command a) takes a silent step if and only if it produces a graph with an enabled first vertex, b) returns a value if and only if it produces an empty graph and c) takes a sync step if and only if it produces a

graph with an incoming join edge. Note that part (2) does not require that the delay on the thread be 0, as commands of delayed threads are still capable of stepping in the transition semantics, but are simply prohibited from doing so because of the delay. Parts (3) and (4) extend part (2) to thread pools, and require that thread delays be 0 in order for the thread to take a step. Part (4) in particular states that if the first vertex of a thread is ready in a graph, the corresponding thread in the thread pool can take a silent step. The key observation in proving part (4) from part (2) is that if a vertex u has an incoming join edge (b, u) but thread b is empty, then thread b must be returning a value and u can perform the sync, taking a silent step with rule D-SYNC.

Recall from Section 4.2.2 that thread pools step until all `poll` operations have "committed" to whether they will be successful or unsuccessful. Lemma 7, and several other results in this section, require preconditions involving the `polled` judgment, defined earlier, that captures this property that a command or thread pool has fully committed.

Lemma 7. *1. If $\cdot \vdash^R_\Sigma e : \tau$ and $e \Downarrow_\Delta v; \vec{u}$, then $e \to^\Delta_\Sigma (\delta, e')$ for some e', δ if and only if $\vec{u}$ is nonempty.*

2. *If m polled and $\cdot \vdash^R_\Sigma m \simeq \tau \mathbin{@} \rho$ and $\sigma; \Sigma; m \Downarrow^{(a,\rho)}_\Delta \sigma'; \Sigma'; v; g$, then $g = (a \underset{\rho}{\hookrightarrow} (\delta, \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w)$, and g has no spawn edges to threads in Σ and has no join or weak edges to active threads other than a, and one of the following is true:*
   (a) *There exists m' such that $m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta', \Sigma, m', \mu)$ and $\vec{u} = u \cdot_{\delta'} \vec{u}'$.*
   (b) *There exists v such that v $\texttt{val}_\Sigma$ and $m = \texttt{ret}\ v$ and $\vec{u} = []$.*
   (c) *There exist v and m' such that $m \overset{v \triangleleft b}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta', \Sigma, m', \mu)$ and $\vec{u} = u \cdot_{\delta'} \vec{u}'$ and there exists an edge $(b, u) \in g$, which is the only in-edge of u.*
3. *If μ polled and $\vdash^R_\Sigma \mu : \Sigma', a \sim \tau @ \rho$ and $\sigma; \Sigma; \mu \Downarrow_\Delta \sigma'; g$ where $g = (\mathcal{T}, E_s, E_j, E_w)$ and $\mu \overset{a/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^\Delta \mu'$, then g has no spawn, join or weak edges to threads not in $\mathcal{T}$ and*
   (a) *If $\alpha = \epsilon$, then $a \underset{\rho}{\hookrightarrow} (0, u \cdot_\delta \vec{u}) \in \mathcal{T}$ and u is ready in g.*
   (b) *If $\alpha = v \triangleright$, then $a \underset{\rho}{\hookrightarrow} (0, []) \in \mathcal{T}$.*
   (c) *If $\alpha = v \triangleleft b$, then $a \underset{\rho}{\hookrightarrow} (0, u \cdot_\delta \vec{u}) \in \mathcal{T}$ and there exists an edge $(b, u) \in g$, which is the only in-edge of u.*
4. *If $\vdash^R_\cdot \mu : \Sigma$ and $\sigma; \Sigma; \mu \Downarrow_\Delta \sigma'; g$ and the first vertex of a is ready in g, then there exists μ' such that $\mu \overset{a/\epsilon}{\underset{\cdot}{\Longrightarrow}}{}^\Delta \mu'$.*

Proof. 1. By Theorem 4, either $e \to^\Delta_\Sigma e'$ or e $\texttt{val}_\Sigma$. It remains to show that e $\texttt{val}_\Sigma$ if and only if $\vec{u} = []$. Both directions are clear by inspection of the cost semantics.

2. By Theorem 4 and inspection of the dynamic semantics, the three cases given are exhaustive. If $m = \texttt{ret}\ v$, then apply C-VAL and C-RET. Otherwise, proceed by induction on the derivation of $m \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta, \Sigma', m', \mu')$. There are no cases for rules D-POLL2A and D-POLL2B because of the assumption m polled.

Case

$$\text{D-BIND1}\quad \frac{e \to_{\Sigma}^{\Delta} (\delta, e')}{x \leftarrow e; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, x \leftarrow e'; m, \emptyset)}$$

(1) $e \Downarrow_{\Delta} v'; \vec{u}',\ \vec{u}'$ nonempty (inversion on C-BIND,part 1)
(2) $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u}' \cdot \vec{u}'') \uplus \mathcal{T}, E_s, E_j, E_w)$ (inversion on C-BIND)

Case

$$\text{D-BIND2}\quad \frac{m_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \Sigma', m_1', \mu')}{x \leftarrow \texttt{cmd}[\rho]\ \{m_1\}; m_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \Sigma', x \leftarrow \texttt{cmd}[\rho]\ \{m_1'\}; m_2, \mu')}$$

(1) $\sigma; \Sigma; m_1 \Downarrow_{\Delta}^{(a,\rho)} \sigma'; \Sigma''; v; g_1$ (inversion on cost semantics)
(2) g_1, α meet condition 2(a) or 2(c) of the lemma (induction)
(3) g, α meet condition 2(a) or 2(c) of the lemma (C-BIND)

Case

$$\text{D-BIND3}\quad \frac{e\ \texttt{val}_{\Sigma}}{x \leftarrow \texttt{cmd}[\rho]\ \{\texttt{ret}\ e\}; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (0, \cdot, [e/x]m, \emptyset)}$$

(1) $g = (a \underset{\rho}{\hookrightarrow} (0, u \cdot \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w)$ (inversion on C-BIND)

Case

$$\text{D-SPAWN}\quad \frac{b \text{ fresh}}{\texttt{spawn}[\rho; \tau]\ \{m\} \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (0, b{\sim}\tau@\rho, \texttt{ret}\ \texttt{tid}[b], b \underset{\rho}{\hookrightarrow} (0, m))}$$

(1) $g = (a \underset{\rho}{\hookrightarrow} (0, u) \uplus \mathcal{T}, E_s \cup \{(u, b)\}, E_j, E_w)$ (inversion on C-SPAWN)
(2) $b \notin \Sigma$, so no spawn edges to threads in Σ are added (b fresh)

Case

$$\text{D-SYNC1}\quad \frac{e \to_{\Sigma}^{\Delta} (\delta, e')}{\texttt{sync}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, \texttt{sync}\ e', \emptyset)}$$

(1) $e \Downarrow_{\Delta} v'; \vec{u}',\ \vec{u}'$ nonempty (inversion on C-SYNC,part 1)
(2) $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u}' \cdot \vec{u}'') \uplus \mathcal{T}, E_s, E_j, E_w)$ (inversion on C-SYNC)

Case

$$\text{D-SYNC2}\quad \frac{v\ \texttt{val}_{\Sigma}}{\texttt{sync}\ (\texttt{tid}[b]) \overset{v \lhd b}{\underset{\Sigma}{\mapsto}}^{\rho} (0, \cdot, \texttt{ret}\ v, \emptyset)}$$

(1) $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u}' \cdot u), \emptyset, \{(b, u)\}, \emptyset)$ (inversion on C-SYNC)
(2) $v \Downarrow_\Delta v; \vec{u}'$ (inversion on C-SYNC)
(3) $\vec{u}' = []$ (part 1)

Case

$$\text{D-POLL1}\ \frac{e \rightarrow_\Sigma^\Delta (\delta, e')}{\texttt{poll}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, \texttt{poll}\ e', \emptyset)}$$

(1) $e \Downarrow_\Delta v'; \vec{u}', \vec{u}'$ nonempty (inversion on C-POLL, part 1)
(2) $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u}' \cdot u), \emptyset, \emptyset, E_w)$ (inversion on C-POLL-SOME, C-POLL-NONE)

Case

$$\text{D-RET}\ \frac{e \rightarrow_\Sigma^\Delta (\delta, e')}{\texttt{ret}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, \texttt{ret}\ e', \emptyset)}$$

(1) $e \Downarrow_\Delta v; \vec{u}', \vec{u}'$ nonempty (inversion on C-RET, part 1)
(2) $(a \underset{\rho}{\hookrightarrow} (0, \vec{u}'), \emptyset, \emptyset, \emptyset)$ (inversion on C-RET)

3. By induction on the derivation of $\mu \overset{a/\alpha}{\underset{\Sigma}{\Longrightarrow}}^{\Delta} \mu'$.

Case

$$\text{DT-THREAD}\ \frac{m \overset{\alpha}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \Sigma', m', \mu')}{a \underset{\rho}{\hookrightarrow} (0, m) \overset{a/\alpha}{\underset{a \sim \tau @ \rho, \Sigma}{\Longrightarrow}}^{\Delta} \nu \Sigma' \{a \underset{\rho}{\hookrightarrow} (\delta, m') \uplus \mu'\}}$$

(1) $\sigma; \Sigma; m \Downarrow_\Delta^{(a,\rho)} \sigma'; \Sigma'; v; g$ (inversion on CT-THREAD)
(2) $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w)$ (part 2)
(3) condition (a) or (c) holds on g (part 2)

Case

$$\text{DT-RET}\ \frac{v\ \texttt{val}_{a \sim \tau @ \rho, \Sigma}}{a \underset{\rho}{\hookrightarrow} (\delta, \texttt{ret}\ v) \overset{a/v\triangleright}{\underset{a \sim \tau @ \rho, \Sigma}{\Longrightarrow}}^{\Delta} a \underset{\rho}{\hookrightarrow} (\delta, \texttt{ret}\ v)}$$

(1) $\sigma; \Sigma; \texttt{ret}\ v \Downarrow_\Delta^{(a,\rho)} \sigma'; \Sigma'; v; g$ (inversion on CT-RET)
(2) $g = (a \underset{\rho}{\hookrightarrow} (0, []), \emptyset, \emptyset, \emptyset)$ (part 2)

Case

$$\text{DT-SYNC}\ \frac{\Sigma = \Sigma', a \sim \tau_a @ \rho_a, b \sim \tau_b @ \rho_b \qquad \mu_1 \overset{a/v \triangleleft b}{\underset{\Sigma}{\Longrightarrow}}^{\Delta} \mu_1' \qquad \mu_2 \overset{b/v\triangleright}{\underset{\Sigma}{\Longrightarrow}}^{\Delta} \mu_2}{\mu_1 \uplus \mu_2 \overset{a/\epsilon}{\underset{\Sigma}{\Longrightarrow}}^{\Delta} \mu_1' \uplus \mu_2}$$

(1) $\sigma, \sigma_2; \Sigma; \mu_1 \Downarrow_\Delta \sigma_1; (a \underset{\rho_a}{\hookrightarrow} (0, u \cdot_\delta \vec{u}_a) \uplus \mathcal{T}, E_s, E_j, E_w)$

(inversion on CT-CONCAT, induction)

(2) $\sigma, \sigma_1; \Sigma; \mu_2 \Downarrow_\Delta \sigma_2; (b \underset{\rho_b}{\hookrightarrow} (\delta, []) \uplus \mathcal{T}', E'_s, E'_j, E'_w)$

(inversion on CT-CONCAT, induction)

(3) $g = (a \underset{\rho_a}{\hookrightarrow} (0, u \cdot_\delta \vec{u}_a) \uplus b \underset{\rho_b}{\hookrightarrow} (\delta, []) \uplus \mathcal{T} \uplus \mathcal{T}', E_s \cup E'_s, E_j \cup E'_j, E_w \cup E'_w)$

(CT-CONCAT)

(4) no edges in E'_s, E'_j, E'_w target u (induction)

(5) u is ready in g (b is empty, so we may ignore the edge (b, u))

Case

$$\text{DT-CONCAT}\ \frac{\mu_1 \overset{a/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu'_1}{\mu_1 \uplus \mu_2 \overset{a/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu'_1 \uplus \mu_2}$$

(1) $\sigma, \sigma_2; \Sigma; \mu_1 \Downarrow_\Delta \sigma_1; g_1, g_1 = (a \underset{\rho_a}{\hookrightarrow} (0, \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w)$ (induction)

(2) $\sigma, \sigma_1; \Sigma; \mu_2 \Downarrow_\Delta \sigma_2; (\mathcal{T}', E'_s, E'_j, E'_w)$ (induction)

(3) $g = (a \underset{\rho_a}{\hookrightarrow} (0, \vec{u}) \uplus \mathcal{T} \uplus \mathcal{T}', E_s \cup E'_s, E_j \cup E'_j, E_w \cup E'_w)$ (CT-CONCAT)

Subcase: $\alpha = \epsilon$

(a) $\vec{u} = u \cdot_\delta \vec{u}', u$ is ready in g_1 (induction)

(b) no edges in E'_s, E'_j, E'_w target u (induction)

(c) u is ready in g

Subcase: $\alpha = v\triangleright$

(a) $\vec{u} = []$ (induction)

Subcase: $\alpha = v \triangleleft b$

(a) $\vec{u} = u \cdot_\delta \vec{u}', \exists (b, u)$ the only in-edge of u in g_1 (induction)

(b) no edges in E'_s, E'_j, E'_w target u (induction)

(c) (b, u) is the only in-edge of u in g

Case

$$\text{DT-EXTEND}\ \frac{\mu \overset{b/\alpha}{\underset{\Sigma, a{\sim}\tau@\rho}{\Longrightarrow}}{}^{\Delta} \mu'}{\nu a{\sim}\tau@\rho\{\mu\} \overset{b/\alpha}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \nu a{\sim}\tau@\rho\{\mu'\}}$$

(1) $\sigma; \Sigma, a{\sim}\tau@\rho; \mu \Downarrow_\Delta \sigma'; g$ (inversion on CT-EXTEND)

(2) g meets the conditions of the lemma (induction)

4.

(1) $\mu \equiv \nu\Sigma'\{a \underset{\rho}{\hookrightarrow} (0, m) \uplus \mu_0\}$, where $dom(\mu_0) \cup \{a\} = dom(\Sigma')$ (Theorem 4)

(2) $\mu \overset{a/\alpha}{\underset{\Sigma,\Sigma'}{\Longrightarrow}}{}^{\Delta} \mu'$ and $\vdash^R_{\Sigma,\Sigma'} \alpha$ action (Theorem 4)

(3) $\alpha \neq v \triangleright$ (part 3 would imply a has no vertices in g, a contradiction)

Subcase: $\alpha = \epsilon$

(a) Conclusion holds trivially

Subcase: $\alpha = v \triangleleft b$

(a) $(b, u) \in g$, where u is the first vertex of a and this is the lone in-edge of u (part 3)

(b) $b{\sim}\tau_b@\rho_b \in \Sigma, \Sigma'$ (inversion on the static semantics for actions)

(c) $b \underset{\rho_b}{\hookrightarrow} (0, m_b) \in \mu_0$ (assumption)

(d) b is empty in g (u is ready in g)

(e) $m_b = \texttt{ret}\ v$ (part 2)

(f) $\exists \mu''.\mu \overset{a/\epsilon}{\underset{\Sigma,\Sigma'}{\Longrightarrow}}{}^{\Delta} \mu''$ (DT-RET, DT-SYNC)

□

Parts (3) and (4) of Lemma 7 state that a thread can take a silent step if and only if its first vertex is ready in the corresponding graph. However, this result still considers only sequential execution: if threads a and b are both ready in the graph, it says nothing about whether a and b can step *in parallel*. Lemma 8 extends the result to parallel steps. It states that a set $a_1, \ldots, a_n$ of threads that are ready in g may all step simultaneously, and that any set of threads that can take a parallel step must be ready in g.

Lemma 8. *Let* $R = \{a \mid a \underset{\rho}{\hookrightarrow} (0, u \cdot_\delta \vec{u}) \in g, u \text{ is ready in } g\}$. *If* μ polled *and* $\vdash^R_\Sigma \mu : \Sigma_\mu$ *and* $\sigma; \Sigma; \mu \Downarrow_\Delta \sigma'; g$, *then*

1. *For any subset* $\{a_1, \ldots, a_n\}$ *of* R, *we have* $\mu \overset{\{a_1,\ldots,a_n\}}{\underset{P}{\Longrightarrow}}{}^{\Delta} \mu'$.
2. *If* $\mu \overset{\{a_1,\ldots,a_n\}}{\underset{P}{\Longrightarrow}}{}^{\Delta} \mu'$, *then* $\{a_1, \ldots, a_n\} \subset R$.

Proof. 1.

(1) $\mu \equiv \nu\Sigma'\{a_1 \underset{\rho_1}{\hookrightarrow} (0, m_1) \uplus \ldots \uplus a_m \underset{\rho_m}{\hookrightarrow} (0, m_m)\}$ (Theorem 4)

(2) $\forall\, a_i \in R.\mu \overset{a_i/\epsilon}{\underset{\Sigma}{\Longrightarrow}}{}^{\Delta} \mu'_i$ (Lemma 7)

(3) $\forall a_i.\mu'_i \equiv \nu\Sigma''\{a_1 \underset{\rho_1}{\hookrightarrow} (0, m_1) \uplus \ldots \uplus a_i \underset{\rho_i}{\hookrightarrow} (0, m'_i) \uplus \mu''_i \uplus \ldots \uplus a_m \underset{\rho_m}{\hookrightarrow} (0, m_m)\}$ (inspection of transition rules)

(4) $\mu \overset{\{a_1,\ldots,a_n\}}{\underset{P}{\Longrightarrow}}{}^{\Delta} \mu'$ (DT-PAR)

2. Let $i \in [1, n]$. By inversion on rule DT-PAR, $\mu \xrightarrow{a_i/\epsilon}_{\cdot}^{\Delta} \mu_i'$. By Lemma 7, $a_i \in R$.

□

Recall from Chapter 3 that a schedule is admissible if it never leaves a weak edge without a corresponding strong edge. Lemmas 9 and 10 will be used to show that the schedule corresponding to an execution of a λ^4 program is admissible. Intuitively, this is true because a weak edge targeting a ready vertex would correspond to an uncommitted `poll` operation, which is ruled out in the dynamics.

More formally, we define a *graph* $g = (\mathcal{T}, E_s, E_j, E_w)$ to be admissible if for all $u \in g$, there is a weak edge $(u_1, u) \in E_w$ only if there is a strong edge $(u_2, u) \in g$. A schedule is admissible if, at every step, the graph consisting of the unexecuted vertices is admissible. We show that if μ polled, then the graph corresponding to μ is admissible.

Lemma 9. *If $\sigma; \Sigma; m \Downarrow_{\Delta}^{(a,\rho)} \sigma'; \Sigma'; v; g$ and $g = (a \underset{\rho}{\hookrightarrow} (\delta, \vec{u}) \uplus \mathcal{T}, E_s, E_j, E_w)$ then every vertex of g has a strong incoming edge except possibly the first vertex of $\vec{u}$.*

Proof. By induction on the derivation of $\sigma; \Sigma; m \Downarrow_{\Delta}^{(a,\rho)} \sigma'; \Sigma'; v; g$. □

Remark 1. Because of Lemma 9, if $\sigma; \Sigma; m \Downarrow_{\Delta}^{(a,\rho)} \sigma'; \Sigma'; v; g$, then g is admissible if and only if the first vertex of thread a doesn't have a weak incoming edge.

Lemma 10. *1. If e polled and $e \Downarrow_{\Delta} v; \vec{u}$, then v polled or $\vec{u} \neq []$.*

2. If m polled and $\sigma; \Sigma; m \Downarrow_{\Delta}^{(a,\rho)} \sigma'; \Sigma'; v; g$, then g is admissible.

3. If μ polled and $\sigma; \Sigma; \mu \Downarrow_{\Delta} \sigma'; g$, then g is admissible.

Proof. 1. By induction on the derivation of $e \Downarrow_{\Delta} v; \vec{u}$.

2. By induction on the derivation of $\sigma; \Sigma; m \Downarrow_{\Delta}^{(a,\rho)} \sigma'; \Sigma'; v; g$. By the above Remark, all but two cases are trivial.

