Competitive Parallelism: Getting Your Priorities Right

STEFAN K. MULLER, Carnegie Mellon University
UMUT A. ACAR, Carnegie Mellon University and Inria
ROBERT HARPER, Carnegie Mellon University

Multi-threaded programs have traditionally fallen into one of two domains: *cooperative* and *competitive*. These two domains have traditionally remained mostly disjoint, with cooperative threading used for increasing *throughput* in compute-intensive applications such as scientific workloads and cooperative threading used for increasing *responsiveness* in interactive applications such as GUIs and games. As multicore hardware becomes increasingly mainstream, there is a need for bridging these two disjoint worlds, because many applications mix interaction and computation and would benefit from both cooperative and competitive threading.

In this paper, we present techniques for programming and reasoning about *parallel interactive* applications that can use both cooperative and competitive threading. Our techniques enable the programmer to write rich parallel interactive programs by creating and synchronizing with threads as needed, and by assigning threads user-defined and partially ordered priorities. To ensure important responsiveness properties, we present a modal type system analogous to S4 modal logic that precludes low-priority threads from delaying high-priority threads, thereby statically preventing a crucial set of *priority-inversion* bugs. We then present a cost model that allows reasoning about responsiveness and completion time of well-typed programs. The cost model extends the traditional work-span model for cooperative threading to account for competitive scheduling decisions needed to ensure responsiveness. Finally, we show that our proposed techniques are realistic by implementing them as an extension to the Standard ML language.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; *Concurrent programming languages*; *Concurrent programming structures*; • **Theory of computation** → Interactive computation;

ACM Reference format:
Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Competitive Parallelism: Getting Your Priorities Right. *Proc. ACM Program. Lang.* 2, ICFP, Article 95 (September 2018), 30 pages.
https://doi.org/10.1145/3236790

1 INTRODUCTION

The increasing proliferation of multicore hardware has sparked a renewed interest in programming-language support for *cooperative threading*. In cooperative threading, threads correspond to pieces of a job and are scheduled with the goal of completing the job as quickly as possible—or to maximize *throughput*. Cooperative thread scheduling algorithms are therefore usually non-preemptive: once a thread starts executing, it is allowed to continue executing until it completes.

Cooperatively threaded languages such as NESL [Blelloch et al. 1994], Cilk [Frigo et al. 1998], parallel Haskell [Chakravarty et al. 2007; Keller et al. 2010] and parallel ML [Fluet et al. 2011; Jagannathan et al. 2010; Raghunathan et al. 2016], have at least two important features:

2475-1421/2018/9-ART95
https://doi.org/10.1145/3236790

- The programmers can express opportunities for parallelism at a high level with relatively simple programming abstractions such as *fork/join* and *async/finish*. The run-time system of the language then handles the creation and scheduling of the threads.
- The efficiency and performance of parallel programs written at this high level can be analyzed by using cost models based on *work* and *span* (e.g. [Blelloch and Greiner 1995, 1996; Eager et al. 1989; Spoonhower et al. 2008]), which can guide efficient implementations.

Cooperative threading is elegant and expressive but it mostly excludes the important class of *interactive applications*, which require communication with the external world, including users and other programs. Such interactive applications typically require *responsiveness*, such as the requirement to process user input as soon as possible. Ensuring responsiveness usually requires *competitive threading*, where threads are scheduled pre-emptively, usually based on priorities. To guarantee responsiveness, most competitive threading libraries in use today expose a fixed range of numerical priorities which may be assigned to threads. Regardless of the threading primitives used, this greatly complicates the task of writing programs:

- Writing effective competitively threaded programs requires assigning priorities to threads. While this can be simple for simple programs, using priorities at scale is a big challenge because most current approaches to priorities are inherently anti-modular. Because priorities are totally ordered, writing responsive programs might require reasoning about whether a thread should be given a higher or lower priority than a thread introduced in another part of the program, or possibly even in a library function.
- To compensate for this lack of modularity, many systems expose large numbers of priorities: the POSIX threads (pthreads) API exposes scheduling policies with as many as 100 levels. Without clean guidelines governing their use, however, programmers must still reason globally about how to assign these numbers to threads. Studies have shown that programmers struggle to use systems with even 7 priorities [Hauser et al. 1993].
- Reasoning about performance is much more difficult: the clean work-span model of cooperative threading does not apply to competitive threading, because of the impact of priorities on run-time. Furthermore, in competitive threading, *priority inversions*, where a low-priority thread delays a high-priority one, can have harmful and even disastrous consequences. For example, "Mars Pathfinder", which landed on Mars on 4 July 1997, suffered from a software bug, traced to a priority inversion, that caused the craft to reset itself periodically. The bug had to be patched remotely so the mission could continue.