Case

$$\textsc{C-Bind}\ \frac{e \Downarrow_{\Delta} \mathtt{cmd}[\rho]\ \{m_1\}; \vec{u}_1 \qquad \sigma; \Sigma; m_1 \Downarrow_{\Delta}^{(a,\rho)} \sigma_1; \Sigma_1; v; g_1 \qquad u\ \mathsf{fresh} \qquad \sigma_1; \Sigma_1; [v/x]m_2 \Downarrow_{\Delta}^{(a,\rho)} \sigma_2; \Sigma_2; v'; g_2}{\sigma; \Sigma; x \leftarrow e; m_2 \Downarrow_{\Delta}^{(a,\rho)} \sigma_2; \Sigma_2; v'; [\vec{u}_1] \oplus_a g_1 \oplus_a [u] \oplus_a g_2}$$

(1) e polled (inversion)
(2) $\mathtt{cmd}[\rho]\ \{m_1\}$ polled or $\vec{u}_1 \neq []$ (part 1)

Subcase: $\mathtt{cmd}[\rho]\ \{m_1\}$ polled
(a) m_1 polled (inversion)
(b) g_1 admissible (induction)
(c) all vertices of g_2 have a strong incoming edge (Lemma 9)
(d) $[\vec{u}_1] \oplus_a g_1 \oplus_a [u] \oplus_a g_2$ admissible (Remark)
Subcase: $\vec{u}_1 \neq []$
(a) $[\vec{u}_1] \oplus_a g_1 \oplus_a [u] \oplus_a g_2$ admissible (Remark)

Case

C-POLL-SOME

$$\frac{e \Downarrow_\Delta \texttt{tid}[b]; \vec{u} \qquad u \text{ fresh}}{\begin{array}{c}\sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \texttt{poll}\ e \\ \Downarrow_\Delta^{(a,\rho)} \sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho', \Sigma'; \texttt{1} \cdot v; (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \emptyset, \{(b, u)\})\end{array}}$$

(1) e polled (inversion)
(2) $\vec{u} \neq []$ (part 1, since $\texttt{tid}[b]$ polled is not derivable)
(3) $(a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \emptyset, \{(b, u)\})$ admissible (Remark)

3. By induction on the derivation of $\sigma; \Sigma; \mu \Downarrow_\Delta \sigma'; g$

□

We now move on to showing that a parallel transition corresponds to a step of a schedule. At a more precise level, Lemma 11 shows that if a thread pool μ' produces a graph g' and μ steps to μ', then μ produces a graph isomorphic to g sequentially post-composed with one vertex for each thread that was stepped. Stating this formally requires us to define a new graph composition operator $\overline{\oplus}_a$ which composes a thread with a graph g by adding outgoing edges from the thread to *all* sources of g, with the edge to a being a continuation edge and all other edges being spawn edges (as opposed to $\oplus_a$ which adds an edge only to thread a).

$$\begin{array}{rl} & [\vec{u}]\overline{\oplus}_a(a \underset{\rho}{\hookrightarrow} (\delta, \vec{u}') \uplus a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, \vec{u}_1) \cdots \uplus a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, \vec{u}_n), E_s, E_j, E_w) \\ \triangleq & (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot_\delta \vec{u}') \uplus a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, \vec{u}_1) \cdots \uplus a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, \vec{u}_n), E_s \cup \{(u, a_1), \ldots, (u, a_n)\}, E_j, E_w) \end{array}$$

Lemma 11. *1. If $e' \Downarrow_\Delta v; \vec{u}$ and $e \to_\Sigma^\Delta (\delta, e')$, then $e \Downarrow_\Delta v; u \cdot_{\delta+1} \vec{u}$.*

2. If $\sigma; \Sigma; a \underset{\rho}{\hookrightarrow} (\delta, m') \uplus \mu' \Downarrow_\Delta \sigma''; g$ and m polled and $m \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta, \Sigma', m', \mu')$, then

$$\sigma; \Sigma; m \Downarrow_\Delta^{(a,\rho)} \sigma''; \Sigma''; v; g_0$$

where g_0 is isomorphic to $[u]\overline{\oplus}_a g$.

3. If $\sigma; \Sigma; a \underset{\rho}{\hookrightarrow} (\delta, m') \uplus \mu' \Downarrow_\Delta \sigma''; g$ and $m \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^\Delta (\delta, \Sigma', m', \mu')$ where $\alpha = \not{?} b$ or $\alpha = v\ ?\ b$, then

$$\sigma; \Sigma; m \Downarrow_\Delta^{(a,\rho)} \sigma''; \Sigma''; v; g_0$$

where g_0 is isomorphic to g.

4. If $\sigma; \Sigma; \mu' \Downarrow_\Delta \sigma'; g$ and $\mu \overset{\alpha}{\underset{\Sigma}{\Rightarrow}}{}^\Delta \mu'$ where $\alpha = ?$ or $\alpha = \not{?} b$ or $\alpha = v\ ?\ b$, then

$$\sigma; \Sigma; \mu' \Downarrow_\Delta \sigma'; g_0$$

where g_0 is isomorphic to g.

5. If $\sigma; \Sigma; \mu' \Downarrow_\Delta \sigma'; g'$ and μ polled and $\mu \overset{\{a_1/\epsilon, \ldots, a_n/\epsilon\}}{\underset{P}{\longrightarrow}}{}^\Delta \mu'$, then g' can be decomposed into $g_0 \uplus g_1 \uplus \cdots \uplus g_n$, $\mu \equiv \nu\Sigma\{\mu \uplus \mu_1 \uplus \ldots \uplus \mu_n\}$, and $\sigma; \Sigma; \mu \Downarrow_\Delta \sigma'; g$, where g is isomorphic to $g_0 \uplus ([u_1]\overline{\oplus}_{a_1} g_1) \uplus \cdots \uplus ([u_n]\overline{\oplus}_{a_n} g_n)$.

Proof. 1. By induction on the derivation of $e \to^{\Delta}_{\Sigma} (\delta, e')$. We present the interesting cases.

Case

$$\textsc{D-Let-Step}\ \frac{e_1 \to^{\Delta}_{\Sigma} (\delta, e_1')}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to^{\Delta}_{\Sigma} (\delta, \texttt{let}\ x = e_1'\ \texttt{in}\ e_2)}$$

(1) $e_1' \Downarrow_{\Delta} v_1; \vec{u}_1$ (inversion on C-Let)
(2) $[v_1/x]e_2 \Downarrow_{\Delta} v; \vec{u}_2, \vec{u} = \vec{u}_1 \cdot u \cdot \vec{u}_2$ (inversion on C-Let)
(3) $e_1 \Downarrow_{\Delta} v_1; u' \cdot_{\delta+1} \vec{u}_1$ (induction)
(4) $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow_{\Delta} v; u' \cdot_{\delta+1} \vec{u}_1 \cdot u \cdot \vec{u}_2$ (C-Let)

Case

$$\textsc{D-Input}\ \frac{\delta \in \Delta(d)}{\texttt{input}_d \to^{\Delta}_{\Sigma} (\delta - 1, \texttt{in})}$$

(1) $\vec{u} = u', v = \overline{n}$ for some $n \in \mathbb{N}$ (inversion on C-In)
(2) $\texttt{input}_d \Downarrow_{\Delta} \overline{n}; u \cdot_{\delta} u'$ (C-Input)

2. By induction on the derivation of $m \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\Sigma', m', \mu',)$.

Case

$$\textsc{D-Bind1}\ \frac{e \to^{\Delta}_{\Sigma} (\delta, e')}{x \leftarrow e; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \cdot, x \leftarrow e'; m, \emptyset)}$$

(1) $e' \Downarrow_{\Delta} \texttt{cmd}[\rho']\ \{m_1\}; \vec{u}_1$ (inversion on C-Bind)
(2) $\sigma; \Sigma; m_1 \Downarrow_{\Delta}^{(a,\rho)} \sigma_1; \Sigma_1; v_1; g_1$ (inversion on C-Bind)
(3) $\sigma_1; \Sigma_1; [v_1/x]m \Downarrow_{\Delta}^{(a,\rho)} \sigma''; \Sigma''; v; g_2$ (inversion on C-Bind)
(4) $\sigma; \Sigma; a \underset{\rho}{\hookrightarrow} (\delta, x \leftarrow e'; m) \Downarrow_{\Delta} \sigma''; [\vec{u}_1]\ \lfloor{}^{a}_{\delta} \oplus_a g_1 \oplus_a [u] \oplus_a g_2$
(inversion on CT-Concat, CT-Thread, C-Bind)
(5) $e \Downarrow_{\Delta} \texttt{cmd}[\rho']\ \{m_1\}; u' \cdot_{\delta+1} \vec{u}_1$ (part 1)
(6) $\sigma; \Sigma; x \leftarrow e; m \Downarrow_{\Delta}^{(a,\rho)} \sigma''; \Sigma''; v; [u' \cdot_{\delta+1} \vec{u}_1] \oplus_a g_1 \oplus_a [u] \oplus_a g_2$ (C-Bind)
(7) $\sigma; \Sigma; a \underset{\rho}{\hookrightarrow} (\delta, x \leftarrow e; m) \Downarrow_{\Delta} \sigma''; [u']\overline{\oplus}_a g$ (CT-Concat)

Case

$$\textsc{D-Bind2}\ \frac{m_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \Sigma', m_1', \mu')}{x \leftarrow \texttt{cmd}[\rho]\ \{m_1\}; m_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \Sigma', x \leftarrow \texttt{cmd}[\rho]\ \{m_1'\}; m_2, \mu')}$$

(1) $\sigma; \Sigma; m_1' \Downarrow_\Delta^{(a,\rho)} \sigma_1; \Sigma_1; v_1; g_1$ (inversion on C-BIND)
(2) $\sigma_1; \Sigma_1; [v_1/x]m \Downarrow_\Delta^{(a,\rho)} \sigma''; \Sigma''; v; g_2$ (inversion on C-BIND)
(3) $\sigma, a \hookrightarrow (v_1, \Sigma_1); \Sigma; a \underset{\rho}{\hookrightarrow} (\delta, x \leftarrow \mathtt{cmd}[\rho]\ \{m_1'\}; m_2) \Downarrow_\Delta \sigma''; (g_1 \oplus_a [u] \oplus_a g_2) \lfloor_\delta^a$
(inversion on CT-CONCAT, CT-THREAD, C-BIND)
(4) $\sigma, a \hookrightarrow (v_1, \Sigma_1); \Sigma; \mu' \Downarrow_\Delta \sigma_3; g_3$ (inversion on CT-CONCAT)
(5) $\sigma; \Sigma; \mu' \Downarrow_\Delta \sigma_3; g_3$ (inversion on DT-THREAD, Theorem 5, Lemma 3)
(6) $\sigma; a \underset{\rho}{\hookrightarrow} (\delta, m_1') \uplus \mu'; \sigma_1, \sigma_3 \Downarrow_\Delta g_1 \uplus g_3;$ (CT-CONCAT)
(7) $\sigma; \Sigma; m_1 \Downarrow_\Delta^{(a,\rho)} \sigma_1, \sigma_3; \Sigma_1, \Sigma_3; v_1; [u']\overline{\oplus}_a(g_1 \uplus g_3)$ (induction)
(8) $\sigma; \Sigma; x \leftarrow \mathtt{cmd}[\rho]\ \{m_1\}; m_2 \Downarrow_\Delta^{(a,\rho)} \sigma'', \sigma_3; \Sigma'', \Sigma_3; v; [u']\overline{\oplus}_a((g_1 \oplus_a [u] \oplus_a g_2) \lfloor_\delta^a \uplus g_3)$
(C-VAL, C-BIND)
(9) $\sigma, a \hookrightarrow (v_1, \Sigma_1); \Sigma; a \underset{\rho}{\hookrightarrow} (\delta, x \leftarrow \mathtt{cmd}[\rho]\ \{m_1\}; m_2) \Downarrow_\Delta \sigma'', \sigma_3; [u']\overline{\oplus}_a g$
(CT-CONCAT)

Case

D-BIND3
$$\frac{e\ \mathtt{val}_\Sigma}{x \leftarrow \mathtt{cmd}[\rho]\ \{\mathtt{ret}\ e\}; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (0, \cdot, [e/x]m, \emptyset)}$$

(1) $\sigma; \Sigma; [e/x]m \Downarrow_\Delta^{(a,\rho)} \sigma'; \Sigma'; v; g$ (inversion on CT-THREAD)
(2) $\sigma; \Sigma; \mathtt{cmd}[\rho]\ \{\mathtt{ret}\ e\} \leftarrow x; m \Downarrow_\Delta^{(a,\rho)} \sigma'; \Sigma'; v; [u]\overline{\oplus}_a g$ (C-VAL, C-RET, C-BIND)

Case

D-SPAWN
$$\frac{b \text{ fresh}}{\mathtt{spawn}[\rho; \tau]\ \{m\} \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (0, b{\sim}\tau@\rho, \mathtt{ret}\ \mathtt{tid}[b], b \underset{\rho}{\hookrightarrow} (0, m))}$$

(1) $\sigma; \Sigma; m \Downarrow_\Delta^{(b,\rho)} \sigma''; \Sigma, \Sigma''; v'; g$ (inversion on CT-CONCAT, CT-THREAD, C-RET)
(2) $\sigma; \Sigma; \mathtt{spawn}[\rho; \tau]\ \{m\} \Downarrow_\Delta \sigma'', b \hookrightarrow (v', \Sigma''); [u]\overline{\oplus}_a g$ (C-SPAWN)

Case

D-SYNC1
$$\frac{e \rightarrow_\Sigma^\Delta (\delta, e')}{\mathtt{sync}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}^{\Delta} (\delta, \cdot, \mathtt{sync}\ e', \emptyset)}$$

(1) $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \{(b, u)\}, \emptyset), e' \Downarrow_\Delta \mathtt{tid}[b]; \vec{u}$ (inversion on C-SYNC)
(2) $e \Downarrow_\Delta \mathtt{tid}[b]; u' \cdot_{\delta+1} \vec{u}$ (part 1)
(3) $\sigma, b \hookrightarrow (v, \Sigma_b); \Sigma; \mathtt{sync}\ e$
$\Downarrow_\Delta^{(a,\rho)} \sigma'', b \hookrightarrow (v, \Sigma_b); \Sigma''; v; (a \underset{\rho}{\hookrightarrow} (0, u' \cdot_{\delta+1} \vec{u} \cdot u), \emptyset, \{(b, u)\}, \emptyset)$ (C-SYNC)

Case

D-SYNC2
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{sync}\ (\mathtt{tid}[b]) \overset{v \lhd b}{\underset{\Sigma}{\mapsto}}^{\rho} (0, \cdot, \mathtt{ret}\ v, \emptyset)}$$

(1) $g = \emptyset$ (inversion on C-RET)

(2) $\sigma, b \hookrightarrow (v, \Sigma_b); \Sigma; \texttt{sync}\ (\texttt{tid}[b]) \Downarrow_\Delta^{(a,\rho)} \sigma''; \Sigma''; v; (a \underset{\rho}{\hookrightarrow} (0, u), \emptyset, \{(b, u)\}, \emptyset)$ (C-SYNC)

Case

$$\frac{e \rightarrow_\Sigma^\Delta (\delta, e')}{\texttt{poll}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \cdot, \texttt{poll}\ e', \emptyset)}\ \text{D-POLL1}$$

(1) $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u} \cdot u), \emptyset, \emptyset, E_w), e' \Downarrow_\Delta \texttt{tid}[b]; \vec{u}$ (inversion on C-POLL-SOME, C-POLL-NONE)

(2) $e \Downarrow_\Delta \texttt{tid}[b]; u' \cdot_{\delta+1} \vec{u}$ (part 1)

(3) $\sigma, b \hookrightarrow (v_b, \Sigma_b); \Sigma; \texttt{poll}\ e \Downarrow_\Delta^{(a,\rho)} \sigma'', b \hookrightarrow (v_b, \Sigma_b); \Sigma''; v; (a \underset{\rho}{\hookrightarrow} (0, u' \cdot_{\delta+1} \vec{u} \cdot u), \emptyset, \emptyset, E_w)$ (C-POLL-SOME, C-POLL-NONE)

Case

$$\frac{e \rightarrow_\Sigma^\Delta (\delta, e')}{\texttt{ret}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta, \cdot, \texttt{ret}\ e', \emptyset)}\ \text{D-RET}$$

(1) $g = (a \underset{\rho}{\hookrightarrow} (0, \vec{u}), \emptyset, \emptyset, \emptyset), e' \Downarrow_\Delta v; \vec{u}$ (inversion on the cost semantics)

(2) $e \Downarrow_\Delta v; u \cdot_{\delta+1} \vec{u}$ (part 1)

(3) $\sigma; \Sigma; \texttt{ret}\ e \Downarrow_\Delta^{(a,\rho)} \sigma''; \Sigma''; v; (a \underset{\rho}{\hookrightarrow} (0, u \cdot_{\delta+1} \vec{u}), \emptyset, \emptyset, \emptyset)$ (C-RET)

3. Case

$$\frac{v\ \texttt{val}_\Sigma}{\texttt{poll}\ (\texttt{tid}[b]) \overset{v?b}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (0, \cdot, \texttt{ret inl}\ v, \emptyset)}\ \text{D-POLL2A}$$

(1) $g = [u]$ (inversion on C-RET, C-INL)

(2) $\sigma, b \hookrightarrow (v, \Sigma_b); \Sigma; \texttt{poll}\ (\texttt{tid}[b]) \Downarrow_\Delta^{(a,\rho)} \sigma''; \Sigma''; \texttt{l} \cdot v; (a \underset{\rho}{\hookrightarrow} (0, u), \emptyset, \emptyset, \{(b, u)\})$ (C-POLL-SOME)

Case

$$\frac{v\ \texttt{val}_\Sigma}{\texttt{poll}\ (\texttt{tid}[b]) \overset{\not v b}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (0, \cdot, \texttt{ret inr}\ \langle\rangle, \emptyset)}\ \text{D-POLL2B}$$

(1) $g = [u]$ (inversion on C-RET, C-INR, C-VAL)

(2) $\sigma, b \hookrightarrow (v, \Sigma_b); \Sigma; \texttt{poll}\ (\texttt{tid}[b]) \Downarrow_\Delta^{(a,\rho)} \sigma''; \Sigma''; \texttt{r} \cdot \langle\rangle; (a \underset{\rho}{\hookrightarrow} (0, u), \emptyset, \emptyset, \emptyset)$ (C-POLL-NONE)

4. By induction on the derivation of $\sigma'; \Sigma'; \mu' \Downarrow_\Delta \sigma''; g$.

5.

(1) $\sigma; \Sigma; \nu\Sigma\{\mu_0 \uplus a_1 \underset{\rho_1}{\hookrightarrow} (\delta_1, m_1') \uplus \mu_1' \uplus \ldots a_n \underset{\rho_n}{\hookrightarrow} (\delta_n, m_n') \uplus \mu_n'\} \Downarrow_\Delta \sigma'; g_0'$ (part 4)

(2) g_0' with delays decreased is isomorphic to g' (part 4)

(3) $g' = g_0 \uplus g_1' \uplus \cdots \uplus g_n', \forall i. \exists \sigma_i, \Sigma_i. \sigma_i; \Sigma_i; a_i \underset{\rho_i}{\hookrightarrow} (\delta_i, m_i') \uplus \mu_i' \Downarrow_\Delta \sigma_i'; g_i'$ (inversion on the cost semantics)

(4) $a_i \underset{\rho_i}{\hookrightarrow} (0, m_i) \overset{\alpha_i}{\underset{\Sigma}{\mapsto}}{}^{\Delta} (\delta_i, \Sigma_i', m_i', \mu_i')$ (inspection of transition rules)

(5) $\sigma_i; \Sigma_i; m_i \Downarrow_\Delta^{(a_i, \rho_i)} \sigma_i'; \Sigma_i'; v; g_i, g_i$ is isomorphic to $[u_i] \overline{\oplus}_a g_i'$ (part 2)

(6) $\sigma; \Sigma; \mu \Downarrow_\Delta \sigma'; g, g$ is isomorphic to $g_0 \uplus ([u_1] \oplus_{a_1} g_1) \uplus \cdots \uplus ([u_n] \overline{\oplus}_{a_n} g_n)$ (C-CONCAT)

□

We can now repeatedly apply the above results to show a step-by-step correspondence between certain executions of λ^4 programs and schedules of the corresponding DAG. To be more precise, we show that, for any execution of a program, there exists a cost graph g corresponding to the program, and a schedule of g that corresponds to the execution. If the threads at each parallel transition are chosen in a "fairly prompt" manner by stepping as many threads as possible and prioritizing high-priority threads, then the corresponding schedule is fairly prompt. We do not specify in this formalism how to pick the threads of a parallel transition, but we discuss an appropriate scheduling algorithm in Chapter 6.

Lemma 12. *Suppose* $\vdash^R \mu : \Sigma$ *and* μ polled *and* $\mu \underset{P}{\Rightarrow}^{\Delta^*} \mu'$ *where* $\cdot; \cdot; \mu' \Downarrow_\Delta \cdot; \emptyset$ *and thread* a *is active for* T *transitions and at each transition, threads are chosen in a fairly prompt manner. Then* $\cdot; \cdot; \mu \Downarrow_\Delta \cdot; g$ *and there exists a fairly prompt and admissible schedule of* g *in which* $T(a) = T$.

Proof. By induction on the derivation of $\mu \underset{P}{\Rightarrow}^{\Delta^*} \mu'$. If $\mu = \mu'$, then the result is clear. Suppose $\mu \overset{\{a_1, \ldots, a_n\}}{\underset{P}{\longrightarrow}}{}^{\Delta} \mu'' \underset{P}{\Rightarrow}^{\Delta^*} \mu'$, and a is active for T transitions of the latter execution. By . By induction, $\cdot; \cdot; \mu'' \Downarrow_\Delta \cdot; g''$ and there exists a fairly prompt and admissible schedule of g'' where $T(a) = T$. By Lemma 11, $\cdot; \cdot; \mu \Downarrow_\Delta \cdot; g$, where g is isomorphic to $g_0 \uplus ([u_1] \oplus g_1) \uplus \cdots \uplus ([u_n] \oplus g_n)$.

By Lemma 8, vertices $u_1, \ldots, u_n$ are ready in g, so the schedule that executes $u_1, \ldots, u_n$ in step 1 and then follows the schedule of g'' is a valid schedule of g. Because (also by Lemma 8), all threads that are ready in g are available to be executed and (by inspection of the cost semantics) thread priorities are preserved between g and μ, the schedule is also a prompt schedule of g. We must also show that this schedule is admissible. By construction of DT-PAR, we have μ'' polled, so by Lemma 10, g'' is admissible. Because the remainder of the schedule is admissible, the entire schedule is thus admissible.

Finally, if $a \in dom(\mu)$, then by Lemma 8, a is ready in g and the resulting schedule has $T(a) = T + 1$. Otherwise, the resulting schedule has $T(a) = T$. □

Finally, we conclude by applying Theorem 2 to bound the response time of fairly prompt schedules, and therefore of the corresponding executions of the operational semantics.

Theorem 7. *If* $\cdot \vdash_{\cdot}^{R} m \div \tau \mathbin{@} \rho$ *and* $a \underset{\rho}{\hookrightarrow} (0, m) \underset{P}{\Rightarrow}^{\Delta^*} \mu'$*, where* $\cdot; \cdot; \mu' \Downarrow_{\Delta} \cdot; \emptyset$ *and thread* a *is active for* T *transitions and at each transition, threads are chosen in a fairly prompt manner, then there exists a graph* g *such that* $\cdot; \cdot; m \Downarrow_{\Delta}^{(a,\rho)} \sigma; \Sigma; v; g$ *and for any* $\rho' \preceq \rho$*,*

$$E[T] \leq \frac{1}{C(\not\prec \rho')} \left(\frac{W_{\not\prec \rho'}(\sharp a)}{P} + S_a(\sharp a) \right)$$

Proof. By Lemma 12, there exists such a g and a fairly prompt schedule of g where $T(a) = T$. By Lemma 6, g is well-formed. Thus, the result follows from Theorem 2. □

Chapter 6

Scheduling Algorithm for Prioritized Threads

> If it were done when 'tis done, then 'twere well
> it were done quickly.
>
> *Macbeth* (I.7.1–2)

In Chapter 5, we showed that a fairly prompt schedule of a PriML program obeys reasonable bounds on response time. However, this result omitted any discussion of how to choose threads to execute in a fairly prompt manner. In this sense, the result of that chapter could be considered an "offline bound": the bound it gives applies to schedules computed offline, prior to or separate from the execution of the program.

Brent's Theorem [25] and similar results for classical parallel programs are also offline bounds. For these non-interactive programs, *randomized work stealing* is an online scheduling algorithm that has been shown to approximate the bound given by Brent's Theorem and perform well in practice [6, 22]. Inspired by these results, we build an algorithm called Prioritized Private Deques (PPD) based on randomized work stealing to schedule threads in PriML programs. Formally proving that the algorithm asymptotically approximates the bounds of Chapter 5 is outside the scope of the thesis, but we will explain the motivation behind the algorithm's design and the intuition for why it is a reasonable approximation of fairly prompt scheduling. In Chapter 9, we give a quantitative evaluation of an implementation of this scheduling algorithm in the context of the PriML runtime.

6.1 PPD Algorithm Overview

The PPD algorithm differs from typical approaches to parallel scheduling in that it must handle threads of different priorities according to the fairness criterion described in Section 3.4. To this end, scheduling proceeds in rounds. At the beginning of each round, each processor picks a *primary priority* by sampling the probability distribution indicated by the fairness criterion. Essentially, the processor runs a standard work stealing scheduler on threads of the primary priority until a specified interval called the scheduling *quantum* has passed. At this point, the processor begins a new round by choosing a new primary priority.

To meet the promptness requirement, when a processor's primary priority is unavailable, it temporarily defaults to working on the highest-priority threads available at that processor. In this case, we say the processor is "donating" cycles to another priority. We use the term *current priority* to refer to the priority at which a processor is currently working, which may differ from the primary priority if it is donating cycles.

Once a priority is chosen, the PPD algorithm behaves more or less like a standard work stealing algorithm working on threads of that priority. We have some flexibility in the particular work stealing scheduler we use for this purpose. We use the *sender-initiated private deques* variant of work stealing [2]. In this algorithm, each processor executes threads which are stored in a processor-local set. Unlike in more traditional work stealing algorithms (e.g., [6, 22]), processors do not directly access other processors' queues; instead, load balancing is performed by message passing between threads. The algorithm is *sender-initiated* in that processors periodically attempt to send (or "deal") extra threads to processors which are idle.

Our use of sender-initiated private deques is motivated by the unique concerns of our algorithm. First, private deques allow us flexibility in choosing performant implementations of the processor-local sets of ready threads, because the data structure need not be concurrent. Second, we expect sender-initiated schedulers to distribute high-priority work more effectively than their receiver-initiated counterparts. Specifically, if high-priority work is being spawned on one processor, a sender-initiated algorithm immediately begins distributing that work to others.

We now give a high-level overview of the PPD algorithm. All of the elements will be described in more detail in the following sections. Each processor maintains a set of ready threads called a *thread bank*. In keeping with the private deques model, thread banks are accessible only to the processor that owns them. Threads are shared between processors by message passing, using an abstraction we call *mailboxes*.

The core of the algorithm is a scheduling loop. Each iteration of the loop performs some housekeeping operations and then attempts to find a thread to execute. The housekeeping operations are:

1. Choose a new primary priority and start a new round if it is time to do so.
2. Add back to the thread bank any threads previously suspended on I/O operations which have resumed since the last iteration.
3. If it is time to do so, perform a "deal attempt": pick a random processor and attempt to deal it a thread at the current priority.

After performing these tasks, the processor attempts to execute a thread, first by looking in the processor's own thread bank at the primary priority, then at the highest available priority. In both cases, a processor considers both threads already in its thread bank and threads that may have been dealt to it by other processors.

6.2 Notation, Terminology and Data Structures

This section describes in more detail how we represent the state, parameters and data structures mentioned in the overview.

```
type bank
insert : bank * thread -> unit
removeMin : bank -> thread
split : bank -> thread_set
```

Figure 6.1: The thread bank interface.

Parameters. The scheduling algorithm takes a number of parameters:

- P, the number of processors
- R, the number of priorities.
- `roundQuantum`, the length of a round (in arbitrary units)
- `dealInterval`, the interval between deal attempts (in the same units)
- `fc`, the fairness criterion, represented as a probability distribution over priorities.

Threads. The units of work handled by the scheduler are *threads*. Each thread has an associated priority, which is accessible with the function `priority`. Threads may be executed with the function `execute`, which runs the thread until it terminates, spawns, or suspends. Executing a thread returns a set of zero, one, or two threads which have become enabled (zero if the thread terminates or suspends waiting for an unavailable resource, one if it can be rescheduled immediately, and two if it spawns a new thread).

Processor-local state. All of the state in the algorithm, with the exception of the mailboxes used for inter-processor communication, is processor-local. Each processor has access to a globally synchronized timer accessible via the function `now`, which returns the current time. The local variable `nextRound` is used to record the time at which the processor should switch to a new primary priority. The local variable `nextDeal` records the time at which the processor should next attempt to deal threads to another processor. The variable `primaryPriority` records the primary priority of the current round, and `currentPriority` records the priority at which the processor is actually working. Each processor maintains a set `ioThreads` of I/O-blocked threads which have resumed since the last iteration of the scheduling loop. One could imagine that this set is populated asynchronously with the execution of the scheduler by callbacks which are called by the system when I/O events occur. We discuss our actual implementation of I/O in Chapter 7. Finally, each processor maintains R local thread banks, where R is the number of priorities. Each bank stores threads of a particular priority.

Thread banks. Thread banks store a set of threads and permit insertion, removal, and splitting. An interface is shown in Figure 6.1. When new threads are spawned, they are inserted into the appropriate bank. Threads are removed from the bank to be executed locally. When a processor decides to deal work to another processor, it splits its bank to acquire a set of threads to send.

Threads in a bank are ordered according to a heuristic we call *fork potential*, inspired by a related concept used by Acar et al. [2]. We write $f(v) \in \mathbb{R}^+$ for the fork potential of a

```
type mailbox
tryClaim : mailbox -> bool
send : mailbox * thread_set -> unit
open : mailbox -> unit
close : mailbox -> thread_set
```

Figure 6.2: The mailbox interface.

thread v. The core requirement for fork potential is that, if a thread v depends upon another thread u, then $f(u) > f(v)$. Fork potential is a good approximation of a thread's "size" in that a thread with high fork potential is more likely to spawn additional work. This suggests that threads with high fork potential are good candidates to be dealt to other processors during load balancing. Thus the function `removeMin`—which is used by a processor to select a thread to execute locally—removes the thread from the bank with the minimum fork potential. The function `split` removes a set of threads from the bank comprising approximately half of the total fork potential of the bank, to be dealt to another processor.

Mailboxes. All communication between processors in the algorithm is performed through *mailboxes*. The algorithm uses $P \cdot R$ mailboxes, one for each processor at each priority. A mailbox for a priority ρ at a processor p is used for two purposes: first, for p to indicate to other processors whether it is accepting work at priority ρ (i.e., it does not already have work at that priority), and second, for other processors to send it work. To avoid races, a remote processor must *claim* a mailbox before placing work in it.

A mailbox can be in one of four states:

- A mailbox is CLOSED if the processor is not accepting work at that priority.
- A mailbox is OPEN if the processor is accepting work at that priority and another processor is not yet attempting to send it work at that priority.
- A mailbox is CLAIMED if a remote processor has claimed the mailbox, expressing an intention to place work in it.
- A mailbox is FULL if a remote processor has placed work in it.

The mailbox functions, shown in the interface in Figure 6.2, transition mailboxes between the above states. An OPEN mailbox may be CLAIMED using the function `tryClaim`. If multiple processors attempt to claim a mailbox simultaneously, at most one may succeed. Once a processor has CLAIMED a mailbox, it may call `send` to place a set of threads in the mailbox, transitioning it to FULL. A processor may transition its mailbox from CLOSED to OPEN, indicating it is accepting work in that mailbox, by calling the function `open`. Closing a mailbox (with `close`) requires more care, since the mailbox may be OPEN, CLAIMED or FULL. An OPEN mailbox is simply closed atomically, causing subsequent claim attempts to fail. This call returns an empty thread set. Closing a FULL mailbox returns the contents of the mailbox, and transitions the mailbox to CLOSED so no future claims are possible. If the mailbox is CLAIMED, the calling processor blocks until the processor that claimed the mailbox sends work, at which point the work is returned and the mailbox is closed. Attempting to close a CLOSED mailbox has

no effect.

6.3 The Scheduling Loop

Pseudocode for the scheduling loop of a single processor is shown in Figure 6.3. This code calls several auxiliary functions defined in Figure 6.4. On each iteration of the loop, the processor performs the following.

1. If it is time to switch to the next round, draw a new primary priority and set the next switch time (lines 21-23).
2. Insert into the appropriate thread banks any newly resumed I/O-blocked threads. Close the mailboxes for the priorities of these threads since work is now available (lines 24-27).
3. Perform a deal attempt if it is time to do so (lines 28-30). Deal attempts are described in more detail below.
4. Attempt to get work at the primary priority (line 31).
5. If the previous step was unsuccessful, determine the highest locally available priority and get work at it (lines 32-33).
6. If a thread is found, execute it and handle any new threads it enables (lines 34-39).

Several of these steps require additional explanation.

Handling resumed I/O-blocked threads. In this step, the scheduler simply iterates over the set of newly resumed I/O threads and adds them to the appropriate thread banks. This is quite a different strategy from our prior work on I/O in work stealing scheduling [86], in which we created new deques on steals in order to maintain deque ordering invariants when I/O-blocked threads resumed. This policy was necessary to ensure the efficiency of that algorithm, in which threads were always pushed and popped from the bottoms of deques. This is unnecessary in PPD, where the deque ordering is maintained explicitly. This is discussed further in Section 6.4.

As in the prior work, handling resumed I/O threads requires work that is non-constant in the size of `ioThreads`, but this work can be amortized over the work of the threads themselves. Our prior work used parallel for loops to reduce the span of adding I/O threads back to the deque. We don't consider this parallelization strategy here since the practical impact should be minimal, but it should be possible to parallelize the addition of I/O threads in this algorithm as well.

Deal attempts. To perform a deal attempt, a processor p picks another processor q at random and checks if q is idle at p's current priority by attempting to claim the appropriate mailbox (which will fail if the mailbox is CLOSED). If p succeeds in claiming the mailbox, it splits the thread bank at the current priority and sends the resulting thread set to processor q by writing them into the mailbox.

Getting Work. When a processor attempts to get work at a particular priority, function `getWork` first checks if it has been dealt work at this priority by closing the appropriate mailbox and adding any returned threads to the thread bank. The function then returns the "bottom" thread from the

```
// Parameters
int P // number of processors
int R // number of priorities
time roundQuantum
time dealInterval
fairness_criterion fc

// Global (shared) state
mailbox mailboxes[P, R]

scheduleLoop(p): // p = identifier of processor
  // Processor-local state
  bank banks[R]
  time nextRound
  time nextDeal
  priority primaryPriority
  priority currentPriority
  thread_set ioThreads

  repeat:
    if now() > nextRound:
      nextRound := now() + roundQuantum
      primaryPriority := randomFrom(fc)
    for t in ioThreads:
      insert(banks[priority(t)], t)
      closeMailbox(p, banks, priority(t))
    ioThreads := {}
    if now() > nextDeal:
      nextDeal := now() + dealInterval
      dealAttempt(p, banks, currentPriority)
    t := getWork(p, banks, primaryPriority)
    if t = NULL:
      t := getWork(p, banks, findHighestPrio(banks))
    if t != NULL:
      currentPriority := priority(t)
      enabled := execute(t)
      for t' in enabled:
        insert(banks[priority(t')], t')
        closeMailbox(p, banks, priority(t'))
```

Figure 6.3: Scheduler loop pseudocode

```
dealAttempt(p, banks, prio):
  q := randomFrom([1,P]\{p})
  m := mailboxes[q, prio]
  if tryClaim(m):
    send(m, split(banks[prio]))

getWork(p, banks, prio):
  closeMailbox(p, banks, prio)
  t := removeMin(banks[prio])
  if t = NULL:
    open(mailboxes[p, prio])
  return t

closeMailbox(p, banks, prio):
  threads := close(mailboxes[p, prio])
  for t in threads:
    insert(banks[prio], t)
```

Figure 6.4: Scheduler auxiliary functions

bank. If the bank is empty, it reopens the mailbox to indicate that the processor is still accepting work at that priority.

Finding the highest available priority. When the thread bank of a processor's primary priority is empty, it attempts to find other work to execute while waiting for a deal to arrive. To find other work, it calls the function `findHighestPrio` which returns the highest priority at which the corresponding bank is not empty (if all banks are empty, then it returns any priority arbitrarily). This helps the scheduler be *prompt* by executing high priority work not only during rounds dedicated to that priority, but also when other work is currently unavailable. Note that this is entirely a local notion: the function `findHighestPrio` does not search for the highest available priority across all processors, but rather only looks for the highest available priority amongst the banks stored locally at the current processor.

A naïve implementation of `findHighestPrio` could simply inspect every bank, beginning with the highest priority used in the program. We describe a more optimized implementation in Section 7.1.

6.4 Intuitions for Cost Bound

An analysis of the PPD algorithm to show that it approximates the bounds for fairly prompt schedules given in Chapter 5 is outside the scope of this thesis. However, the design of the algorithm was motivated by the desire to have scheduling remain as fair and prompt as possible. In addition, many of our design choices were inspired by prior implementations and analyses of

randomized work stealing. This gives us reason to believe that the algorithm described in this chapter constructs a good approximation of a fairly prompt schedule. In this section, we explain these intuitions at an informal level.

Fairness and work stealing. The use of work stealing is partially motivated by a classic result by Arora et al. [6], which shows that a randomized work stealing scheduler running in a multiprogrammed environment receiving an average of P_A processors executes in time $O\left(\frac{W}{P_A} + S\frac{P}{P_A}\right)$. This result suggests that a work stealing scheduler receiving a fraction $C(\rho)$ of cycles on each of P processors can behave as if it were running on a dedicated machine with $P \cdot C(\rho)$ processors. To a first level of approximation, the PPD algorithm behaves like a work stealing scheduler at its primary priority for the fraction of cycles allotted to that priority. Thus, we might expect the response time of threads at priority ρ under this approximation of the scheduler to be

$$\frac{W_\rho(\sharp a)}{P \cdot C(\rho)} + S_a(\sharp a)\frac{P}{P \cdot C(\rho)} = \frac{1}{C(\rho)}\left(\frac{W_\rho(\sharp a)}{P} + S_a(\sharp a)\right)$$

which begins to look suggestively like the bounds of Chapter 5, and in particular looks a great deal like that of Theorem 3, which considers only the affects of fairness and not the donation of unused cycles.

There are a number of obstacles to applying the analysis of Arora et al. to our setting. First and most problematic, our scheduler is not that of Arora et al.'s randomized work stealing: we are using a private deques scheduler instead. The private deques algorithm has been thoroughly analyzed using a somewhat different technique from Arora et al.'s analysis [2], resulting in a different but comparable bound[1] It seems likely that the private deques analysis could be extended to our algorithm using techniques similar to Arora et al.'s for multiprogrammed environments.

Promptness. The above intuitions only account for *fairness* in suggesting that the algorithm should behave greedily at a particular priority for the fraction of the time indicated by the fairness criterion. These intuitions do not account for the fact that cycles at which the primary priority is unavailable are donated to the highest available priority. The purpose of these "donations" is to ensure that the scheduler is also *prompt*. This donation policy closely follows the requirement of the fairly prompt scheduling principle that, as stated in Section 3.4, "When a particular priority is unavailable (i.e., has no threads available in the system), it 'donates' its cycles to the highest available priority." Because of this correspondence, we expect that the scheduler meets the "promptness" requirement of fairly prompt scheduling. Donations are, on the other hand, likely to increase the execution time by a factor of R, since finding the highest available priority, in the worst case, requires a linear scan. Our implementation (7.1) avoids this linear scan in most cases, and causes the algorithm to empirically perform well even on contrived benchmarks with large numbers of priorities, but it is not yet clear if this is an asymptotic improvement in general.

[1] Acar et al.'s bound for private deques work stealing is $\left(1 + \frac{1}{\delta - 1}\right) \cdot \left(\frac{W}{P} + S + O(\delta F)\right)$, where δ is related to the interval between deal attempts and F is the branching depth.

Work stealing invariants. Applying existing analyses of work stealing to the PPD algorithm, of course, requires that our algorithm maintains the same invariants assumed by these analyses. One such important invariant is the so-called *deque invariant* that tasks are ordered in the work-stealing deques from top to bottom in order of increasing depth. This ensures that shallower, larger tasks are stolen first, amortizing the cost of steals over a large piece of work. Standard work stealing algorithms maintain this property by construction because processors push and pop from the bottom of their deque.

In responsive prioritized programs, two processes can break the deque invariant: first, threads that are suspended on I/O can resume at arbitrary points in the program, and in particular could resume after the thread bank has been emptied and repopulated with work stolen from another processor. Without tracking additional information, the scheduler would have no idea of the relative depths of the newly resumed thread and the threads in the bank. In prior work [86], we maintain the deque invariant in this situation by creating a new deque on steals when tasks are suspended. The deque invariant applies individually to each deque, but the additional deques drive up the execution time because steals take longer on average.

The second situation in which the deque invariant could break is when a thread at priority ρ_1 spawns work at a priority ρ_2. The new thread must be placed in the thread bank at priority ρ_2 but, as in the situation above, there may already be unrelated work in the thread bank.

The PPD algorithm handles both situations by explicitly ordering the thread bank. This strategy is enabled by the use of private deques: because the thread bank data structure does not need to be concurrent, we have much more flexibility in how we maintain the data structure. By explicitly ordering the thread banks, we avoid the need for multiple deques per processor at the cost of making out-of-order insertions logarithmically more expensive (we are effectively maintaining a priority queue, so either insertion or removal must be non-constant). In our implementation, discussed in Chapter 7, in-order insertion, which is a much more frequent operation, is constant time. The asymptotic overhead for out-of-order insertion (and split) will show up in any formal analysis of the algorithm, but we expect these operations to be rare and may be able to amortize them over other operations (e.g., steals, which existing analyses of work stealing show to be rare).

Chapter 7

Implementation

> I like thy counsel; well hast thou advised:
> And that thou mayst perceive how well I like it,
> The execution of it shall make known.
>
> *The Two Gentlemen of Verona* (I.3.35–37)

We have implemented the ideas of this thesis in three major pieces: the *front end*, the *threading library* and the *back end*. The front end is a type checker and elaborator for the PriML language which implements the type system described in Chapter 4. The front end generates code in Standard ML, with thread and priority operations replaced by calls to the threading library, an SML library in which we have implemented the threading operations of PriML. The library is built on top of a parallel extension of the MLton Standard ML compiler developed by Daniel Spoonhower [112], which is being maintained and updated in ongoing work [99]. The updated version of Spoonhower's framework, known as `mlton-parmem`, provides basic facilities for running MLton threads on multiple processors, and allows user-defined threading libraries and schedulers (Spoonhower also defined his own threading primitives and schedulers, upon which we draw in developing our library and back end). The core of our back end is an implementation of the PPD algorithm of Chapter 6, which schedules the threads generated by our threading library.

In the remainder of this chapter, we discuss each piece of the implementation in more detail.

7.1 Back end

The back end consists of an SML implementation of the PPD scheduling algorithm as a scheduler for `mlton-parmem`. The main scheduler code consists of approximately 600 lines of ML code. The thread bank implementation is approximately an additional 75 lines. The remainder of this section describes the implementation details of the scheduler, as well as various optimizations we implemented for better practical performance.

The implementation follows much the same structure as the description in Chapter 6: the main body of the code consists of the scheduling loop and its auxiliary functions, and the thread bank and mailbox data structures are factored out as separate modules. The remainder of this

section describes the implementation of these data structures, as well as lower-level details of the scheduler implementation not covered in the high-level algorithm specification, and optimizations present in our implementation.

Threads. Threads in our scheduler are lightweight wrappers around MLton's user-level thread objects, which are abstract representations of paused computations. As an additional optimization to avoid the small overhead of creating a user-level thread, threads in our system are allowed to simply be a thunk representing the work to be done by that thread. If the work is being performed locally, the existing MLton thread can simply execute the thunk. PPD threads that are stolen or interrupted (see "Interrupts" below) are automatically converted to MLton threads.

In addition to the underlying work of the thread, stored as either a MLton thread or a thunk, PPD threads are also tagged with their fork depth, an integer representing the thread's depth in the DAG (this is closely related to the fork potential, but easier to manipulate) and a *cancellable* data structure, discussed in more detail below.

Thread banks. The implementation of thread banks is not considered a contribution of this thesis, and so we describe it only briefly here. One key observation about the use of thread banks guides the design of a practical implementation: by far the most common way the scheduler uses the thread bank is to pop a thread from the local thread bank, and soon after push a thread of the same priority. In this use case, the pushed thread is guaranteed to have a deeper fork depth than all of the elements in the thread bank at its priority (because it descended from the previously popped thread). This observation leads us to support a `pushMin` operation which is intended to be much faster than the more general `insert`, which may have to insert out-of-order. Resumptions and cross-priority spawns (which require out-of-order insertion) and deal attempts (which require a split of the thread bank) are much rarer. Given these observations, an efficient implementation of thread banks should heavily optimize for the performance of `pushMin` and `removeMin`, even at the cost of making `insert` and `split` fairly expensive.