In this paper, we develop language techniques and a cost model for writing parallel interactive programs that use a rich set of cooperative and competitive threading primitives. This problem is motivated by the fact that as shared-memory hardware becomes widely used, competitively threaded, interactive applications will need to take advantage of the benefits of this parallel hardware, and not just cooperatively threaded, compute-intensive applications.

We present a programming language with features for *spawning* and *syncing* with asynchronous threads, which may be assigned priorities by the programmer. Aside from priorities, these threads are equivalent to futures, a powerful general-purpose cooperative threading mechanism. Like futures, threads are first-class values in the language. To enable modular programming with priorities, we allow the programmer to declare any number of priorities and define a partial order between them. The resulting language is sufficiently powerful to enable both cooperative and competitive threading. For example, the programmer can write a purely compute intensive program (e.g., parallel Quicksort), a purely interactive program (e.g. a simple graphical user interface), and anything that combines the two (e.g. an email client that sorts tens of thousands of emails in parallel in the background while remaining responsive to user interaction events).

To reason about the efficiency and responsiveness of the programs written in this language, we present a cost model that bounds both the total computation time of the program and the response time of individual threads. Our cost semantics extends prior cost models of cooperative parallel programs to enable reasoning about the response time of threads with partially-ordered priorities. The main theoretical result of the paper shows that the response time of a thread does not depend on the amount of computation performed at lower priorities for any program in which threads do not sync on threads of lower priority. Such a sync clearly allows the response time of a high-priority thread to depend on low-priority work and is an example of the classic problem of priority inversions described above.

Our prior work on extending cooperative threading with priorities [Muller et al. 2017] also observed that priority inversions prevent responsiveness guarantees and presented static mechanisms for avoiding them. That work, however, considers only two priorities (high and low). Research in languages such as Ada [Cornhill and Sha 1987; Levine 1988] also discusses the importance of preventing priority inversion in a general setting with rich priorities, but we are aware of no prior static language mechanisms for doing so.

To guarantee appropriate bounds on responsiveness, we specify a type system that statically identifies and prevents priority inversions that would render such an analysis impossible. The type system enforces a monadic separation between *commands*, which are restricted to run at a certain priority, and *expressions*, which are priority-invariant. The type system then tracks the priorities of threads and rejects programs in which a high-priority thread may synchronize with a lower-priority one. In developing this system, we draw inspiration from modal logics, where the "possible worlds" of the modal logic correspond to priorities in our programs. More specifically, our type system is analogous to S4 modal logic, where the accessibility relation between worlds is assumed to be reflexive and transitive. This accessibility relation reflects the fact that the ways in which priorities are intended to interact is inherently asymmetric. Modal logic has proved to be effective in many problems of computer science. For example, Murphy et al. [2004], and Jia and Walker [2004] use the modal logic S5, where the accessibility relation between worlds is assumed to be symmetric (as well as reflexive and transitive), to model distributed computing.

The dynamic semantics of our language is a transition system that simulates, at an abstract level, the execution of a program on a parallel machine. We show that, for well-typed programs, our cost model accurately predicts the response time of threads in such an execution. Finally, we show that the proposed techniques can be incorporated into a practical language by implementing a compiler which typechecks prioritized programs and compiles them to a parallel version of Standard ML. We also provide a runtime system which schedules threads according to their priorities.

The specific contributions of this paper include the following.

- An extension of the Parallel ML language, called PriML, with language constructs for user-defined, partially ordered priorities.
- A core calculus λ^4 that captures the essential ideas of PriML and a type system that guarantees inversion-free use of threads.
- A cost semantics for λ^4 which can be used to make predictions about both overall computation time and responsiveness, and a proof that these predictions are accurately reflected by the dynamic semantics.
- An implementation of the compiler and the runtime system for PriML as an extension of the Parallel MLton compiler.
- Example benchmarks written in our implementation that give preliminary qualitative evidence for the practicality of the proposed techniques.