Thread banks for PriML implementations must also support a `tryRemove` operation which was not discussed in the algorithm description, but is necessary to support cancellation as well as a fast-path optimization used in our threading library. This operation takes a handle to a thread in the thread bank (handles are returned by insertion operations). If the thread is still in the thread bank, the operation removes the thread and returns true, indicating success. If the thread has been popped from the thread bank, the operation leaves the thread bank unaltered and returns false.

Our implementation uses a doubly-linked list (DLL) sorted by fork potential, and so `pushMin` and `removeMin` are simply a single append and remove, respectively, on the underlying DLL. To implement the `tryRemove` operation, we tag DLL nodes with boolean flags indicating whether the element has been removed from the list into which it was originally inserted. Handles are implemented simply as pointers to DLL nodes. On a call to `tryRemove`, the implementation checks the flag to see if the element is still part of the list, and if so, removes it.

Mailboxes. Here we describe in more detail how we implement mailboxes. The implementation is straightforward and largely follows from the original description of private deques [2].

A mailbox is represented by a flag and a contents cell. The contents cell is either `NULL`, or a pointer to a set of threads. The flag indicates the status of the mailbox with one of three different states: CLOSED, OPEN, or CLAIMED, corresponding approximately to the states of the same names as described for the mailbox interface in Chapter 6. The fourth state, FULL, is distinguished from CLAIMED by a non-`NULL` value in the contents cell.

The `open` function simply edits the flag, and similarly the `send` function simply edits the contents cell. The `tryClaim` function uses an atomic compare-and-swap on the flag and returns true if the compare-and-swap succeeded. Finally, `close` reads reads the flag to determine what it should do:

- if CLAIMED, it spins on the contents cell until a non-`NULL` pointer is found. It then closes the mailbox with a plain write to the flag.
- if OPEN, it uses a compare-and-swap on the flag to attempt to close it. A plain write in this case is not correct, because it could overwrite a successful transition of the flag to CLAIMED. If the compare-and-swap does not succeed, then it executes the approach for when the flag is CLAIMED.

I/O Queues. As in our prior work [86], our implementation does not rely on callbacks to implement I/O. Instead, the I/O queue contains elements of type

```
(unit -> bool) * thread * priority
```

The function component of the tuple is a predicate that returns true if the I/O event has occurred and the thread is ready to resume. Each time the I/O queue is processed, a loop checks each element `(f, t, r)` to see if it is ready by calling `f ()`. If this call returns true, the thread is added back to the thread bank of the appropriate priority.

Cancellables. The PriML runtime supports cancellation of threads using a system derived from *cancellables* in Manticore [51] (which is also similar to *try trees* in JCilk [37])[1]. The purpose of a cancellable is to explicitly record ancestor-descendant relationships among threads so that when one thread is cancelled, indicating that its results will no longer be needed, all of its descendants can be easily found and transitively cancelled. In our implementation, a cancellable is a record of four items:

1. A *cancelled* flag indicating whether the corresponding thread has already been cancelled (running threads periodically check this flag and stop working if the thread has been cancelled).
2. A reference to a *cancellation function* of type `unit -> unit`, which is called in order to cancel the thread. The function should remove the corresponding thread from a thread bank if it is in one. The cancellation function is used to cancel only one thread, and not its descendants.
3. A list of *children* which is traversed to transitively cancel the descendants of the thread.

[1]The Manticore literature used the spelling *cancelable*. We use the alternate spelling for consistency with the rest of the text in this thesis.

```
signature PRIORITY =
sig
    type t

    val top             : t
    val bot             : t

    val new             : unit -> t
    val new_lessthan    : t -> t -> unit

    val init            : unit -> unit
    val check           : unit -> unit

    (* Comparison operators *)
    val ple             : t * t -> bool
    val plt             : t * t -> bool
    val pe              : t * t -> bool

    (* Fairness criterion operations *)
    val installDist     : (t -> int) -> unit
    val chooseFromDist  : real -> t

    (* Convert to and from indices in the total order *)
    val toInt : t -> int
    val fromInt : int -> t
end
```

Figure 7.1: The priority interface.

4. A mutex to protect concurrent operations on cancellables.

When a thread is spawned, it is given a new cancellable that is marked as a descendant of the cancellable of the parent thread. Every time a thread is pushed to or popped from a thread bank, its cancellation function is updated accordingly.

Priorities. Because priorities are so central to PPD, priorities are built into the implementation of the back end (the remainder of the threading library, described in the next section, is built on top of the back end and can largely be separated from it). Priorities adhere to the interface shown in Figure 7.1, which is implemented in the back end by a structure `Priority : PRIORITY`. In addition to an abstract type of priorities, the interface provides functions for creating new priorities and ordering constraints. In the scheduler, priorities are represented using a total ordering consistent with the programmer-defined constraints. This allows for efficient and well-defined implementations of operations such as cycling through priorities and finding the highest available

priority. Internally, priorities are represented using their index in the total order. These indices can be converted to and from the abstract priority representation using the `toInt` and `fromInt` functions.

Before running prioritized code, a program should declare all necessary priorities using `new ()`, followed by a call to `init ()`. The latter initializes a Boolean matrix which will be used to track the ordering relation (the ordering relation is stored this way to make checking ordering constraints a constant-time operation). After the priorities are initialized, the program may define ordering constraints such as `p < q` using `new_lessthan (p, q)`. Finally, the program must call `check ()` to finalize and check the ordering relation. This operation first performs a topological sort to compute the total ordering used internally to identify priorities, and then uses Warshall's transitive closure algorithm to populate the above matrix with the least partial order consistent with the defined constraints. In the process, it raises an error if the ordering relation is found to be cyclic.

Finally, the priority interface provides functions for installing and using fairness criteria. The function `installDist` allows programs to install a fairness criterion to be used by the scheduler. The fairness criterion is specified as a function from priorities `p` to integers specifying the relative value of $C(\mathtt{p})$. These values are automatically normalized to 1. When the scheduler wishes to draw a priority at random according to the probability distribution indicated by the fairness criterion, it calls `chooseFromDist` with a pseudorandom real number between 0 and 1.

State. Though the algorithm description differentiates between global state (which consists only of the parameters to the algorithm and the mailboxes) and processor-local state, all of the mutable state in the SML implementation is global. This state is implemented as a collection of arrays of length P, the number of processors (the arrays of thread banks and mailboxes have length $P \cdot R$, since each processor needs one for each priority; these arrays are initialized at priority initialization when R is known). In addition to the processor-local state mentioned in the algorithm, the implementation also maintains the fork depth of the thread on which each processor is working, as well as its cancellable. This state is necessary to create new thread structures when a thread is spawned, because the new thread must be given a fork depth one deeper than the current depth and a cancellable that is a descendant of the current cancellable.

The scheduler interface. The high-level description of the algorithm assumes that a thread runs for a short amount of time and produces zero, one or two child threads. Reality is, of course, more complicated. Running threads need a way to interact with the runtime to create new threads and place them in the thread banks. The scheduler exposes a low-level interface which can be used for this purpose by threading libraries such as the one we describe in the next section. User-level code should generally not need these functions. The SML signature for this interface is given in Figure 7.2.

To spawn new work, a threading library would call `newThread` with the work to be done by the new thread, represented either as a thunk or a runnable MLton thread. This operation wraps the work in a thread data structure by incrementing the fork depth and allocating a new cancellable for the thread. The library would then call `push` to push the thread onto the appropriate

```
signature SCHEDULER =
sig
    type thread
    type hand
    datatype work = Empty
                  | Thunk of unit -> unit
                  | Thread of MLton.Thread.Runnable.t

    val newThread     : work -> thread
    val cancellable    : thread -> Cancellable.t

    val push          : Priority.t * thread -> hand
    val tryRemove     : Priority.t * hand -> bool

    val suspend       : (Priority.t * thread -> unit) -> unit
    val suspendIO     : (unit -> bool) -> unit

    val returnToSched : unit -> void
end
```

Figure 7.2: The scheduler interface.

thread bank. This function and `tryRemove` are fairly thin wrappers around the corresponding thread bank operations but also update the cancellation functions of the threads' cancellables.

The function call `suspend f` blocks the calling thread and calls `f` on the current thread's priority and continuation (where the continuation is represented as a thread). This operation is used, for example, to implement `sync`, which would call `suspend` with a function `f` that adds the thread to a list of continuations waiting on the target thread. Another use of `suspend` is to implement I/O operations. Simply calling a system function would block the system-level thread running the scheduling algorithm, preventing that processor from running other code. Instead, we call `suspend` with a function that adds the current thread to the I/O queue. Because this special use of `suspend` is so common, the interface also provides a wrapper `suspendIO` for this purpose. The caller must supply `suspendIO` with a predicate that returns true when the I/O-blocked thread is ready to resume (e.g., when user input is available).

Interrupts. The scheduling algorithm of Chapter 6 assumes that the user code returns to the scheduler frequently. As in all private deques algorithms, this is necessary to ensure that deals occur [2], but is especially important in our multi-priority setting so that priority switches happen on time and long-running low-priority computations are not allowed to delay the execution of a computation at a higher priority.

Most well-written parallel code is fairly fine-grained, i.e., sequential computations do not run for long without spawning or synchronizing. However, to ensure proper load balancing and responsiveness even in the presence of long-running sequential computations, we use periodic

interrupts delivered by interval timers to return control to the scheduler. When a thread is interrupted to return to the scheduler, the scheduler pushes the continuation of the running thread onto the thread bank so that it may be continued at a later time (possibly immediately if a priority switch does not occur). We determined experimentally that these interrupts should be at a finer granularity than the round-switch quantum. Specifically, our implementation performs interrupts at 1ms intervals and switches rounds at 5ms intervals.

Finding the highest priority. To implement `findHighestPrio` (as described in Chapter 6) efficiently, we use a heuristic which avoids unnecessary linear scans through the priorities.

Each processor maintains its highest currently available priority in a global variable, the *top-priority cell*. The function `findHighestPrio` first checks the bank of the top priority to see if it has work before inspecting every one of its thread banks. Processors update their own top-priority cell when they insert work at higher priorities, and atomically update other processors' cells on deals.

Requests. Sender-initiated algorithms such as ours perform well in situations where work on a small number of processors must be distributed. This is useful for distributing high-priority computations, which may be generated at one processor and need to be balanced onto the others. Sender-initiated algorithms are less useful when most processors have work and so most randomly targeted deal attempts will fail. We improve our implementation's performance in this case by adding an element of receiver-initiated algorithms in the form of *requests*. Each processor has a request cell. Processors which are idle at a particular priority may request work at that priority by writing their processor ID and the requested priority into a random processor's request cell. When a busy processor deals work, it will first check its request cell. If a request is present, it will attempt to deal work to the requesting processor instead of targeting randomly.

7.2 Threading library

The primitives exposed by the scheduler interface in Figure 7.2 are general enough to encode many interesting threading constructs. We have used these primitives to encode a threading library that matches the thread operations of PriML. The signature for our library is shown in Figure 7.3. The `THREAD` signature provides the operations for creating and handling threads. The `BASIC` signature provides a minimal interface with the scheduler itself, allowing user programs to indicate that all priorities have been declared (with `finalizePriorities`) and get the current priority (an operation that is not available in PriML since priorities are not first class, but which is occasionally useful in writing code using the library).

The thread operations `spawn`, `sync`, `poll` and `cancel` perform essentially the same functions as their PriML counterparts. Because the library is used in bare, unextended SML, the `spawn` construct takes a thunk for the new thread to execute. In addition, SML's type system is not powerful enough to encode the priority constraints on `sync` that the type system of PriML does, and so in programs using the threading library, priority checking is done dynamically rather than statically. A call to `sync` on a thread of priority lower than the current priority will raise an `IncompatiblePriorities` exception. A call to `sync` on a thread that has been cancelled

```
signature THREAD =
sig
    exception IncompatiblePriorities
    exception Thread

    type 'a t

    val spawn  : (unit -> 'a) -> Priority.t -> 'a t
    val sync   : 'a t -> 'a
    val poll   : 'a t -> 'a option
    val cancel : 'a t -> unit
end

signature BASIC =
sig
    val finalizePriorities : unit -> unit
    val currentPrio        : unit -> Priority.t
end
```

Figure 7.3: Signatures for the threading library.

raises a `Thread` exception. A call to `poll` on a thread that has been cancelled simply returns `NONE`.

We now discuss in detail the implementation of threads. This implementation is partially based on Spoonhower's implementation of futures [112]. As in Spoonhower's original implementation, the core of the thread representation is a reference cell to store the result (initially set to `Waiting` and overwritten when the thread finishes), and a mechanism for tracking and waking reading threads (which we call readers to avoid confusion) that are waiting on this thread to complete. The former uses a data type `'a result` for a thread with a return value of type `'a`. In our implementation, results also store the fork depth:

```
datatype 'a result =
         Waiting
       | Finished of 'a * int (* result, depth *)
       | Raised of exn * int
```

For tracking readers, we use a collection data structure called a Bag, whose signature is given in Figure 7.4. The Bag implementation, by Sam Westrick, performs appropriate synchronization on the `insert` operation, and between `insert` and `dump`. When a bag is `dump`ed, a list of previously inserted elements is returned. Subsequent insertions (or concurrent insertions not included in the dump) will fail, as will subsequent attempts to dump the bag.

The implementation of the thread type is shown in Figure 7.5. In addition to the result reference and the bag, a thread contains its priority, a handle to its thread in the deque, the thread's cancellable object and the thunk representing the work the thread is to perform.

```
signature BAG =
sig
  type 'a t

  val new : unit -> 'a t
  val insert : 'a t * 'a -> bool
  val isDumped : 'a t -> bool
  val dump : 'a t -> 'a list option
end
```

Figure 7.4: The BAG signature.

```
    type 'a t =
     {
       result : 'a result ref,
       prio   : Priority.t,
       bag    : (Priority.t * thread) Bag.t,
       hand   : hand,
       cancel : Cancellable.t,
       thunk  : unit -> 'a
     }

```

Figure 7.5: The thread implementation.

```
fun writeResult fr f =
    let val r = f ()
        val d = getDepth (processorNumber ())
    in
        fr := Finished (r, d)
    end
    handle e => fr := Raised (e, getDepth (processorNumber ()))

fun run (r, fr, bag) f () =
    let fun wake (r', t) =
            ((if Priority.pe (r, r') then push else insert) (r', t);
             ())
    in
        writeResult fr f;
        (case Bag.dump bag of
             NONE => raise Thread
           | SOME l => List.app (ignore o insert) l);
        returnToSched ()
    end

fun spawn f r' =
    let
        val p = processorNumber ()
        val r = curPrio p
        val fr = ref Waiting
        val bag = Bag.new ()
        val thread = newThread (Thunk (run (r', fr, bag) f))
        val hand = (if Priority.pe (r, r') then push else insert)
                       (r', thread)
        val c = cancellable thread
    in
        {result = fr, prio = r, bag = bag, hand = hand,
         cancel = c, thunk = f}
    end

```

Figure 7.6: The implementation of spawn.

```
1 fun poll ({result, bag, ...} : 'a t) =
2     if not (Bag.isDumped bag) then NONE
3     else case !result of
4               Finished (x, _) => SOME x
5             | Raised (e, _) => raise e
6             | Waiting => raise Thread
7
```

Figure 7.7: The implementation of poll.

The code to spawn a thread is straightforward and is shown in Figure 7.6. The code initializes the result cell to `Waiting`, and creates a new bag and new thread. The thread is initially a thunk that runs the function `f` inside a wrapper `run` that writes the result (or exception raised), together with the appropriate fork depth, into the result cell, dumps the bag, inserts any waiting threads into the deque, and returns control to the scheduler. This thread is then inserted into the deque. Note that, both in inserting readers and in inserting the new thread, we may use the faster `push` operations if the thread to be inserted is at the current priority.

The implementation of `poll` (Figure 7.7) is quite simple. If the bag has not been dumped yet, the thread is still active and `poll` returns `NONE`. Otherwise, the result should be available and the appropriate value is returned. (It is an invariant that if the bag has been dumped, the result should not be `Waiting`, so this should never raise `Thread`.)

We discuss two implementations of `sync`. The first implementation is the straightforward one which returns the result if the target thread has completed and blocks if not. We then discuss a version that eliminates the overhead of returning to the scheduler in the case when the target thread has not been stolen, similar to the fast clone of Cilk [54].

The standard implementation (Figure 7.8) first checks if the thread has been cancelled, and then checks whether this sync will cause a priority inversion. If both checks pass, the implementation then checks whether the bag has already been dumped. If so, the result can be returned immediately. Otherwise, the calling thread suspends after adding itself to the bag (if the bag is dumped in the meantime, the thread simply adds itself back to the deque). The thread will resume at line 21 when the target thread has completed. Before returning the result, we set the current depth of the processor to the appropriate fork depth of the continuation (one level deeper than the target thread or the calling thread, whichever is deeper).

If the target thread has not been stolen, we can perform a "fast-path" optimization by performing the work of the target thread locally rather than suspending and returning to the scheduler. The optimized implementation is shown in Figure 7.9. In this version, before suspending, we attempt to remove the target thread from the deque using its handle. If the removal succeeds, we know that the thread has not been stolen (and will not be), and it is safe to simply execute the thunk locally. We must still write the result and wake any readers, but can then immediately return the value. If removal fails, the thread has been stolen and we must fall back to the slow path, which is the same as in the previous version. Since steals are rare in most applications, we expect that the fast version will run most of the time.

```
fun sync {result, prio, bag, cancel, ...} =
    let val c = if Cancellable.isCancelled cancel then
                    raise Thread
                else
                    ()
        val p = processorNumber ()
        val r = curPrio p
        val _ = if Priority.ple (r, prio) then ()
                else
                    raise IncompatiblePriorities
        fun f rt =
            if Bag.insert (bag, rt) then
                (* Successfully waiting on the thread *)
                ()
            else
        (* Bag was just dumped, so we can directly add the thread *)
                ignore (push rt)
        val d = getDepth p
        val _ = if Bag.isDumped bag then ()
                else suspend f
    in
        case !result of
            Finished (x, d') => (setDepth (p, Int.max (d, d') + 1);
                                 x)
          | Raised (e, d') => (setDepth (p, Int.max (d, d') + 1);
                               raise e)
          | Waiting => raise Thread
    end

```

Figure 7.8: The implementation of sync.

```
fun sync {result, prio, bag, hand, cancel, thunk} =
    let val c = if Cancellable.isCancelled cancel then
                    raise Thread
                else
                    ()
        val p = processorNumber ()
        val r = curPrio p
        val _ = if P.ple (r, prio) then ()
                else
                    raise IncompatiblePriorities
        val _ = if tryRemove (prio, hand) then
                    (* execute the thunk locally *)
                    (writeResult result thunk;
                     (case Bag.dump bag of
                          NONE => raise Thread
                        | SOME l => List.app wake l))
                else
                    (* have to block on it *)
                    let fun f rt =
                            if Bag.insert (bag, rt) then
                                (* Successfully waiting on the thread *)
                                ()
                            else
                                (* Bag was just dumped, so we can
                                 * directly add the thread *)
                                ignore (push rt)
                    in
                        if Bag.isDumped bag then ()
                        else suspend f
                    end
        val d = getDepth p
    in
        case !result of
            Finished (x, d') => (setDepth (p, Int.max (d, d') + 1);
                                 x)
          | Raised (e, d') => (setDepth (p, Int.max (d, d') + 1);
                               raise e)
          | Waiting => raise Thread
    end

```

Figure 7.9: Optimized sync implementation.

7.2.1 I/O Library

Our ML threads library includes a library with blocking I/O functions that can be called without blocking the system-level scheduler thread. This library includes functions for console input, network operations, and a simple graphics library based on X11. Many of the functions simply call into equivalent functions in the SML Basis or third-party libraries. Most of the graphics functions, for example, are essentially non-blocking to begin with.

Other functions, such as `accept`, which blocks on a network socket until a connection is available and then accepts the incoming connection, require some intervention to prevent the entire scheduler thread from blocking. For these functions, we use an equivalent non-blocking version (for example, the SML Basis library also provides `acceptNB` which returns `NONE` if no connection is available). If the non-blocking call returns `SOME v`, then we immediately return `v`. Otherwise, we call `suspendIO` on the calling thread. Recall that `suspendIO` takes a predicate `f` to associate with the I/O-blocked thread which is used to check whether the I/O event is ready; for this purpose, we again use the non-blocking version of the I/O function.

Many of the functions in our I/O library are implemented with a wrapper `B_of_NB` which converts a non-blocking SML library function into a function which will suspend the calling user-level thread. The code for this function, shown in Figure 7.10, accepts the non-blocking I/O function as its argument `f` and builds a function `f'` to be passed to `suspendIO`. In addition to calling `f` and returning true if and only if the I/O is available, `f'` memoizes the result to avoid losing data and/or unnecessary calls. The recursive internal function `bnb_rec` then suspends the thread as long as `f'` returns false, before returning the memoized value (`bnb_rec` should call itself recursively at most once, assuming the thread is not prematurely woken up by the scheduler).

7.3 Front end

Our compiler modifies the parser and elaborator of ML5/pgh [89], which also extends Standard ML with modal constructs, although for a quite different purpose. Elaboration converts the PriML abstract syntax tree to a typed intermediate language, and type checks the code in the process. For the PriML extensions, elaboration proceeds broadly along the lines of the rules defined in Section 4.3. At the same time, the elaborator collects the priority and ordering declarations into a set of priorities and a set of ordering constraints (raising a type error if inconsistent ordering declarations ever cause a cycle in the ordering relation). It also collects the fairness declarations into a mapping from priorities to their fairness value.

For our purposes, the elaboration pass is used only for type checking. We generate the final ML code from the original AST (which is closer to the surface syntax of ML), so as not to produce highly obfuscated code. Before generating the code, the compiler passes over the AST, converting PriML features into calls to our threading library. Priority names and variables are converted into ordinary SML variables of type `Priority.t`. Priority-polymorphic functions become ordinary functions, with extra arguments for the priorities, and their instantiations become function applications. Commands and instructions become SML expressions, with a sequence of bound instructions becoming a let binding. Encapsulated commands become thunks

```
fun B_of_NB (f: 'a -> 'b option) (v: 'a) : 'b =
    let val (r: 'b option ref) = ref NONE
        fun f' () = (* memoized version of f *)
            (case !r of
                 NONE => (r := f v;
                          case !r of
                              NONE => false
                            | SOME _ => true)
               | SOME _ => true)
        fun bnb_rec () =
            if f' () then
                (* Ready to resume, return value *)
                case !r of
                    NONE => raise OS.SysErr ("Impossible", NONE)
                  | SOME c => c
            else
                (* Not ready, suspend until ready *)
                (suspendIO f';
                 bnb_rec ())
    in
        bnb_rec ()
        end

```

Figure 7.10: User-level blocking I/O from non-blocking I/O.

PriML	Translation
cmd[ρ] {m}	`fn () => translate(m)`
do(e)	`translate(e) ()`
spawn[$\rho;\tau$] {m}	`Thread.spawn (fn () => translate(m))`
sync e	`Thread.sync (translate(e))`
poll e	`Thread.poll (translate(e))`
cancel e	`Thread.cancel (translate(e))`
ret e	`translate(e)`
$x \leftarrow i; m$	`let val x = translate(i) in translate(m) end`
t cmd[ρ]	`unit -> translate(t)`
t thread[ρ]	`translate(t) Thread.t`
$\forall \pi : C.t$	`Priority.t -> translate(t)`
fun[$\pi_1 : C_1 \ldots \pi_n : C_n$] $f(x_1 \ldots x_m) = e$	`fun p1 ... pn x1 ... xm = translate(e)`

Table 7.1: Translation to Standard ML

(so as to preserve the semantics that they are delayed). The translations are summarized in Table 7.1.

The AST generated by the above process is then prefaced by a series of declarations which call `Priority.new` and `Priority.new_lessthan` to register all of the priorities and ordering constraints with the runtime, and bind the priority names to the generated priorities. If the programmer has written fairness declarations, the compiler also prefaces the program with a call to `Priority.installDist` using a function that searches for the given priority in the list of priorities for which the programmer gave fairness values. Because priorities are not totally ordered, we can't do substantially better than a linear scan through the list of fairness values, which must be performed whenever the fairness criterion is queried (this occurs each time a processor changes its primary priority). This linear scan is another part of the implementation, in addition to finding the highest priority at a given time, that introduces behavior linear in the number of priorities. Though this is not a substantial bottleneck in any of our benchmarks, it would be a prime target for future optimizations. The compiler finally generates Standard ML code from the AST, and passes it to `mlton-parmem` for compilation to an executable.

Chapter 8

Case Studies

This way, my lord; for this way lies the game.

Henry VI, Part III (IV.5.14)

8.1 Motion planning

The first benchmark we consider is a motion planner for a robot, which uses two planning algorithms, organized in a hierarchical fashion [73]. A thread at low priority runs a parallel version of the A* search algorithm to compute a coarse-grained plan consisting of a series of landmarks leading toward the goal. At medium priority, several instances of the Rapidly-exploring Random Trees (RRT) algorithm compute detailed paths to the nearest several landmarks. A high priority interaction loop polls the planning threads to update its understanding of the optimal path, and spawns new planning threads as the robot moves toward the goal. We test our planner in the context of a simulated environment using a map from a standard set of 2-dimensional motion planning benchmarks taken from video games [115]. The simulated robot has nonzero length and is able to turn and move forward and backward, giving it three degrees of freedom (x- and y-coordinates and rotation angle). We can also add obstacles that are not part of the map and which are invisible to the long-term (A*) planner. These obstacles will be found by the short-term planner when the robot nears the obstacles. The RRT algorithm will then find a way around the obstacle to the nearest reachable landmark.

Long-term A* Planner. The long-term planner uses a parallel variant of the A* search algorithm called PWSA* [40]. We modify PWSA* slightly to take advantage of our threading library, and describe our modified version here. The purpose of PWSA* and A* is to find the shortest path in a graph from a start vertex s to a goal vertex t. Like A*, PWSA* requires an admissible heuristic function $h(v)$ which (under-)estimates the distance from a vertex v to t. The algorithm maintains two values f and g for each vertex, where $g(v)$ is the shortest path found so far from s to v and $f(v) = g(v) + h(v)$. The A* algorithm proceeds by maintaining a priority queue of vertices and iteratively exploring the unexplored vertex with the lowest f value. When exploring a vertex, A* adds its unexplored neighbors to the queue (if not already in the queue) and "relaxes" each neighbor by recomputing its g and f values. For an admissible heuristic (that

is, one which conservatively estimates the distance to the goal), A* is guaranteed to terminate with an optimal path, but is naturally sequential.

The idea behind PWSA* is to trade some amount of optimality for parallelism. Our version of the algorithm achieves parallelism by having busy processors split their priority queues in half when the queue reaches a specified threshold size (200 in our implementation). The algorithm spawns a new thread to execute PWSA* on half of the priority queue. This new thread is then migrated to an idle processor by the scheduler. Each busy processor proceeds as in A* by attempting to explore the vertex in its queue with the lowest f value. However, because the vertex may have already been explored by another processor, the processor attempts to "claim" the vertex by performing an atomic compare-and-swap against a designated cell for that vertex. The first processor to claim the vertex explores it. Each vertex also has a global mutable cell to store its g value. When relaxing a vertex v, processors perform a "priority update" on this cell, an operation which stores the lowest g value seen so far.

Parallelism introduces the possibility of a sub-optimal solution because, unlike in the sequential setting, it is not guaranteed that a vertex is explored only when it has the globally minimum f value. Because the amount of sub-optimality increases with the number of queue splits, processors only split their queue and spawn a new thread when the total number of PWSA* threads in the system is less than the number of processors.

Short-term RRT Planner. The Rapidly-exploring Random Trees (RRT) algorithm [77] finds paths in continuous domains by building up trees rooted at the start vertex that explore the space. The algorithm operates in rounds in which it chooses a point x in the space. With some fixed probability p, it chooses x to be the goal and with probability $1 - p$, it chooses a random point in the space. The algorithm then attempts to expand the tree in the direction of x by adding an edge from the existing tree vertex v that is nearest to x. If extending an edge from v some fixed distance in the direction of x (using appropriate generalizations of "distance" and "direction" for the space, which may be high-dimensional) would not result in a collision, the edge is added to the tree and the process repeats.

We attempted to, as in PWSA*, trade optimality (paths in RRT are not guaranteed to be optimal to begin with) for parallelism by choosing multiple random points at each round and extending the tree in several directions simultaneously. This approach, however, induces a synchronization point after each round and so does not result in a very practical parallel algorithm. As such, we chose to use sequential RRT and parallelize by running several instances of RRT simultaneously.

Main Loop. The main event loop of the planner is responsible for keeping track of the planning threads, interfacing with the simulation to move the "robot" and replanning if it senses an obstacle. The loop runs at high priority. At a high level, the planner keeps track of the "current" long-term and short-term plans. There is continuously a collection of low-priority threads running PWSA* to generate a new long-term plan from the current position to the end goal. At the same time, there is a collection of N medium-priority threads (where N is a parameter to the algorithm) running RRT to generate new short-term plans from the current position to various points along the long-term plan. Because the long-term plan may collide with obstacles that are

not on the map, each RRT thread uses as its goal a different point along the long-term plan. For example, if the route to the first point on the long-term plan is blocked, the thread computing an RRT plan to that point will fail, but the thread computing an RRT plan to the next point might succeed. The main loop continuously polls the planning threads, keeping the best long and short term plans, and spawning new planning threads as the plans complete.

We now discuss the operation of the loop, pseudocode for which appears in Figure 8.1, in more detail. Throughout the simulation, the main loop keeps track of several pieces of data:

- The current long-term plan (`long_plan`)
- The current short-term plan (`short_plan`)
- The thread currently computing a long-term plan (`long_plan_thread`)
- A list of threads currently computing short-term plans (`short_plan_threads`)

Each iteration of the loop performs the following sequence of operations:

1. Get the current position of the robot. If we have reached the goal, terminate (lines 2-4).
2. Poll the PWSA* thread. If it has generated a new long-term plan, replace the current plan. Because the PWSA* thread is performing a large computation at low priority, the robot's position may have changed substantially since the planning began and we must attempt to reconcile the new long-term plan with the current short-term plan. If the current short-term goal is still a point along the long-term plan, the operation succeeds and removes any prefix of the long-term plan that has been completed. Otherwise, we must throw out our current short-term plan and compute a new one that is compatible with the long-term plan. (Lines 6-14.)
3. If we are nearing the first point on the long-term plan, remove it from the plan (lines 16-19).
4. Poll the RRT threads. If any have completed, remove them from the list and update the current short-term plan to the best available plan (the function `better_plan`, not shown, uses a combination of which plan is shorter and which plan gets closer to the goal). (Lines 21-25.)
5. If all RRT threads have completed, spawn new ones to compute paths to each of the first N points on the long-term plan (lines 27-29.)
6. If we are nearing the first point on the short-term plan, remove it from the plan (lines 31-34).
7. Move the robot toward the first point on the short-term plan, unless doing so would cause a collision or we do not currently have a short term plan, in which case stop (lines 36-41).

The planner currently runs in a simulator that tracks the current position of the robot within the map, moves it and stops it in real time as directed by the planner's event loop, and detects collisions. The planning loop also registers its current short-term and long-term plans with the simulation so that the simulation can display a visualization in real time showing the current position of the robot, and its short- and long-term plans. An example of this visualization is shown in Figure 8.2, with the current long-term plan in blue and the current short-term plan in red. Black areas are out of bounds, and green areas are trees (also considered to be obstacles to the planners). The current location of the robot is shown as a small black line (centered in the orange circle, which was added to the figure for visibility). The simulation allows the user

```
repeat:
  pos := get_robot_position ()
  if distance (pos, goal) < threshold:
    break

  new_long_plan := false
  case poll long_plan_thread:
    | SOME plan =>
        long_plan := plan
        long_plan_thread := spawn[lp_prio] (new_long_plan (pos, goal))
        if not (reconcile_plan long_plan (last_point short_plan)):
          short_plan := NULL
          reconcile_plan long_plan pos
    | NONE => skip

  case long_plan:
      h::t => if distance (pos, h) < threshold:
                long_plan := t
    | _    => skip

  for t in short_plan_threads:
    case poll t:
      | SOME plan => short_plan := better_plan short_plan plan
                     remove (short_plan_threads, t)
      | NONE => skip

  if short_plan_threads = []:
    short_plan_threads = {  spawn[sp_prio] (new_short_plan (pos, p))
                          | p in first(long_plan, N) }

  case short_plan:
    | h::t => if distance (pos, h) < threshold:
                short_plan := t
    | _    => skip

  case short_plan:
    | h::t => if would_collide pos h:
                robot_stop ()
              else:
                robot_move_toward h
    | _ => robot_stop ()
```

Figure 8.1: Pseudocode for the main planning loop.

to click to introduce new obstacles which are visible to the robot when it is performing its short term plans, but which are not taken into account by the long-term planner.

Performance evaluation The main purpose of this case study is to show a substantial application for which our threading and priority model is suitable, and give some evidence that our techniques can scale to examples of this size. We make no claims that we have developed a highly competitive or performant motion planner, or that the motion planner is a particularly suitable benchmark for testing the performance of our scheduler. Chapter 9 is devoted to a more substantial performance evaluation. Still, it is worthwhile to show some performance results to evaluate the suitability of our techniques for the case study.

The simulation code runs in the same thread as the main loop. In keeping with the ratio suggested by Knepper et al. [73], we ran the motion planning simulation with a fairness criterion that assigns four times as many cycles to the short-term planner as to the long-term planner. The interaction loop was given the same number of cycles as the short-term-planner. The number N of short term planning threads was set to be equal to the number of processors.

We ran the motion planner in parallel, using between 1 and 70 processors. Because the algorithm uses a substantial amount of randomization, experiments on the planner are necessarily fairly noisy. We ran the planner 9 times for each number of processors, and discarded runs where the robot became stuck and failed to reach the goal within 5 minutes. Each data point presented represents an average over the remaining runs.

Figure 8.3 shows the total time the simulated robot takes to reach the goal for each number of processors, measured from when the simulation is initialized and the initial PWSA* computation is started until the robot is within the threshold distance of the goal. The total time decreases sharply up to eight processors, but then begins to substantially increase starting at approximately 16 processors. We suspect that this pattern is the result of two competing factors. The initial decrease is most likely due to the increase in planning throughput enabled by parallelism. After this, however, the benefits of parallelism decrease and the tradeoff between optimality and parallelism take over. The robot appears to be traveling longer distances, canceling out the decrease in planning time.

The results indicate that it might be possible to improve the parallel performance of the planner by using parallelism to a greater extent to generate *better plans* rather than to simply speed up planning. For example, if more than eight processors are available, we could run multiple instances of PWSA* concurrently (each using up to between four and eight processors) and choose the best plan.

8.2 Real-time, human v. computer game

The other extended case study we consider in this chapter is a game in which a human player, interacting with the terminal, competes against a collection of computer-controlled players. The game is real-time (as opposed to turn-based; this gaming terminology should not be confused with real-time software), and so the software must react quickly to user actions in order to maintain the user experience. At the same time, five AI-controlled players simultaneously plan their strategy using a tree search algorithm, which itself is parallelizable. Because the game proceeds

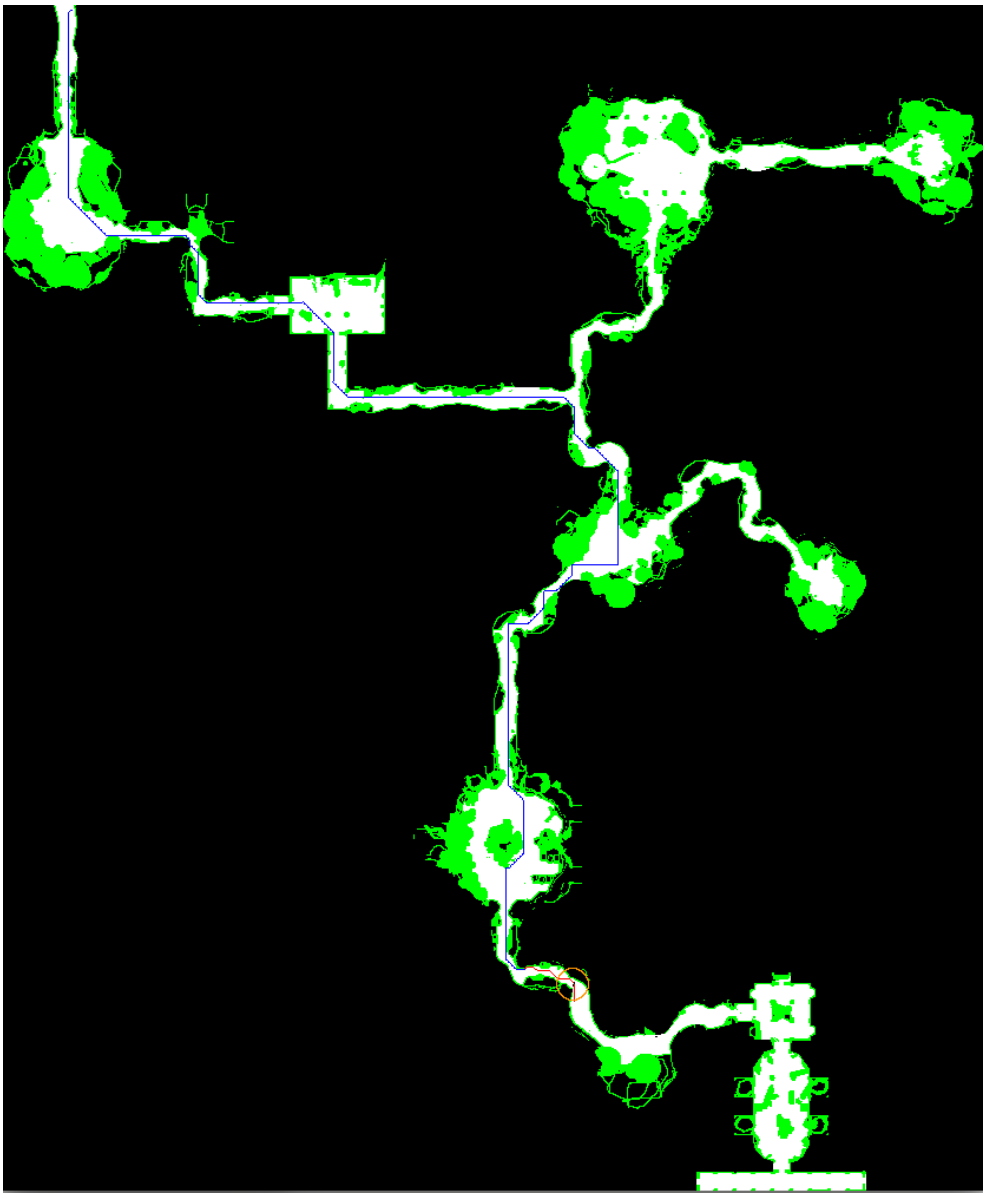


Figure 8.2: The motion planner.

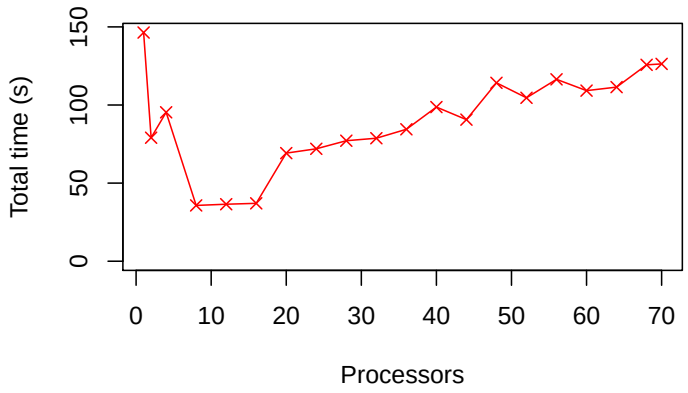


Figure 8.3: Total time to reach the goal.

in real time, with players taking actions as quickly as they can (or as quickly as the user interface will allow), the AI players benefit by being able to compute their strategy as quickly as possible.

The game. The storyline of the game is not important for understanding its relevance as a case study, but we include it here for completeness. In the game, the user plays as a student defending a thesis, and the AI players are the thesis committee. The student has two health bars, Knowledge and Stamina, which allow the user to take certain actions. The committee members also have two health bars, Patience and Confusion. Confusion will result in the committee members asking questions, to which the student must respond in a fixed amount of time. Failure to do so decreases the committee's Patience. If any committee member's Patience drops to zero, the student player loses. The student wins by covering a fixed amount of the talk before the committee runs out of Patience. A screenshot of the game appears in Figure 8.4.

Monte Carlo Tree Search. As in most game-playing AIs, the AI players work by searching through the *game tree*, a representation of the possible outcomes of the game in which the branches are moves that can be taken by each player, the nodes are game states and the leaves are evaluations of the final outcome of the game: either a score, or simply which player won. In most games, the game tree is too large to search completely. Monte Carlo Tree Search (MCTS) has been successfully used recently to develop AI players for games with very large state spaces, notably Go (e.g., [36, 95]).

In MCTS, the game tree is sampled randomly to find statistically good moves for the AI player. While playing the game, and simulating possible ways the rest of the game could play out, the AI tracks, for each explored game state, the number of simulated games played starting at that state and the number that resulted in wins ("play" and "win" counts). For each simulated game, MCTS explores the game tree starting at the current game state. The simulated game begins with a *selection* phase when MCTS is exploring nodes that have already been explored. During this phase, MCTS already has information about the probability of winning for each move it could take on this turn, so it selects which move to take (in the simulated game) using this information, according to some algorithm. We use the Upper Confidence Bound on Trees

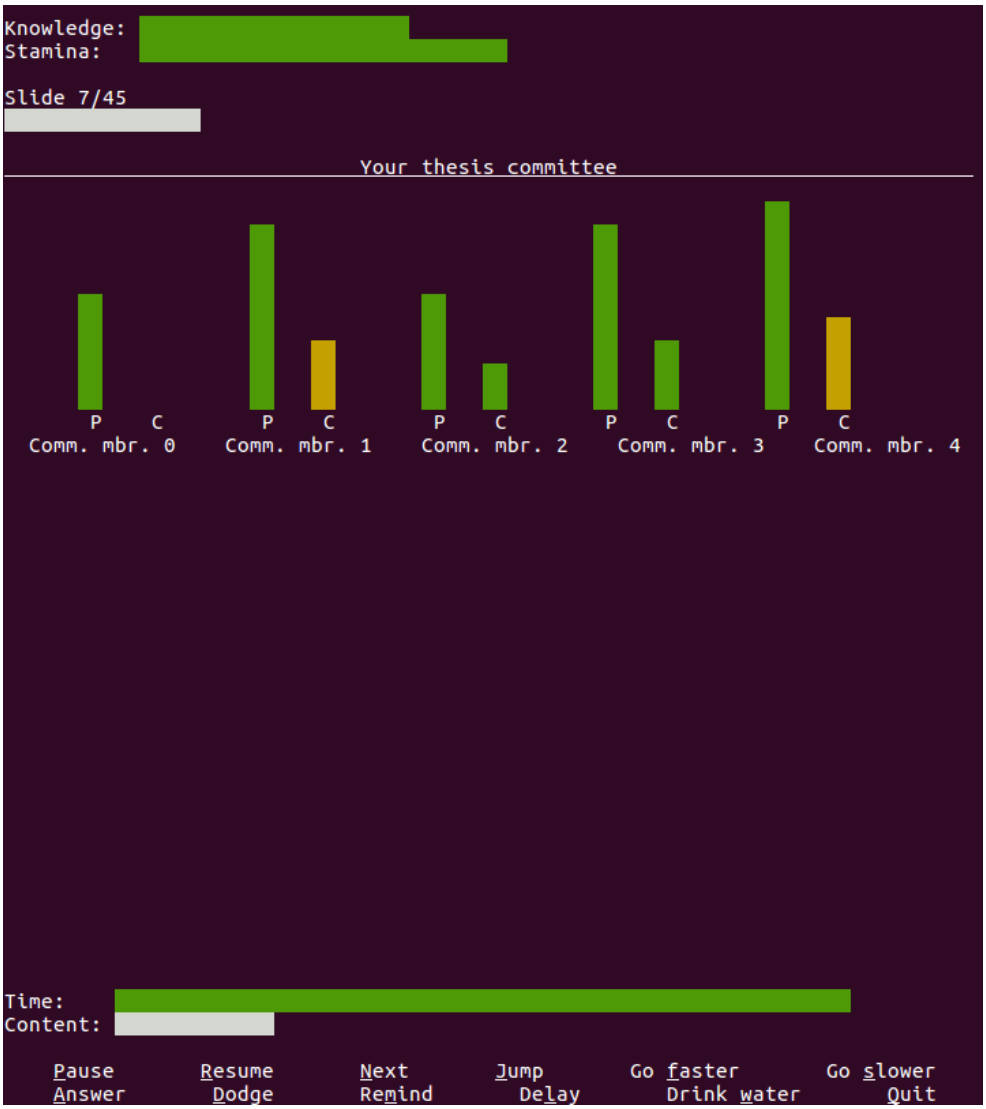


Figure 8.4: The human v. computer game.