2 OVERVIEW

We present an overview of our approach to multithreaded programming with priorities by using a language called PriML that extends Standard ML with facilities for prioritized multithreaded programming. As a running example, we consider an email client which interacts with a user while performing other necessary tasks in the background. The purpose of this section is to highlight the main ideas. The presentation is therefore high-level and sometimes informal. The rest of the paper formalizes these ideas (Section 3), expands on them to place performance bounds on PriML programs (Section 4) and describes how they may be realized in practice (Section 6).

Priorities. PriML enables the programmer to define priorities as needed and specify the relationships between them. For example, in our mail client, we sometimes 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`. 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`.

Modal type system. To ensure responsive use of priorities, PriML provides a modal type system that tracks priorities. The types of PriML include the standard types of functional programming languages as well as a type of thread handles, by which computations can refer to, and synchronize with, running threads.

To support computations that can operate at multiple priorities, the type system supports *priority polymorphism* through a polymorphic type of the form $\forall \pi : C.\tau$, where π is a newly bound priority variable, and C is a set of constraints of the form $\rho_1 \preceq \rho_2$ (where ρ_1 and ρ_2 are priority constants or variables, one of which will in general be π), which bounds the allowable instantiations of π.

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 with threads. *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 (which have a distinguished type) and abstract over priorities to introduce priority-polymorphic expressions.

Threads. Once declared, priorities can be used to specify the priority of threads. 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. Spawned threads are annotated with a priority and run asynchronously with the rest of the program.

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

```
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
```

Fig. 1. Code for multithreaded quicksort, which is priority polymorphic.

The spawn command takes a command to run in the new thread and returns a handle to the spawned thread. In the above code, this handle is ignored, but it can also be bound to a variable using the notation x <- m; 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
```

Example: priority-polymorphic multithreaded quicksort. Priority polymorphism allows prioritized code to be compositional. For example, several parts of our email client might wish to use a library function qsort 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.) 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 1 illustrates the code for a multithreaded implementation of Quicksort in PriML. The code 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, the function takes as an argument a priority p, to which the body of the function may refer (e.g. to spawn threads at that priority)[1]. 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 sub-sequences

[1] 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.

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

(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   case next_event () of
 7   SORT_BY_DATE =>
 8     cmd[loop_p] {
 9       spawn[sort_p] {
10         l <- do ([sort_p]qsort
11                      date emails);
12         ret (display_ordered l)
13       }
14     }
15     | ...
```

(b) Well-typed event loop code

Fig. 2. Two implementations of the event loop, one of which displays a priority inversion.

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 at priority `p`.

This code also highlights the interplay between expressions and commands in PriML. The expression `cmd[p] m` introduces an encapsulated command, and the command do e evaluates e to an encapsulated command, and then runs the command.

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. An illustration of such a situation appears in Figure 2a. 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 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 12) causes the remainder of the event loop (high-priority) to wait on the sorting thread (lower priority), this code will be correctly rejected by the type system. The programmer could instead write the code as shown in Figure 2b, 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 2a could easily be noticed by a programmer, the type system also rules out more subtle priority inversions. Consider the ill-typed code in Figure 3, 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"

```
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      l <- sync t;
10     ret (display_ordered l)
11   }
12
13 fun loop emails : unit cmd[loop_p] =
14   case next_event () of
15   SORT_BY_DATE =>
16     cmd[loop_p] {
17       t <- spawn[sort_p] { do ([sort_p]qsort date emails) };
18       spawn[display_p] { do ([sort_p]disp t) }
19     }
20     | ...
```

Fig. 3. An ill-typed attempt at chaining threads together.