(UCT) algorithm [74]. Once the search reaches the frontier of where it has already explored, it *expands* the stored game tree by adding a new node and initializing its play and win counts to zero. MCTS then enters the *simulation* phase in which it plays one or more simulated games at random (we choose to play one), starting at the new node. Finally, depending on the outcome of the simulated game, it updates the play and win counts of every stored node along the traversed path, including the newly added node. The entire process is repeated as many times as desired before the AI chooses its next actual move.

Several approaches to parallelizing MCTS have been explored in the literature (e.g., [8, 29, 44]). We chose to implement *single-run* or *root* parallelism [29], in which multiple separate instances of MCTS run simultaneously and then combine their play and win counts at the end. This version of MCTS is straightforward to implement and is a natural fit for dynamic parallelism, but the lack of sharing between MCTS instances might slightly impact the results. Because MCTS uses information from previous simulations to inform the selection phase, each parallel run of MCTS has less information than it would if the runs were sequential. As in the PWSA* algorithm of the previous section, we are trading parallelism for optimality. This seems like an especially reasonable tradeoff in probabilistic search algorithms which are, by their nature, not optimal.

The main loop. The game consists of one interactive thread, controlling the terminal interaction with the user, and five collections of computation threads, controlling the AI players. Both human and AI players implement a function `move` which takes the current game state and player state, and optionally returns the player's move as well as the player's new state. If the player is not ready to move, `move` returns `NONE`. The player state includes any state the player needs to make its move (e.g., the play and win counts for the AI players, which are preserved between MCTS runs), and is passed to and returned from `move` in order to preserve a functional interface. The AI players' `move` function runs 1,000 runs of MCTS as described above and then chooses the best available move based on the existing play and win information for the accessible game states. The student player's `move` function displays a representation of the game state on the terminal and polls the keyboard to see if the user has entered a move since the last call.

The main loop of the game is quite simple: it maintains the state of the game, and calls `move` for each player at a fixed frame rate, updating the game state accordingly. Because `move` for an AI player may take a substantial amount of time, the loop spawns a vector `cthreads` of threads (each running at separate, incomparable, priorities), one for each AI player, and polls the threads at each frame, making the player's move each time one is available. Pseudocode for the game's main loop, which runs at a priority higher than that of the AI threads, is shown in Figure 8.5.

Qualitative performance evaluation. We did not attempt a thorough performance evaluation of the game, but we did make some qualitative observations during testing. When the game is run on one processor, the AI players are slow to respond: although their confusion levels get high enough to ask questions, they take longer to do so. This effect is less noticeable when the game is run on more processors. It appears that the parallelism among AI players (the fact that each of the five AI players can plan their strategies independently in parallel) is more beneficial than the parallelism within each AI player. Parallelizing each MCTS computation may be profitable to a point, but we did notice that performance began to degrade when each MCTS thread performed

```
state = initial_state ()
aistate = {initialize_ai i | i in committee_mbrs}
humstate = initialize_human ()
cthreads = {spawn[aiprio_i] (AI.move (state, aistate[i]))
             | i in committee_mbrs}

repeat:
  for each i in committee_mbrs:
    case cthreads[i]:
      | SOME (move, aistate') =>
          state := update (state, move)
          aistate[i] := aistate'
          cthreads[i] = spawn[aiprio_i] (AI.move (state, aistate[i]))
      | NONE => skip
  case Human.move (state, humstate) of
    | SOME (move, humstate') =>
        state := update (state, move)
        humstate := humstate'
    | NONE => skip
  sleep(frame_interval)

```

Figure 8.5: The main loop of the game.

fewer than 100 simulations. It would be interesting to, in future work, run the game on more processors and attempt a more quantitative evaluation of the benefits of parallelism.

Chapter 9

Evaluation

> And here I stand: judge, my masters.
>
> *Henry IV, Part I* (II.4.454)

Most of this chapter discusses an empirical evaluation of the runtime of PriML. The purpose of this evaluation is primarily to determine how well the scheduling framework approximates the fairly prompt scheduling principle. To this end, we evaluate the scheduler on three main criteria:

1. **Fairness:** Does the fraction of processor cycles devoted to work at each priority match the fairness criterion provided to the scheduler?
2. **Promptness:** When work at other priorities is scarce, does high priority work get handled quickly and effectively?
3. **Greediness:** Does load balancing allow threads to effectively utilize the processor cycles available to them?

A schedule that perfectly meets all of the above criteria is fairly prompt by the definition established in Section 3.4, and therefore obeys the cost bounds we have shown.

We evaluate the runtime on two sets of benchmarks: larger "application benchmarks" which we use to show the power and flexibility of the system, and smaller "orthogonal benchmarks" which provide for more controlled experiments. Because the goal is to evaluate the performance of the runtime rather than the speed of the code generated by the compiler, these benchmarks are written in Parallel ML using calls to our threading library. We justify this choice in Section 9.3.4 by evaluating one of our benchmarks written in PriML against the handwritten Parallel ML version and showing their performance to be comparable.

At the end of this chapter, we present another set of "expressiveness" benchmarks, which are written in PriML and can be compiled using our prototype compiler. We do not present a performance evaluation of these benchmarks, but include them in this thesis to show a variety of uses of the language, as well as the practicality of our compiler front-end.

We begin with a description of our experimental setup.

9.1 Experimental setup

All experiments were run on a machine with 1TB of main memory and 72 2.4GHz Intel Xeon processors. Unless otherwise noted, we ran our experiments on 70 processors, reserving two for the experiment scripts and other background processes. In the version of Parallel MLton we use, garbage collection is sequential and stops all processors, so for all execution times reported, we subtract the time spent on GC. We *do not* subtract GC time from response times.

Evaluation framework For testing many of our interactive benchmarks, we use a driver program, written in C, which shares pipes and network connections with the benchmark application. The driver simulates interactions as specified by an "event trace" consisting of a sequence of interactive actions, encoded in a domain-specific language. The benchmarks are instrumented to produce predictable outputs on standard output when the program responds to a particular interactive action. For example, when the user advances the photo in a slideshow and the photo viewer displays the photo, it also outputs a log message indicating the newly displayed photo. When the driver initiates an event, it adds an entry in a buffer with the start time and expected response. A separate thread in the driver reads the output of the program and checks it against the buffer entries. When a line of output matches the expected response for an event, the thread records the time difference and uses it to compute the average response time.

9.2 Application benchmarks

9.2.1 Web server

Our simple web server accepts incoming HTTP connections using one high-priority thread per processor. The server spawns a medium-priority thread for each new connection, which serves HTTP requests on that connection. Each request is logged, and a low-priority background computation periodically processes the logs and computes usage statistics [1]. In our tests, we gave accepting threads, serving threads and background threads 4/9, 4/9 and 1/9 of cycles, respectively.

We evaluated the web server using ApacheBench (`ab`), running on another machine on the same Local Area Network. We varied two parameters: the number of concurrent connections C and the number of processors running the server. In each trial, `ab` opens C connections to the server and performs 2,000 total HTTP requests. Table 9.1 reports the mean response times, in milliseconds, for each experiment. ApacheBench computes the mean response time end-to-end: it records the total time to complete the benchmark, and divides by the number of requests (2,000 in this case).

As expected, for a fixed number of processors, the response time increases as the degree of concurrency becomes higher, since this puts a great deal more stress on the scheduler and the server threads. For a fixed value of C, the response time appears to initially decrease as more processors are added. This indicates that the additional processors are effectively balancing the

[1] Since, in all of our tests, the logs remain small, we inflate the work of the background process by having it periodically perform a large Fibonacci computation in addition to tabulating page views.

	Processors		
C	1	10	50
1	1.1	1.1	1.3
10	8.2	1.6	1.8
50	333.4	40.0	6.5

Table 9.1: Mean response time (ms) for the web server.

load of the concurrent requests. When the number of processors exceeds C, however, response time increases slightly, suggesting that when there is not enough interactive work to keep all processors busy, load balancing adds a small amount of overhead.

9.2.2 Photo viewer

The photo viewer benchmark displays JPEG images from a local folder, and allows the user to scroll forward and backward, and to jump to images out of order. When a new image is selected, the next eight images in order are decoded by low-priority threads. The interaction with the user occurs in a high-priority foreground thread. When the user selects a new image (either by scrolling or jumping), the foreground thread checks if the image has already been decoded and either displays the decoded image or decodes it and displays it immediately.

The third-party libraries we use for decoding and rendering images are not thread-safe, and so we are only able to run the photo viewer on one processor, using threads for latency hiding rather than parallelism. Still, latency hiding can be quite beneficial in this benchmark, if the thread waiting for user input properly yields to the background decoding threads, and these background threads properly yield to each other. Thus, even on one processor, this benchmark provides a good test of our scheduler's fairness and promptness. In our tests of the photo viewer, we directed the driver program to request a series of 10 images, out of 100 total images, each of which is 4.3MB in size and takes approximately 225ms to load from file and decode. Figure 9.1 shows response times for four interaction rates (1, 2, 5 and 10 requests per second) and three types of traces: in-order access, random access and a mix of the two. The y-axis is the response time on a log scale, measured from when the driver requests the image to when the decoded image is displayed on the screen. For in-order traces where requests are made slowly enough to use pre-decoded images, average response times are under 20ms. These response times show that the interaction thread properly donates its spare cycles to decoding, and quickly wakes up and responds to interaction when necessary. As expected, the benefit of pre-decoding images decreases when viewing the photos in a random order or scrolling faster than the viewer can decode.

9.3 Orthogonal benchmarks

To gain a more controlled experimental setting for doing a thorough quantitative evaluation of the scheduler, we use a set of *orthogonal benchmarks*, in which computations at different

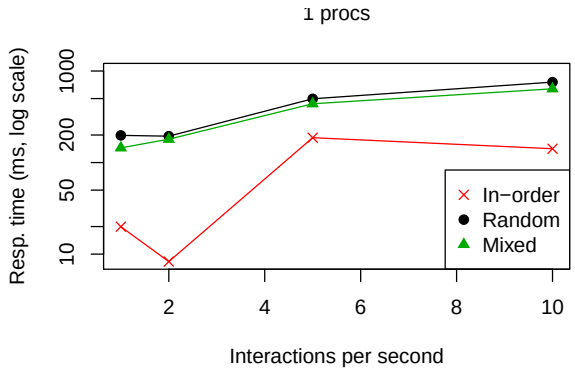


Figure 9.1: Response time results for the photo viewer.

priorities do not spawn or synchronize with each other, allowing us to tune them independently.

Each of our orthogonal benchmarks consists of some combination of *compute components* and *interactive components*. Each of the components operates independently of the others, and they may all be run at any priority. All of the compute components and one of the two interactive components are themselves parallel. The set of components was chosen to cover both predictable workloads, where the parallel structure is relatively static, and unpredictable workloads, where the parallel structure is determined dynamically during computation. For the compute components, we also consider both compute-intensive and memory-intensive workloads.

Interactive components:

- **Terminal echo.** The component repeatedly accepts a line of text on standard input and echoes it back to the user on standard output. This component is sequential and predictable.
- **Network echo.** The component listens for network connections in a main, single-threaded loop. For each connection, the loop spawns a new thread to handle this client. Each such thread interacts with the client as in terminal echo above, accepting a line of text over the socket and echoing the text on standard output. This component is parallel and unpredictable, since the parallel structure depends on the incoming connections.

Compute components:

- **Fibonacci.** A naturally parallel recursive algorithm computes the 45^{th} Fibonacci number. While this is not an efficient algorithm for computing Fibonacci numbers, it generates many parallel tasks quickly with little to no allocation, making it an excellent representation of a computation-intensive, predictable parallel workload.
- **Unbalanced Tree Search (UTS).** UTS [94], designed as an adversarial test of load balancing, explores a tree where, at each node, the number of children is randomly selected from a geometric distribution. This results in a high degree of imbalance, stressing the scheduler's distribution of work. Typically, the number of children of a node is computed based on a SHA-1 hash of a representation of the parent. We instead use an implementation of the DotMix deterministic parallel random number generator [80], which is based upon the same principles. The tree we use has depth 11 and average branching factor 4.0.

Since our random number generation is fairly inexpensive, we add a small sequential Fibonacci computation at each node to increase the work per node. This benchmark is highly unpredictable and computation-intensive.

- **Dense Matrix Multiplication (DMM).** This is a parallel, recursive implementation of Strassen's algorithm for matrix multiplication, which we use to multiply two 2048×2048 matrices. This benchmark is still more compute-intensive than memory-intensive, and also results in a fairly predictable parallel structure, but is less artificial than Fibonacci.
- **Breadth-First Search (BFS).** This benchmark performs a breadth-first search of a randomly generated graph with 15M nodes and 240M edges. The graph is stored using a compact, array-based adjacency list representation. This benchmark is memory-intensive, with a somewhat unpredictable parallel structure based on the graph structure.
- **Sample Sort.** Our implementation of sample sort is based on that of Blelloch et al. [20], and is highly optimized to make maximum use of parallel processing and hierarchical caches. As a benchmark, it is memory-intensive but with a more predictable structure than BFS. The benchmark sorts a random sequence of length approximately 128M.

9.3.1 Measuring Fairness

To show that our scheduler meets the first of the three criteria, *fairness* (that is, that it executes a computation in the expected amount of time given the percentage of cycles devoted to the computation), we use a set of benchmarks constructed with three priorities. The top priority runs an interactive component, and the bottom priority runs a compute component. The middle priority acts as a time sink. It runs a very large, massively parallel computation which is not expected to finish or idle during the experiment. It will therefore collect any cycles donated by the top priority, allowing us to view the low-priority computation in isolation. Each benchmark runs until the low-priority computation completes, and then terminates.

We ran each combination of compute and interaction components with three different fairness criteria, written in the form "H-M-L", where H is the percentage of cycles devoted to the high priority, M to the middle priority and L to the low priority. For this evaluation, we used 0-0-100, 50-0-50 and 50-25-25. In each run, the driver program interacted with the interactive component 50 times per second. We also ran each compute component as a serial program, compiled with standard MLton. We refer to this as the *serial baseline* and denote its running time by T_s. The ratio between a parallel execution time and the serial baseline is the *speedup* and is generally used as an evaluation of how greedy a scheduler is, that is, how well it distributes work across processors. To evaluate fairness, how close the scheduler stays to the prescribed fairness criterion, we use as a baseline the 70-processor execution time for 0-0-100, which we denote T^b_{70}.

Table 9.2 gives the two baselines and the speedup for 0-0-100 (T_s/T^b_{70}). For 50-0-50 and 50-25-25, we show the 70-processor execution time T_{70} in seconds, the average response time (RT) of the interaction in milliseconds and the *normalized speedup* on 70 processors. The normalized speedup is calculated as T_s/LT_{70}, where L is the fraction of cycles devoted to the computation: 0.5 for 50-0-50 and 0.25 for 50-25-25. This normalization is done because, for an execution time of T_{70}, the scheduler is expected to spend only LT_{70} on the low-priority computation. The

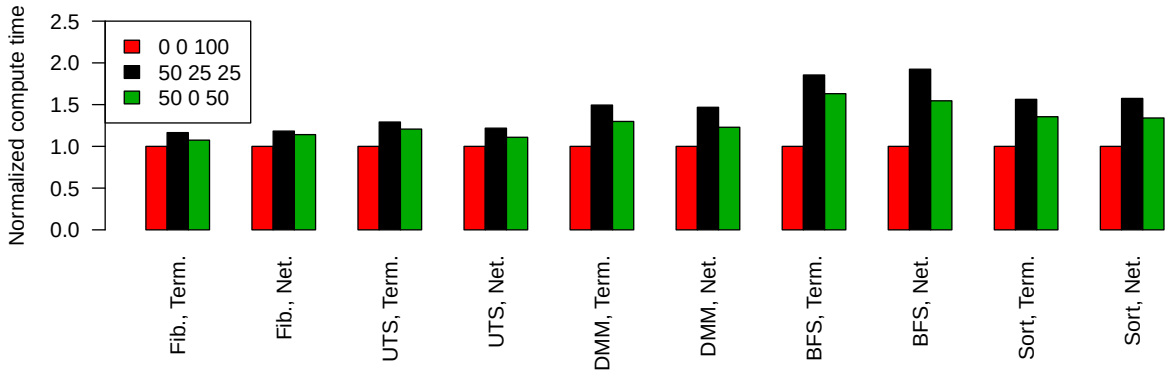


Figure 9.2: Normalized execution times for the low-priority computation.

maximum normalized speedup in all cases is 70x on 70 processors.

We can judge how well the scheduler distributes work by looking at the speedup, but the speedups shown in the table are not very meaningful without a comparison to another scheduler, which we provide in Section 9.3.2. We judge how well the scheduler respects the fairness criterion by looking at the ratios T^b_{70}/LT_{70} between the normalized execution times and the 0-0-100 baseline. If the scheduler is being perfectly fair, these ratios should be 1.0, i.e., the 50-0-50 execution time should be $2T^b_{70}$ and 50-25-25 should be $4T^b_{70}$. These ratios are shown graphically in Figure 9.2. For the compute-bound benchmarks (Fibonacci, UTS and DMM), the ratios stay very close to 1. For the memory-bound benchmarks, the ratios go as high as 2.0. These deviations from fairness are reasonably tolerable, but may merit further study.

We can also use these benchmarks to judge how responsive the programs are by looking at the mean response times, reported in Table 9.2 for the 50-0-50 and 50-25-25 criteria. Response times for 0-0-100 are quite poor: they range as high as 800ms, and in many cases, events are dropped entirely, as is to be expected when no cycles are devoted to the interaction (though, because of promptness, idle processors will work on high-priority tasks). Response times are otherwise quite good, remaining well under 10ms except for the very intensive Sample Sort computation (and the breadth-first search for 50-25-25). To test how many cycles the interaction needs and how this varies with interaction rate, we also measured the response times for 10 and 100 interactions per second, and for fairness criteria 25-25-50 and 5-45-50. The results for 100 terminal interactions per second, which are representative of the other results, are shown in Figure 9.3. As expected, mean response times are higher when very few cycles are devoted to interaction, though they mostly remain under 50ms, showing that for applications where responsiveness is not critical, few cycles need to be devoted to handle infrequent bursts of interaction.

9.3.2 Evaluating Speedups

As discussed above, the three-priority benchmarks provide a good measure of fairness: how well the scheduler respects the fairness criterion. To show how effectively the scheduler uses the cycles devoted to a parallel computation, we compare the parallel speedup of computation-only benchmarks to the parallel speedup of the same computation on a work-stealing scheduler

			0-0-100		50-0-50			50-25-25		
High prio.	Low prio.	T_s (s)	T_{70}^b (s)	Spd.	T_{70} (s)	Spd.	RT (ms)	T_{70} (s)	Spd.	RT (ms)
Terminal	Fibonacci	10.49	0.47	22.5×	1.00	20.9×	7.3	2.17	19.3×	8.2
	UTS	4.99	0.40	12.5×	0.96	10.4×	4.1	2.06	9.7×	6.0
	DMM	6.79	0.57	11.9×	1.48	9.2×	5.2	3.41	8.0×	7.5
	BFS	7.60	0.84	9.0×	2.75	5.5×	4.1	6.25	4.9×	12.6
	Sample Sort	30.61	1.07	28.7×	2.89	21.2×	32.1	6.67	18.3×	27.4
Network	Fibonacci	10.49	0.42	25.0×	0.96	21.9×	6.5	1.98	21.1×	8.6
	UTS	4.99	0.42	12.0×	0.92	10.8×	4.4	2.03	9.8×	7.7
	DMM	6.79	0.56	12.2×	1.37	9.9×	4.8	3.28	8.3×	7.2
	BFS	7.60	0.87	8.7×	2.69	5.7×	4.0	6.70	4.5×	18.8
	Sample Sort	30.61	1.07	28.5×	2.88	21.3×	12.8	6.76	18.1×	12.6

Table 9.2: Execution and response times for three-priority benchmarks.

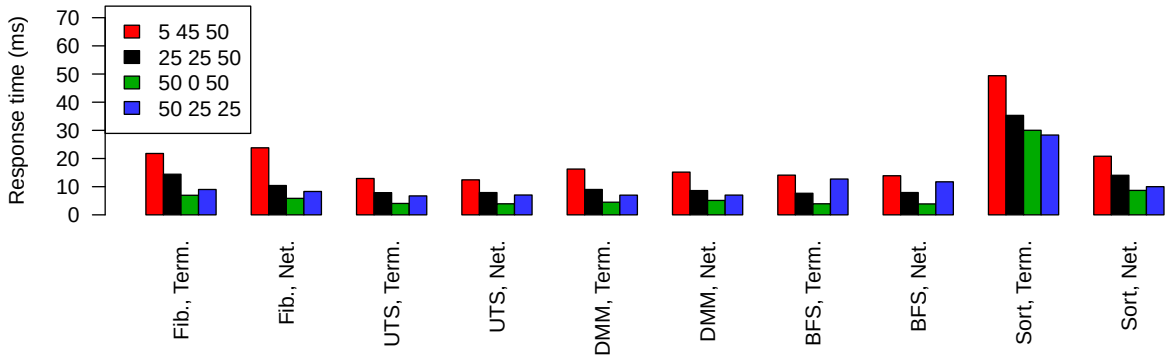


Figure 9.3: Response times for three-priority benchmarks.

	32 procs			70 procs		
Comp.	SWS	PPD	Rat.	SWS	PPD	Rat.
Fib.	16.7×	14.1×	1.18	33.5×	26.8×	1.25
UTS	13.4×	11.5×	1.16	26.0×	17.8×	1.46
DMM	8.6×	10.2×	0.85	13.7×	17.3×	0.79
BFS	18.7×	12.5×	1.50	36.2×	15.0×	2.41
Sort	29.8×	25.3×	1.18	61.7×	43.3×	1.42

Table 9.3: Parallel speedup of computation-only benchmarks.

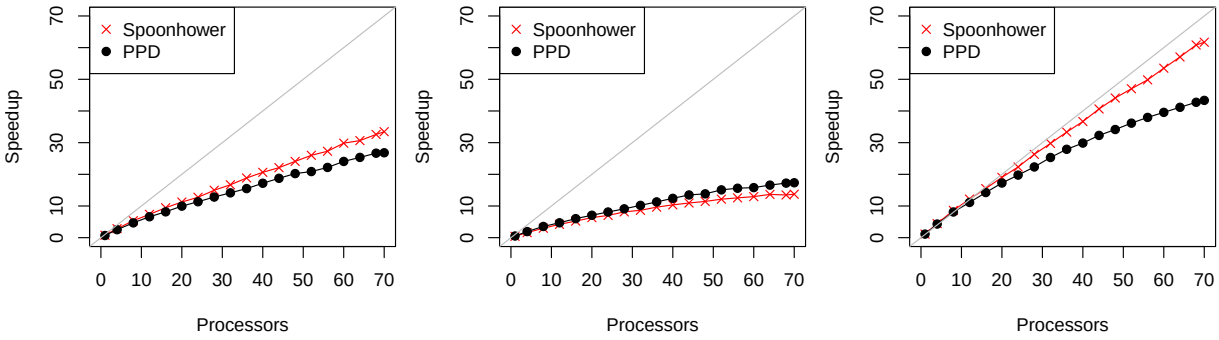
which was specially designed to obtain good speedups on purely computational benchmarks. The scheduler we use for comparison is Spoonhower's original work stealing implementation from his Parallel MLton.

These speedups, shown in Table 9.3, need not be normalized since there is only one computation, which is given all of the cycles. Our scheduler is competitive with Spoonhower on most benchmarks, and even performs substantially faster on DMM. At 70 processors, our performance begins to decline for BFS and Sample Sort. Speedup curves are shown in Figure 9.4. These curves show the parallel speedup versus number of processors of both schedulers. The light gray line indicates perfect speedup.

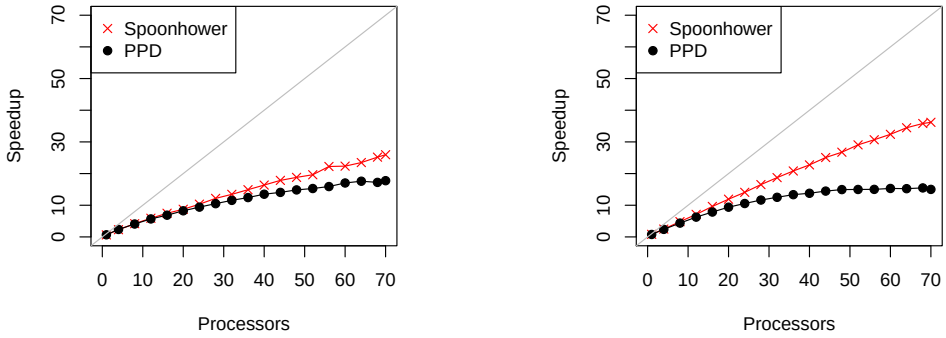
9.3.3 Measuring Promptness

Promptness requires that when work at the primary priority is unavailable, the processor works on the highest available priority. We show that this is the case using benchmarks with one high-priority interactive component and one low-priority compute component. We run these benchmarks with a winner-take-all policy; that is, the fairness criterion assigns all cycles to the top priority and low-priority computation runs only when high-priority work is unavailable. All of these experiments use an interaction rate of 50 interactions per second.

Table 9.4 shows the parallel speedups and response times for each benchmark. As above, the speedups are not normalized because we wish to determine what fraction of cycles are given to



(a) Fibonacci (b) DMM (c) Sample Sort



(d) UTS (e) BFS

Figure 9.4: Speedup on computation-only benchmarks.

Int.	Comp.	1 proc resp. time (ms) PLDI '17	PPD	70 proc Spd. PLDI '17	PPD	Rat.
Term.	Fib.	2.5	2.6	25.9×	29.2×	0.88
	UTS	2.6	2.6	15.8×	18.6×	0.85
	DMM	2.6	2.5	19.2×	18.9×	1.01
	BFS	187.2	2.7	12.6×	16.7×	0.75
	Sort	17.1	2.5	35.0×	44.8×	0.78
Net.	Fib.	2.6	2.6	25.4×	27.8×	0.92
	UTS	2.6	2.6	16.7×	16.2×	1.03
	DMM	2.6	2.6	18.9×	17.4×	1.09
	BFS	10.0	2.6	12.0×	14.9×	0.80
	Sort	3.5	2.6	34.4×	42.9×	0.80

Table 9.4: Speedup and response time of two-priority, winner-take-all benchmarks.

the computation; ideally almost all execution time will be devoted to the low-priority computation. The performance of our scheduler in these experiments is comparable to its performance in Table 9.3, indicating that only those cycles that are necessary to complete the interaction are devoted to the interaction, as required by promptness.

We also compare the performance of the PPD algorithm on these two-priority benchmarks to the same benchmarks running with our specialized two-priority scheduler from our PLDI 2017 paper [87]. The PLDI 2017 scheduler maintains two deques per processor, one for each priority. Upon deal attempts, a processor p picks a target q and sends a high-priority task if p has one and q does not, or a low-priority task if p has no high-priority tasks and q is idle. Because there is no notion of fairness, processors always work on high-priority threads when possible. The primary algorithmic difference, then, between the PLDI '17 scheduler and PPD on a winner-take-all benchmark with two priorities is that PPD will send a low-priority task to a processor that has high-priority tasks while PLDI '17 will not. Because the two schedulers are similar other than the added features of PPD (which also has to account for fairness and a more complex priority order), the comparison between PPD and PLDI '17 serves to highlight the cost of these added features: if PPD is competitive with PLDI '17, then the cost of adding fairness and a complex priority ordering is manageable. In fact, in all benchmarks, our scheduler essentially matches or out-performs the specialized scheduler in both response time and speedup. This shows that the special-purpose data structures and performance optimizations of the PPD algorithm have made up for the cost of the added complexity.

9.3.4 Measuring Impact of the Front End

As mentioned earlier, the benchmarks used in this section were all written in Parallel ML with calls to our threading library (the target of the PriML compiler). This decision allows us to effectively measure the performance of the runtime scheduler without introducing any inefficiencies in the front end. It also allows us to use features of SML, such as its module system, which are

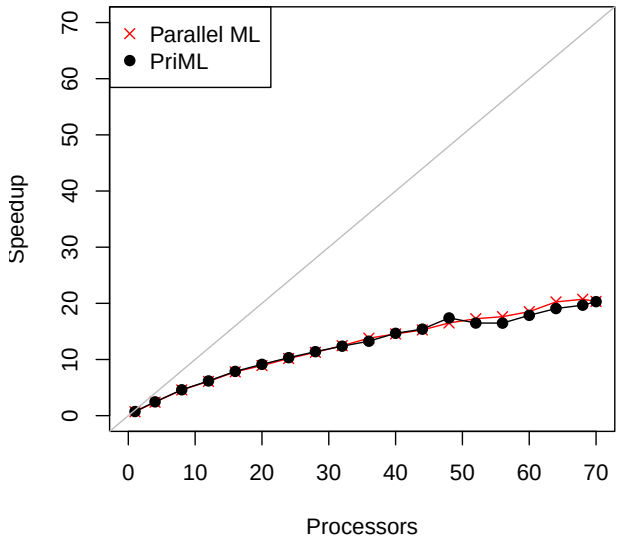


Figure 9.5: Speedup curves for Fibonacci-terminal on PriML and Parallel ML.

convenient for designing large-scale experiments but are not currently supported by PriML.

Here, we give some evidence that the decision to test benchmarks written directly in Parallel ML does not substantially change the results. Figure 9.5 shows the normalized speedup of the Fibonacci-Terminal benchmark with fairness criterion 50-25-25, for both the Parallel ML version of the benchmark evaluated earlier in this chapter and a version of the benchmark written in PriML and compiled using the compiler of Chapter 7. The performance characteristics of the two implementations are effectively identical.

9.4 Expressiveness

We have implemented five sizable programs in PriML. These include the email client of Section 4.1, a version of the web server of Section 9.2.1 (the version evaluated above was implemented in ML using the threading library) and a bank example inspired by an example used to justify partially ordered priorities [9]. We have also adapted the Fibonacci server and streaming music benchmarks of our PLDI 2017 paper [87]. These originally used only two priorities; we generalized them with a more complex priority structure, and implemented them in PriML.

Email Client We have implemented the "email client", portions of which appear in Section 4.1. The program parses emails stored locally, and is able to sort them by sender, date or subject, as requested by the user in an event loop at priority `loop_p` (which currently just takes the commands at the terminal; we don't yet have a graphical interface). The user can also issue commands to send an email (stored as a file) or quit the program.

Bank Simulator Babaoğlu et al. [9] give the example of a banking system that can perform operations *query*, *credit* and *debit*. To avoid the risk of spurious overdrafts, the system prioritizes

credit actions over debit actions, but does not restrict the priority of query actions. We implement such a system, in which a foreground loop (at a fourth priority, higher than all of the others), takes query, credit and debit commands and spawns threads to perform the corresponding operations on an array of "accounts" (stored as integer balances).

Fibonacci Server The Fibonacci server runs a foreground loop at the highest priority `fg` which takes a number n from the user, spawns a new thread to compute the n^{th} Fibonacci number in parallel, adds the spawned thread to a list, and repeats. The computation is run at one of three priorities (in order of decreasing priority): `smallfib`, `medfib` and `largefib`, depending on the size of the computation, so smaller computations will be prioritized. When the user indicates that entry is complete, the loop terminates, prints a message at priority `alert` (which is higher than `smallfib` but incomparable with `fg`), and returns the list of threads to the main thread, which syncs with all of the running threads, waiting for the Fibonacci computations to complete (these syncs can be done safely since the main thread runs at the lowest priority `bot`).

Streaming Music We simulate a hastily-monetized music streaming service, with a server thread that listens (at priority `server_p`) for network connections from clients, who each request a music file. For each client, the server spawns a new thread which loads the requested file and streams the data over the network to the client. The priority of this thread corresponds to the user's subscription (the free Standard service or the paid Premium and Deluxe subscriptions). Standard is lower-priority than both Premium and Deluxe. Due to boardroom in-fighting, it was never decided whether Premium or Deluxe subscribers get a higher level of service, and so while both are higher than Standard, the Premium and Deluxe priorities are incomparable. Both are lower than `server_p`. This benchmark is designed to test how the system handles multiple threads performing interaction; apart from the asynchronous threads handling requests, no parallel computation is performed.

Web Server As described in Section 9.2.1, the web server listens for connections in a loop at priority `accept_p` and spawns a thread (always at priority `serve_p`) for each client to respond to HTTP requests. A background thread (priority `stat_p`) performs the parallel background computation. Both `accept_p` and `serve_p` are higher-priority than `stat_p`, but the ordering between them is unspecified.

9.5 Comparison to Other Approaches

We are aware of no other systems that offer all of the features of PriML (e.g., dynamic fine-grained parallelism, latency hiding, prioritized work stealing with more than two priorities) and so it is difficult to perform direct apples-to-apples comparisons with existing systems and approaches. In this section, we compare the performance of our threading library and runtime system to several alternate systems and approaches, each of which provides a subset of the features of PriML. Because they provide fewer features, these other approaches may somewhat out-perform our approach on some metrics for the more specialized benchmarks for which they

Procs		PPD	Cilk	Cilk (rel. to PPD)	Go	Go (rel. to PPD)
1	Comp. time	14.4 s	—	—	8.3 s	0.6
	Resp. time	2.6 ms	—	—	4635.9 ms	1779.8
8	Comp. time	2.2 s	9.6 s	4.4	1.3 s	0.6
	Resp. time	2.2 ms	2528.1 ms	1170.1	752.0 ms	348.1
32	Comp. time	0.7 s	9.6 s	13.3	0.4 s	0.5
	Resp. time	2.1 ms	2783.0 ms	1302.7	—	—
70	Comp.time	0.4 s	9.8 s	22.6	0.3 s	0.7
	Resp. time	2.0 ms	2834.8 ms	1388.0	—	—

Table 9.5: Comparison to Cilk and Go (Fibonacci-terminal benchmark).

were designed. Still, the fact that our system is competitive with these existing approaches shows that the cost of the additional features provided by PriML is not overly burdensome.

We first (Section 9.5.1) compare PPD to two powerful, frequently used systems for fine-grained parallelism: Cilk and Go. Go is moderately able to handle blocking interactive workloads, but Cilk (like the Spoonhower Work Stealing, or SWS, scheduler of Section 9.3) is not designed for these workloads and cannot maintain responsiveness. Because of this, in Section 9.5.2, we extend both Cilk and SWS to handle certain kinds of interactive workloads in a simple way, and compare these results to ours.

9.5.1 Comparison to Cilk and Go

Cilk and Go (both described in Chapter 2) are two widely used systems for parallel computing. Intel Cilk Plus, which arose out of the Cilk project, has now been incorporated into recent versions of `gcc`. For the experiments in this section, we use the version of the Cilk Plus runtime distributed with `gcc` 6.4.0. Go is a standalone language with parallel features. The experiments in this section use Go 1.6.2.

For the comparison to Cilk and Go, we used the Fibonacci-terminal benchmark of Section 9.3, which is reasonably representative of the other benchmarks and is a good test of each runtime's scheduler (as opposed to the languages' allocation strategies, libraries, or support for efficient data structures). In these experiments, the driver program simulated 50 terminal interactions per second. Table 9.5 shows computation and response times for PPD, Cilk and Go. Because Cilk does not allow lightweight threads to block without blocking the entire worker thread, we could not run the Cilk code effectively on one processor. Response times for Go are not shown for more than eight processors because the run times became too short for our driver program to operate.

Attempting to use Cilk on an interactive workload appears to have defeated its normally very efficient runtime scheduler. The Fibonacci computation and terminal response times were consistent across numbers of processors, indicating that Cilk was not effective at load balancing on this workload. Response times for Cilk were 2-3s, approximately three orders of magnitude

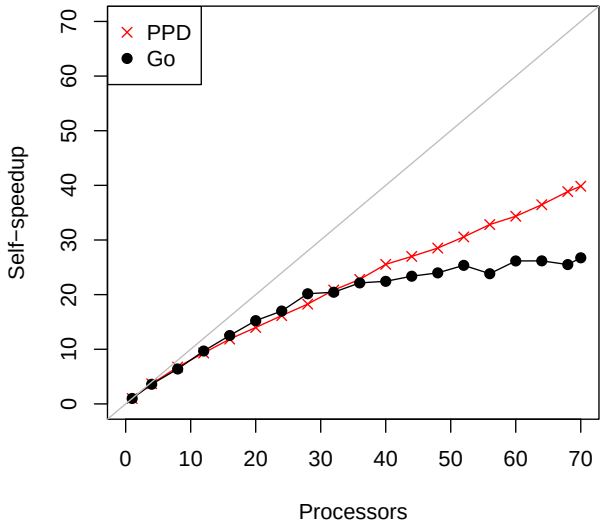


Figure 9.6: Self-speedup curves for Fibonacci-terminal on PPD and Go.

higher than PPD.

Go performs better than Cilk on this benchmark: its computation times were approximately half those of PPD across numbers of processors. Because Go's scheduler has no notion of priority, its response times were still substantially higher than PPD's (by a factor of almost 1,800 on one processor and almost 350 on eight processors). The decrease in response times on more processors is likely because, with larger numbers of processors, the interaction thread will be scheduled more quickly simply through random chance even though it is not prioritized over the computation threads.

We can compare the scalability of PPD and Go using the metric of *self-speedup*. Self-speedup is the computation time on P processors divided by the computation time of *the same runtime system* on 1 processor (as opposed to the "serial" speedup curves we have been showing thus far, in which the comparison is against a purely serial version of the code on a sequential runtime). The self-speedups of the Fibonacci-terminal benchmark on PPD and Go are shown in Figure 9.6. Go initially scales better than PPD, but its scalability begins to drop off after approximately 30 processors, at which point PPD overtakes it in self-speedup.

9.5.2 Interaction with Dedicated Threads

Cilk and SWS, two systems we have explored in this thesis, are quite effective at load-balancing fine-grained parallel computations, but do not have any facilities for allowing threads to perform interaction. In particular, if a lightweight thread performs a blocking operation such as terminal input, the entire worker thread is blocked. The scheduling algorithms used by these systems are not able to temporarily suspend a lightweight thread and resume it later.

One simple way to extend both systems to allow for interaction is to dedicate one or more processors (worker threads) to handle high-priority interactive threads. This approach fails to scale in a number of ways. First, as the number of priorities increases, dedicating even one

processor to each priority becomes unwieldy and even impossible if the machine has a small number of processors. Second, this does not allow resources to be assigned dynamically: if there is little work at high priorities, the dedicated processors will sit idle most of the time, and if there is a sudden burst of interaction, the small number of dedicated processors might not be able to keep up. The Fibonacci server example of the previous section is one program which this approach would not support: much of the CPU time in this benchmark is devoted to computation at several different priorities. If we devote, for example, 1/4 of the processors to small Fibonacci computations and then the user requests only large Fibonacci computations, these processors are wasted.

Despite these shortcomings, the above "dedicated processor" approach might work for benchmarks such as the two-priority, winner-take-all benchmarks of Section 9.3.3. In these experiments, there is one priority for computation and one priority for interaction, and interaction is infrequent enough that one processor should suffice.

In this section, we compare PPD to two other systems: a variant of the Spoonhower Work Stealing (SWS) scheduler designed to dedicate one processor to the interactive component, and a variant of the Cilk benchmarks in which interaction is handled by a separate, dedicated system thread.

First, we use a "dedicated processor" variant of the Spoonhower scheduler. In this variant, each processor maintains separate work stealing deques for low- and high-priority work. All processors but the last one (the one with the highest processor ID) work from their low-priority deque and steal work from other processors' low-priority deques (including that of the last processor). Processors place high-priority work in their own high-priority deque but will not work on it. The last (highest-ID) processor works on tasks in its high-priority deque and, when idle, steals work from other processors' high-priority deques.

Table 9.6 shows the response time and 8- and 70-processor speedup for both PPD and the dedicated processor variant of SWS ("Ded."). As in the two-priority experiments above, the response time of PPD is quite consistent across benchmarks, but the response time of the dedicated processor scheduler is highly dependent on the benchmark. The speedup of the dedicated processor scheduler on P processors is, predictably, quite similar to that of SWS on $P - 1$ processors, because one processor is always dedicated to interaction. When 70 processors are used, this effect is negligible because the marginal benefit of using a 70^{th} processor for computation is quite small. Thus, the dedicated processor strategy, like SWS, outperforms PPD on 70 processors. On only eight processors, however, PPD matches or out-performs the dedicated processor scheduler.

The second comparison is with a version of the Fibonacci-terminal benchmark written in Cilk, in which the terminal interaction is handled by a separate pthread spawned by the C code. This pthread is not under the control of the Cilk scheduler, but rather is handled directly by the operating system. Table 9.7 shows the computation and response times for the Fibonacci-terminal benchmark at 50 interactions per second, on both Cilk Plus (using a separate pthread for interaction) and the dedicated processor variant of SWS. As before, we omit response times for very short runs. The dedicated processor SWS scheduler is unable to run on one processor, because this would leave no processors available for computation. The Cilk benchmark with a dedicated interaction thread can run on one processor because the thread is scheduled by the operating system, which may interleave it with the Cilk worker thread.

With interaction factored into a separate thread, Cilk is able to demonstrate good speedups

Int.	Comp.	Resp. time (ms) Ded.	PPD	8 proc Spd. Ded.	PPD	Rat	70 proc Spd. Ded.	PPD	Rat.
Term.	Fib.	0.1	2.5	3.9×	4.9×	0.79	28.1×	28.9×	0.97
	UTS	0.1	2.0	3.7×	4.2×	0.89	24.8×	18.1×	1.37
	DMM	21.6	2.2	2.5×	3.6×	0.69	12.8×	18.4×	0.69
	BFS	10.1	1.9	4.1×	4.4×	0.93	33.6×	16.4×	2.05
	Sort	158.9	1.9	7.1×	8.1×	0.88	59.3×	45.0×	1.32
Net.	Fib.	0.1	2.3	4.7×	4.9×	0.97	33.0×	28.5×	1.16
	UTS	0.1	2.1	4.1×	4.1×	0.99	28.5×	16.3×	1.74
	DMM	6.8	2.3	2.7×	3.5×	0.76	13.1×	17.7×	0.74
	BFS	1.2	2.0	4.2×	4.4×	0.96	34.2×	14.6×	2.34
	Sort	15.5	1.9	7.5×	8.1×	0.92	60.7×	42.9×	1.41

Table 9.6: Fibonacci-terminal with PPD and a dedicated processor.