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 9:

```
constraint violated at 9.10-9.15: display_p <= p_1
```

This `sync` operation is passed a thread of priority `p` (note from the function signature that the types of thread handles explicitly track their priorities), and there is no guarantee that `p` is higher-priority than `display_p` (and, in fact, the instantiation on line 18 would violate this constraint). We may correct the type error in the `disp` function by adding this 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 18 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 2b 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 a higher priority. 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.

$$\begin{array}{llll}
\textit{Types} & \tau & ::= & \texttt{unit} \mid \texttt{nat} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \mid \tau\ \texttt{thread}[\rho] \mid \tau\ \texttt{cmd}[\rho] \mid \forall \pi : C.\tau \\
\textit{Priorities} & \rho & ::= & \overline{\rho} \mid \pi \\
\textit{Constrs.} & C & ::= & \rho \preceq \rho \mid C \wedge C \\
\textit{Values} & v & ::= & x \mid \langle\rangle \mid \overline{n} \mid \lambda x.e \mid \langle v, v \rangle \mid \texttt{l} \cdot v \mid \texttt{r} \cdot v \mid \texttt{tid}[a] \mid \texttt{cmd}[\rho]\ \{m\} \mid \Lambda \pi : C.e \\
\textit{Exprs.} & e & ::= & v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{ifz}\ v\ \{e; x.e\} \mid v\ v \\
 & & & \mid (v, v) \mid \texttt{fst}\ v \mid \texttt{snd}\ v \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid \texttt{case}\ v\ \{x.e; y.e\} \\
 & & & \mid \texttt{output}\ v \mid \texttt{input} \mid v[\rho] \mid \texttt{fix}\ x{:}\tau\ \texttt{is}\ e \\
\textit{Commands} & m & ::= & x \leftarrow e; m \mid \texttt{spawn}[\rho; \tau]\ \{m\} \mid \texttt{sync}\ e \mid \texttt{ret}\ e
\end{array}$$

Fig. 4. Syntax of λ^4

3 THE λ^4 CALCULUS

In this section, we define a core calculus λ^4 which captures the key ideas of a language with an ML-style expression layer and a modal layer of prioritized asynchronous threads. Some straightforward rules and proof details which are omitted from this section for space reasons are available in the extended version submitted as supplemental material. Figure 4 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 τ thread$[\rho]$ is used for a handle to an asynchronous thread running at priority ρ and returning a value of type τ. The type τ 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 will be 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 6 how a compiler can hoist priority definitions out of the program).

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, e.g. (e_1, e_2) can be expressed using let bindings as let $x = e_1$ in let $y = e_2$ 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 cmd$[\rho]$ $\{m\}$ and priority-level abstractions $\Lambda \pi : C.e$.

Expressions include values, let binding, the if-zero conditional 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), inl v and 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 (separating the two will allow us to account for the cost of performing the allocation). Finally, expressions include the case construct case e $\{x.e_1; y.e_2\}$, output, input, priority instantiation $v[\rho]$ and fixed points.

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. Spawning a thread and synchronizing with a thread are also commands. The spawn

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

$$\texttt{natI}\ \frac{}{\Gamma \vdash^R_\Sigma \overline{n} : \texttt{nat}} \qquad \texttt{natE}\ \frac{\Gamma \vdash^R_\Sigma v : \texttt{nat} \qquad \Gamma \vdash^R_\Sigma e_1 : \tau \qquad \Gamma, x : \texttt{nat} \vdash^R_\Sigma e_2 : \tau}{\Gamma \vdash^R_\Sigma \texttt{ifz}\ v\ \{e_1; x.e_2\} : \tau}$$

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

$$\times I_1\ \frac{\Gamma \vdash^R_\Sigma v_1 : \tau_1 \qquad \Gamma \vdash^R_\Sigma v_2 : \tau_2}{\Gamma \vdash^R_\Sigma (v_1, v_2) : \tau_1 \times \tau_2} \qquad \times E_1\ \frac{\Gamma \vdash^R_\Sigma v : \tau_1 \times \tau_2}{\Gamma \vdash^R_\Sigma \texttt{fst}\ v : \tau_1} \qquad +I_1\ \frac{\Gamma \vdash^R_\Sigma v : \tau_1}{\Gamma \vdash^R_\Sigma \texttt{inl}\ v : \tau_1 + \tau_2}$$