P		PPD	Cilk+pthread	(rel. to PPD)	Ded.	(rel. to PPD)
1	Comp. time	14.4 s	9.4 s	0.7	—	—
	Resp. time	2.6 ms	2569.2 ms	986.4	—	—
8	Comp. time	2.2 s	1.5 s	0.7	2.5 s	1.2
	Resp. time	2.2 ms	513.0 ms	237.5	13.7 ms	6.3
32	Comp. time	0.7 s	0.5 s	0.6	0.7 s	1.0
70	Comp. time	0.4 s	0.3 s	0.6	0.4 s	0.8

Table 9.7: Fibonacci-terminal with PPD and two "dedicated-worker" approaches.

on the Fibonacci computation: the computation time decreases with the number of processors to a similar extent as PPD. Still, response times are considerably slower on Cilk, even though a separate thread is dedicated for interaction. This somewhat counterintuitive result could be due to any number of vagaries having to do with the operating system's scheduling policy and shows the importance of having all of the program's threads under the control of the language's runtime system.

Chapter 10

Conclusion

[Exit, pursued by a bear]

The Winter's Tale (III.3.64)

In this thesis, we have made a contribution toward unifying the models of cooperative and competitive threading by extending cooperative language and cost models with the ability to manipulate and reason about threads that perform interaction or otherwise have responsiveness requirements. In particular, the concrete contributions of this thesis have been:

- A DAG-based cost model for reasoning about the structure of parallel programs with latency-incurring operations and priorities, together with appropriate scheduling principles and formal bounds on throughput and responsiveness of such programs (Chapter 3).
- The PriML language for responsive parallelism, with a type system for preventing priority inversions (Chapter 4).
- A formalization of the PriML type system in a core calculus λ^4 (Chapter 4).
- A cost semantics for reasoning about the throughput and responsiveness of PriML programs, which is validated against an appropriate transition system (Chapter 5).
- A scheduling algorithm which handles threads of varying priorities (Chapter 6).
- An implementation of the proposed methods as an extension to Spoonhower's Parallel ML (Chapter 7). The implementation includes a front end that compiles from PriML to Parallel ML, a threading library that implements prioritized futures, and a runtime scheduler for prioritized threads.
- A set of parallel interactive benchmarks which demonstrate the expressiveness and flexibility of our programming model and exercise a responsive parallel scheduler in a variety of ways (Chapters 8 and 9).
- An evaluation of the proposed methods and their implementation from the points of view of both expressiveness and performance (Chapter 9).

We believe that these contributions have sufficiently demonstrated the thesis statement of Section 1. Still, there is much work to be done, both directly toward the contributions of this thesis, and toward the more general goal of combining cooperative and competitive threading. In the rest of this chapter, we first give some immediate directions for future research leading out of

this thesis, and then conclude with a discussion of where this work can lead more broadly.

10.1 Future Directions

First-class priorities. Priorities are not first-class citizens of PriML, though they are in our threading library (Section 7.2). Priority polymorphism suffices in many situations where first-class priorities would be useful, but the restriction on priorities is still an inconvenience at times. For example, the Fibonacci server of Section 9.4 includes a multi-way conditional to spawn a thread at the appropriate priority:

```
if n < 15 then (* small number *)
  cmd[fg] {td <- spawn[smallfib] {do ([smallfib]fib n)};
    do (loop (td::threads))}
else if n < 25 then (* medium number *)
  cmd[fg] {td <- spawn[medfib] {do ([medfib]fib n)};
    do (loop (td::threads))}
else (* large number *)
  cmd[fg] {td <- spawn[largefib] {do ([largefib]fib n)};
    do (loop (td::threads))}
```

This code is manageable for three priorities, but would become unwieldy for many more priorities. It would be more convenient to be able to have a single spawn statement and programmatically produce the appropriate priority.

First-class priorities would be even more convenient with the ability to check the priority of threads and the ordering of priorities at runtime. For example:

```
if p < Thread.priorityOf t then
  cmd[p] { x <- sync t; ret (SOME x) }
else
  cmd[p] { poll t }
```

Still, we have not encountered an example in our evaluation which is intractable without first-class priorities.

For inspiration in this area, we could look to place-based languages and type systems (see Section 2.3.3), which often allow first-class places.

If priorities can be passed around as first class objects, it's also natural to ask whether they can be generated at runtime (currently, all priorities are declared statically). This seems possible, though the type system would need a static approximation of what priorities exist at what points in the program as well as their ordering relations. Because such an approximation would necessarily be conservative, it is not inherently clear how much additional expressiveness dynamically-generated priorities would allow while still enabling static prevention of priority inversions.

Dynamic priorities and fairness. A related feature would be the ability to dynamically adjust the priority of threads, and/or the fairness criterion. Dynamic priorities would be useful, for example, to increase the priority of a computation if it has been running for a long time and

its results are needed. A similar effect could be achieved by adjusting the fairness criterion at runtime to shift the processor resources given to different computations as the requirements of a program change.

Dynamic priorities would pose substantial challenges to reasoning about priority inversions and cost, and would require quite a bit of thought. Dynamic fairness criteria appear more straightforward; indeed, our threading library in principle already allows the fairness criterion to be modified at runtime, though we do not use this ability and haven't tested it. A somewhat larger challenge would be extending the cost bounds to allow for a fairness criterion that changes over time.

Channels, other synchronization. Although the threading model of PriML is quite expressive and general, another possible direction for future work would be to extend the set of synchronization and communication primitives. For example, we could add support for CML-style communication channels. Because communication on channels is synchronous, we would need to extend the type system to ensure that a high-priority thread never blocks waiting for a low-priority thread to send or receive on a channel. One way to achieve this would be to annotate every channel with a priority ρ and have the type system enforce that only threads of priority greater than ρ may send on the channel and only threads of priority less than ρ may receive on it.

Analysis of work stealing algorithm Possibly the most direct avenue for future work would be to analyze the work stealing algorithm of Chapter 6 and hopefully prove that it meets the bounds given for fairly prompt schedules by Theorem 2. Analyses of algorithms such as this one require an enormous amount of technical detail, but we suspect it would be possible to leverage existing proof techniques (e.g., [2]). We describe in Section 6.4 how the design of the algorithm was guided by intuitions about fairly prompt scheduling, and why we believe that the algorithm presented should produce a good approximation of a fairly prompt schedule.

Development of benchmark suite One of the contributions of this thesis is a set of responsive parallel programs on which we have evaluated our implementation. In developing this set of benchmarks, especially the set of orthogonal benchmarks (Section 9.3), we attempted to provide good coverage of the design space of both parallel programs and interaction. Still, our job was made more difficult by the fact that (to the best of our knowledge) no definitive suite of parallel interactive benchmarks currently exists. It would be worthwhile to extend the set of benchmarks in this thesis into a larger, standalone benchmark suite that could be used to evaluate and compare future approaches to responsive parallelism, in the same way that, for example, the Problem Based Benchmark Suite [109] exists for parallel algorithms.

10.2 Concluding Remarks

As consumer electronic devices continue to scale by increasing the number, rather than the speed, of processors, it will become increasingly important for software to make meaningful use of parallelism. The elegance and programmability of cooperative threading models make them an

attractive choice for using parallelism to improve the throughput of computation, but consumer software has needs beyond simply throughput. For this reason, we seek a unified parallelism model that combines the throughput guarantees of cooperative threading and the responsiveness of competitive threading. In this thesis, we have taken a number of steps from the standpoint of cooperative threading toward including ideas of preemption, interaction and responsiveness while maintaining the high levels of abstraction and provable cost bounds that are the hallmarks of cooperative models. In the future, we will hopefully see research at other points along the design space that spans cooperative and competitive threading, and even a full unification of the two models.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-185-2. 2.1.1

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP '13*, 2013. 2.1.3, 6.1, 6.2, 6.4, 7.1, 7.1, 10.1

[3] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 84–95, 2014. 2.2

[4] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallelizable jobs online to minimize the maximum flow time. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 195–205, New York, NY, USA, 2016. ACM. 2.3.2

[5] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallel DAG jobs online to minimize average flow time. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 176–189, Philadelphia, PA, USA, 2016. Society for Industrial and Applied Mathematics. ISBN 978-1-611974-33-1. 2.3.2

[6] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001. 2.1.3, 3.3, 3.4.1, 6, 6.1, 6.4

[7] Arvind, Kim P. Gostelow, and Wil Plouffe. An asychronous language and computing machine. Technical Report TR-114A, Department of Information and Computer Science, University of California, Irvine, December 1978. 2.1.1

[8] Guillaume M. J b. Chaslot, Mark H. M. Win, and H. Jaap Van Den Herik. Parallel monte-carlo tree search, 2008. 8.2

[9] Özalp Babaoğlu, Keith Marzullo, and Fred B. Schneider. A formalization of priority inversion. *Real-Time Systems*, 5(4):285–303, 1993. 1.1, 2.3.1, 2.3.1, 9.4, 9.4

[10] Benjamin Berg, Jan Pieter Dorsman, and Mor Harchol-Balter. Towards optimality in parallel job scheduling. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, New York, NY, USA, 2018. ACM.

2.3.2

[11] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 81–92, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. 2.1.1

[12] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. 2.3.1

[13] Guy Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990. 2.1.1

[14] Guy E. Blelloch. A nested data-parallel language. Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1995. 1, 2.1.1

[15] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Functional Programming Languages and Computer Architecture*, pages 226–237, 1995. 1

[16] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 213–225, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7. 1, 2.1.2, 5

[17] Guy E. Blelloch and Margaret Reid-Miller. Pipelining with futures. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 249–259, New York, NY, USA, 1997. ACM. ISBN 0-89791-890-8. 2.1.1

[18] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994. 2.1.1

[19] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46:281–321, March 1999. 2.1.3

[20] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 189–199, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0079-7. 9.3

[21] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998. 2.1.2

[22] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999. 2.1.2, 2.1.3, 6, 6.1

[23] 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, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. 1, 2.1.1

[24] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109 – 130, 2002. ISSN 0304-3975. Selected Papers in honour of Maurice Nivat. 2.3.4

[25] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21 (2):201–206, 1974. 2.1.2, 6

[26] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM. ISBN 0-89791-060-5. 1, 2.1.1, 2.1.3

[27] Colin Campbell, Ralph Johnson, Ade Miller, and Stephen Toub. *Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 2010. 2.1.1

[28] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. 2.1.1

[29] Tristan Cazenave and Nicolas Jou. On the parallelization of uct. In *Proceedings of the Computer Games Workshop 2007*, CGW 2007, pages 93–101. 8.2

[30] Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 94–105, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. 2.1.1

[31] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal — nested data parallelism in Haskell. In Rizos Sakellariou, John Gurd, Len Freeman, and John Keane, editors, *Euro-Par 2001 Parallel Processing*, pages 524–534, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44681-1. 2.1.1

[32] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. doi: 10.1145/1248648.1248652. 1, 2.1.1

[33] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, 21 (3):291–312, 2007. 2.2

[34] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 11–22, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. 2.3.3

[35] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kiel-

stra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005. 1, 2.1.1, 2.3.3

[36] Guillaume Chaslot, Jahn takeshi Saito, Jos W. H. M. Uiterwijk, Bruno Bouzy, and H. Jaap Herik. Monte-carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 83–90, 2006. 8.2

[37] John S. Danaher, I.-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147 – 171, 2006. ISSN 0167-6423. Special issue on synchronization and concurrency in object-oriented languages. 7.1

[38] Rowan Davies. A temporal-logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996. 2.3.3

[39] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '89, pages 1–12, New York, NY, USA, 1989. ACM. ISBN 0-89791-332-9. 2.3.2

[40] Laxman Dhulipala and Sam Westrick. PWSA*: Parallel work-stealing A*. Unpublished manuscript, 2018. 8.1

[41] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 261–276, New York, NY, USA, 1999. ACM. ISBN 1-58113-140-2. 2.3.2

[42] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing*, 38(3):408–423, 1989. 2.1.2, 3.3, 3.3

[43] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 685–692, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics. 2.3.2

[44] Markus Enzenberger and Martin Müller. A lock-free multithreaded monte-carlo tree search algorithm. In *Proceedings of the 12th International Conference on Advances in Computer Games*, ACG'09, pages 14–20, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12992-7, 978-3-642-12992-6. 8.2

[45] Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. Automatically splitting a two-stage lambda calculus. In *Proceedings of the 25 European Symposium on Programming, ESOP*, pages 255–281, 2016. 2.3.3

[46] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349 – 366, 1990. ISSN 0743-7315. Data-flow Processing. 1, 1, 2.1.1

[47] C. J. Fidge. A formal definition of priority in CSP. *ACM Trans. Program. Lang. Syst.*, 15 (4):681–705, September 1993. ISSN 0164-0925. 1.1, 2.3.1

[48] Kristián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism and interactive performance of desktop applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 129–138, New York, NY, USA, 2000. ACM. ISBN 1-58113-317-0. 2.3.1

[49] Matthew Fluet, Nic Ford, Mike Rainey, John H. Reppy, Adam Shaw, and Yingqi Xiao. Status report: the Manticore project. pages 15–24, 2007. 2.1.1, 2.2, 4.1

[50] Matthew Fluet, Mike Rainey, John H. Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Workshop on Declarative Aspects of Multicore Programming*, DAMP 2007, pages 37–44, 01 2007. 1, 2.1.1, 2.2

[51] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 241–252, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. 1, 2.2, 7.1

[52] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011. 2.2

[53] Nissim. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer US, New York, NY, 1986. ISBN 9781461248866. 3.4

[54] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. 2.1.1, 7.2

[55] Hans Fugal. Futures for C++11 at Facebook. `https://code.facebook.com/posts/1661982097368498/futures-for-c-11-at-facebook/`. Retrieved May 2018. 2.1.1

[56] The Go Authors. The Go programming language specification. February 2018. URL `https://golang.org/ref/spec#Go_statements`. 1, 2.2

[57] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchial CPU scheduler for multimedia operating systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 107–121, New York, NY, USA, 1996. ACM. ISBN 1-880446-82-0. 2.3.2

[58] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969. 2.1.2

[59] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 11–20, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0079-7. 2.3.2

[60] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985. ISSN 0164-0925. 1, 2.1.1,

2.1.3

[61] Riyaz Haque and Jens Palsberg. Type inference for place-oblivious objects. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 371–395, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-86-6. 2.3.3

[62] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, Cambridge, UK, 2013. 2.3.2

[63] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. 4.2.2

[64] Robert Harper and Chris Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU-CS-96-136R, School of Computer Science, Carnegie Mellon University, September 1996. 4.3

[65] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 94–105, New York, NY, USA, 1993. ACM. 1.1, 2.3.1

[66] Y. He, W. J. Hsu, and C. E. Leiserson. Provably efficient online non-clairvoyant adaptive scheduling. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007. 2.3.2

[67] Sungjin Im, Benjamin Moseley, Kirk Pruhs, and Eric Torng. Competitively scheduling tasks with intermediate parallelizability. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 22–29, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2821-0. 2.3.2

[68] Shams Imam and Vivek Sarkar. Habanero-Java library: A Java 8 framework for multicore programming. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 75–86, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2926-2. 1, 2.1.1

[69] Shams Imam and Vivek Sarkar. Load balancing prioritized tasks via work-stealing. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing*, pages 222–234, 2015. 1, 2.2

[70] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5. 2.1.1

[71] Alan Jeffrey. A distributed object calculus. In *Proceedings of the Foundations of Object Oriented Languages*. ACM Press, 2000. 2.3.3

[72] Limin Jia and David Walker. Modal proofs as distributed programs. In David Schmidt, editor, *13th European Symposium on Programming, ESOP 2004*, pages 219–233, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 2.3.3

[73] R. A. Knepper, S. S. Srinivasa, and M. T. Mason. Hierarchical planning architec-

tures for mobile manipulation tasks in indoor environments. In *2010 IEEE International Conference on Robotics and Automation*, pages 1985–1990, May 2010. doi: 10.1109/ROBOT.2010.5509669. 1, 8.1, 8.1

[74] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-45375-X, 978-3-540-45375-8. 8.2

[75] David A. Kranz, Robert H. Halstead, and Eric Mohr. Mul-t: A high-performance parallel lisp. In Takayasu Ito and Robert H. Halstead, editors, *Parallel Lisp: Languages and Systems*, pages 306–311, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. ISBN 978-3-540-47143-1. 2.1.3

[76] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–117, 1980. 2.3.1

[77] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report TR 98-11, Computer Science Dept., Iowa State University, October 1998. 8.1

[78] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000. 1, 2.1.1

[79] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, volume 44, pages 227–242, 2009. 2.1.1

[80] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, February 2012. 9.3

[81] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.06*. INRIA, 2017. URL `http://caml.inria.fr/pub/docs/manual-ocaml/libthreads.html`. 2.2

[82] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 14:1–14:13, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4092-2. 2.3.2

[83] Jonathan Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, School of Computer Science, Carnegie Mellon University, October 2003. 2.3.3

[84] Daniel Morsing. The Go scheduler. `https://morsmachine.dk/go-scheduler`. Retrieved May 2018. 2.2

[85] Stefan Muller and Stephen Chong. Towards a practical secure concurrent language. In *Proceedings of the ACM International Conference on Object Oriented Programming Sys-*

tems Languages and Applications, OOPSLA '12, pages 57–74, New York, NY, USA, 2012. ACM. 2.3.4

[86] Stefan K. Muller and Umut A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 71–82, 2016. 1.2, 2.2, 6.3, 6.4, 7.1

[87] Stefan K. Muller, Umut A. Acar, and Robert Harper. Responsive parallel computation: Bridging competitive and cooperative threading. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017. 1.2, 2.2, 2.3.3, 3.3, 4.2.2, 9.3.3, 9.4

[88] Stefan K. Muller, Umut A. Acar, and Robert Harper. Competitive parallelism: Getting your priorities right. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming*, ICFP 2018, New York, NY, USA, 2018. ACM. To Appear. 1.2

[89] Tom Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. Available as technical report CMU-CS-08-126. 7.3

[90] Tom Murphy, VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007*, November 2007. 2.3.3

[91] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4UNIX scheduler unacceptable for multimedia applications. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV '93, pages 41–53, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-58404-8. 2.3.2

[92] Rishiyur S. Nikhil. Id language reference manual, 1991. 1, 2.1.1

[93] Rishiyur S. Nikhil and Arvind. *Implicit Parallel Programming In pH*. Morgan Kaufmann, San Francisco, 2001. 1

[94] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: an unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing, 19th International Workshop, LCPC 2006, New Orleans, LA, USA, November 2-4, 2006. Revised Papers*, pages 235–250, 2006. 9.3

[95] Bouzy Universit Paris, B. Bouzy, and B. Helmstetter. Monte-carlo go developments. In *ACG. Volume 263 of IFIP., Kluwer (2003) 159174 5 Typically the*, pages 159–174. Kluwer Academic, 2003. 8.2

[96] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. 2.3.3

[97] Michael J. Pilling. Dangers of priority as a structuring principle for real-time languages. *Australian Computer Science Communications*, 13(1):18.1–18.10, 1991. 1.1, 2.3.1

[98] Python Software Foundation. *The Python standard library*. 2016. URL `https://docs.python.org/3/library/concurrent.futures.html`. 2.2

[99] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical

memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 392–406, New York, NY, USA, 2016. ACM. 1, 7

[100] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, UK, 1999. 1, 2.2, 2.2

[101] John Reppy, Claudio V. Russo, and Yingqi Xiao. Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 257–268, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. 2.2

[102] John H. Reppy. CML: a higher concurrent language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 293–305, 1991. 2.2

[103] Mads Rosendahl. Automatic complexity analysis. In *Functional Programming Languages and Computer Architecture*, FPCA '89, pages 144–156. ACM Press, 1989. 2.1.2

[104] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, Dec 2014. 2.3.2

[105] David Sands. Complexity analysis for a lazy higher-order language. In *In Proceedings of the 3rd European Symposium on Programming*, pages 361–376. Springer-Verlag, 1990. 2.1.2

[106] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. 2.1.3

[107] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 695–706, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. 2.3.3

[108] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990. 2.3.1

[109] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. 10.1

[110] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating Systems Concepts*. John Wiley & Sons, New York, 6 edition, 2002. 2.3.2

[111] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 355–364, 1998. 2.3.4

[112] Daniel Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009. 2.1.1, 2.1.2, 7, 7.2

[113] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008. 1, 2.1.2

[114] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 91–100, New York, NY, USA, 2009. ACM. 2.1.2

[115] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012. URL `http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf`. 8.1

[116] Twitter, Inc. Twitter futures. `https://twitter.github.io/finagle/guide/developers/Futures.html`. Retrieved May 2018. 2.1.1

[117] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975. 2.1.2

[118] Mark T. Vandevoorde and Eric S. Roberts. Workcrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, Aug 1988. 2.1.3

[119] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association. 2.3.2

[120] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Work-stealing with configurable scheduling strategies. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 315–316, 2013. 1, 2.2

[121] Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. Data structures for task-based priority scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 379–380, 2014. 1, 2.2

[122] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *In ACM 1998 Workshop on Java for High-Performance Network Computing (New*. ACM, Ed., ACM Press, 1998. 2.2, 2.3.3

[123] Christopher S Zakian, Timothy AK Zakian, Abhishek Kulkarni, Buddhika Chamith, and Ryan R Newton. Concurrent Cilk: Lazy promotion from tasks to threads in C/C++. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 73–90. Springer International Publishing, 2015. 1, 2.2

[124] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay

Saraswat, and Mikio Takeuchi. GLB: Lifeline-based global load balancing library in X10. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 31–40, New York, NY, USA, 2014. ACM. 2.1.3