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

$$\textsc{Output}\ \frac{\Gamma \vdash^R_\Sigma v : \texttt{nat}}{\Gamma \vdash^R_\Sigma \texttt{output}\ v : \texttt{unit}} \qquad \textsc{Input}\ \frac{}{\Gamma \vdash^R_\Sigma \texttt{input} : \texttt{nat}} \qquad \texttt{cmdI}\ \frac{\Gamma \vdash^R_\Sigma m \mathrel{\underset{\cdot}{\sim}} \tau\ @\ \rho}{\Gamma \vdash^R_\Sigma \texttt{cmd}[\rho]\ \{m\} : \tau\ \texttt{cmd}[\rho]}$$

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

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

Fig. 5. Selected expression typing rules.

command $\mathsf{spawn}[\rho; \tau]\ \{m\}$ is parametrized by both a priority ρ and the type τ of the return value of m for convenience in defining the dynamic semantics.

3.1 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 will ensure 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 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.

$$\textsc{Bind}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathsf{cmd}[\rho] \qquad \Gamma, x : \tau \vdash^R_\Sigma m \mathrel{\dot\sim} \tau' \mathbin{@} \rho}{\Gamma \vdash^R_\Sigma x \leftarrow e; m \mathrel{\dot\sim} \tau' \mathbin{@} \rho} \qquad \textsc{Spawn}\quad \frac{\Gamma \vdash^R_\Sigma m \mathrel{\dot\sim} \tau \mathbin{@} \rho'}{\Gamma \vdash^R_\Sigma \mathsf{spawn}[\rho'; \tau]\ \{m\} \mathrel{\dot\sim} \tau\ \mathsf{thread}[\rho'] \mathbin{@} \rho}$$

$$\textsc{Sync}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathsf{thread}[\rho'] \qquad \Gamma \vdash^R \rho \preceq \rho'}{\Gamma \vdash^R_\Sigma \mathsf{sync}\ e \mathrel{\dot\sim} \tau \mathbin{@} \rho} \qquad \textsc{Ret}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau}{\Gamma \vdash^R_\Sigma \mathsf{ret}\ e \mathrel{\dot\sim} \tau \mathbin{@} \rho}$$

Fig. 6. Command typing rules.

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

Fig. 7. Constraint entailment

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. Most rules for this judgment are shown in Figure 5 (omitted rules are similar to others and are available in the supplementary material). 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 $\mathtt{tid}[a]$ looks up the thread a in the signature. The rule for encapsulated commands $\mathsf{cmd}[\rho]\ \{m\}$ requires that the command m be well-typed and runnable at priority ρ, using the typing judgment for commands, which will be 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 $\Gamma \vdash^R C$ will be 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 6. 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 $\mathsf{spawn}[\rho'; \tau]\ \{m\}$ requires that m be runnable at priority ρ' and return a value of type τ. The spawn command returns a thread handle of type $\tau\ \mathtt{thread}[\rho']$, and may do so at any priority. The $\mathsf{sync}\ e$ 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. Finally, if e has type τ, then the command $\mathtt{ret}\ e$ returns a value of type τ, at any priority.

The constraint checking judgment is defined in Figure 7. 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 and priorities. 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* $\Gamma, \pi\ \texttt{prio} \vdash^R_\Sigma e : \tau$, *then* $[\rho/\pi]\Gamma \vdash^R_\Sigma [\rho/\pi]e : [\rho/\pi]\tau$.
(4) *If* $\Gamma, \pi\ \texttt{prio} \vdash^R_\Sigma m \mathrel{\dot\sim} \tau \mathbin{@} \rho$, *then* $[\rho'/\pi]\Gamma \vdash^R_\Sigma [\rho'/\pi]m \mathrel{\dot\sim} [\rho'/\pi]\tau \mathbin{@} [\rho'/\pi]\rho$.
(5) *If* $\Gamma, \pi\ \texttt{prio} \vdash^R C$, *then* $[\rho/\pi]\Gamma \vdash^R [\rho/\pi]C$.

See the supplementary material for the straightforward proof.

3.2 Dynamic Semantics

We define a transition semantics for λ^4. Because the operational behavior (as distinct from run-time or responsiveness, which will be the focus of Section 4) 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 will be 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 consists of two judgments. The judgment $v\ \texttt{val}_\Sigma$ states that v is a well-formed value and refers only to thread names in the signature Σ. The rules for this judgment are omitted. The transition relation for expressions $e \longrightarrow e'$ is fairly straightforward for a left-to-right, call-by-value lambda calculus and is shown in Figure 8. The signature Σ does not change during expression evaluation and is used solely to determine whether thread IDs are well-formed values. 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.

Define a thread pool μ to be a mapping of thread symbols to threads: $a \underset{\rho}{\hookrightarrow} m$ indicates a thread a at priority ρ running m. 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 $\uplus$[2]. We also introduce the additional congruence rules of Figure 10, which allow for thread name bindings to freely change scope within a thread pool.

Figure 9 gives the typing rules for thread pools. The typing judgment $\vdash^R_\Sigma \mu : \Sigma'$ indicates 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).

[2] 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 paper.

$$\textsc{D-Let-Step}\quad \frac{e_1 \longrightarrow e_1'}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \longrightarrow \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2} \qquad \textsc{D-Let}\quad \frac{v\ \mathtt{val}_\Sigma}{\mathtt{let}\ x = v\ \mathtt{in}\ e \longrightarrow [v/x]e}$$

$$\textsc{D-Ifz-Z}\quad \frac{}{\mathtt{ifz}\ 0\ \{e_1; x.e_2\} \longrightarrow e_1} \qquad \textsc{D-Ifz-NZ}\quad \frac{}{\mathtt{ifz}\ \overline{n+1}\ \{e_1; x.e_2\} \longrightarrow [\overline{n}/x]e_2} \qquad \textsc{D-App}\quad \frac{v\ \mathtt{val}_\Sigma}{(\lambda x.e)\ v \longrightarrow [v/x]e}$$

$$\textsc{D-Pair}\quad \frac{v_1\ \mathtt{val}_\Sigma \qquad v_2\ \mathtt{val}_\Sigma}{(v_1, v_2) \longrightarrow \langle v_1, v_2 \rangle} \qquad \textsc{D-Fst}\quad \frac{v_1\ \mathtt{val}_\Sigma \qquad v_2\ \mathtt{val}_\Sigma}{\mathtt{fst}\ \langle v_1, v_2 \rangle \longrightarrow v_1} \qquad \textsc{D-Snd}\quad \frac{v_1\ \mathtt{val}_\Sigma \qquad v_2\ \mathtt{val}_\Sigma}{\mathtt{snd}\ \langle v_1, v_2 \rangle \longrightarrow v_2}$$

$$\textsc{D-InL}\quad \frac{v\ \mathtt{val}_\Sigma}{\mathtt{inl}\ v \longrightarrow \mathtt{l} \cdot v} \qquad \textsc{D-InR}\quad \frac{v\ \mathtt{val}_\Sigma}{\mathtt{inr}\ v \longrightarrow \mathtt{r} \cdot v} \qquad \textsc{D-Case-L}\quad \frac{v\ \mathtt{val}_\Sigma}{\mathtt{case}\ \mathtt{l} \cdot v\ \{x.e_1; y.e_2\} \longrightarrow [v/x]e_1}$$

$$\textsc{D-Case-R}\quad \frac{v\ \mathtt{val}_\Sigma}{\mathtt{case}\ \mathtt{r} \cdot v\ \{x.e_1; y.e_2\} \longrightarrow [v/y]e_2} \qquad \textsc{D-Output}\quad \frac{}{\mathtt{output}\ \overline{n} \longrightarrow \langle\rangle} \qquad \textsc{D-Input}\quad \frac{n \in \mathbb{N}}{\mathtt{input} \longrightarrow \overline{n}}$$

$$\textsc{D-PrApp}\quad \frac{}{(\Lambda \pi : C.e)[\rho'] \longrightarrow [\rho'/\pi]e} \qquad \textsc{D-Fix}\quad \frac{}{\mathtt{fix}\, x{:}\tau\ \mathtt{is}\ e \longrightarrow [\mathtt{fix}\, x{:}\tau\ \mathtt{is}\ e/x]e}$$

Fig. 8. Dynamic semantics for expressions.

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

Fig. 9. Typing rules for thread pools

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

Fig. 10. Congruence rules for thread pools.

The transition judgment for commands is $m \overset{\alpha}{\underset{\Sigma}{\longmapsto}} (\Sigma', m', \mu')$, indicating that under signature Σ, command m steps to m'. The transition relation carries a label α, indicating the "action" taken by this step. At this point, actions can be the silent action ϵ or the sync action $b\ ?\ v$, indicating that the transition receives a value v by synchronizing on thread b. 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 11. 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

D-Bind1
$$\frac{e \longrightarrow e'}{x \leftarrow e; m \overset{\epsilon}{\underset{\Sigma}{\mapsto}} (\cdot, x \leftarrow e'; m, \emptyset)}$$

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

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

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

D-Sync1
$$\frac{e \longrightarrow e'}{\mathtt{sync}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}} (\cdot, \mathtt{sync}\ e', \emptyset)}$$

D-Sync2
$$\frac{v\ \mathtt{val}_\Sigma}{\mathtt{sync}\ (\mathtt{tid}[b]) \xmapsto[\Sigma]{b?v} (\cdot, \mathtt{ret}\ v, \emptyset)}$$

D-Ret
$$\frac{e \longrightarrow e'}{\mathtt{ret}\ e \overset{\epsilon}{\underset{\Sigma}{\mapsto}} (\cdot, \mathtt{ret}\ e', \emptyset)}$$

Fig. 11. Dynamic rules for commands.

DT-Thread
$$\frac{m \overset{\alpha}{\underset{\Sigma}{\mapsto}} (\Sigma', m', \mu')}{a \underset{\rho}{\hookrightarrow} m \xRightarrow[a\sim\tau@\rho,\Sigma]{a/\alpha} \nu\Sigma'\{a \underset{\rho}{\hookrightarrow} m' \uplus \mu'\}}$$

DT-Ret
$$\frac{v\ \mathtt{val}_{a\sim\tau@\rho,\Sigma}}{a \underset{\rho}{\hookrightarrow} \mathtt{ret}\ v \xRightarrow[a\sim\tau@\rho,\Sigma]{a/!v} a \underset{\rho}{\hookrightarrow} \mathtt{ret}\ v}$$

DT-Sync
$$\frac{\Sigma = \Sigma', a{\sim}\tau_a@\rho_a, b{\sim}\tau_b@\rho_b \qquad \mu_1 \xRightarrow[\Sigma]{a/b?v} \mu_1' \qquad \mu_2 \xRightarrow[\Sigma]{b/!v} \mu_2}{\mu_1 \uplus \mu_2 \xRightarrow[\Sigma]{a/\epsilon} \mu_1' \uplus \mu_2}$$

DT-Concat
$$\frac{\mu_1 \xRightarrow[\Sigma]{a/\alpha} \mu_1'}{\mu_1 \uplus \mu_2 \xRightarrow[\Sigma]{a/\alpha} \mu_1' \uplus \mu_2}$$

DT-Extend
$$\frac{\mu \xRightarrow[\Sigma, a\sim\tau@\rho]{b/\alpha} \mu'}{\nu a{\sim}\tau@\rho\{\mu\} \xRightarrow[\Sigma]{b/\alpha} \nu a{\sim}\tau@\rho\{\mu'\}}$$

DT-Par
$$\frac{(\forall 1 \le i \le n)\nu\Sigma\{\mu \uplus \mu_1 \uplus \ldots \uplus \mu_n\} \xRightarrow[\cdot]{a_i/\epsilon} \nu\Sigma\{\mu \uplus \mu_1 \uplus \ldots \uplus \mu_i' \uplus \ldots \uplus \mu_n\}}{\nu\Sigma\{\mu \uplus \mu_1 \uplus \ldots \uplus \mu_n\} \xRightarrow[P]{\{a_1,\ldots,a_n\}} \nu\Sigma\{\mu \uplus \mu_1' \uplus \ldots \uplus \mu_i' \uplus \ldots \uplus \mu_n'\}}$$

Fig. 12. Dynamic rules for thread pools.

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 $b\ ?\ v$. 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. 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 $\mu \xRightarrow[\Sigma]{a/\alpha} \mu'$ 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

$$\frac{}{\vdash^{R}_{\Sigma} \epsilon \text{ action}} \qquad \frac{\cdot \vdash^{R}_{\Sigma, b\sim\tau@\rho} v : \tau}{\vdash^{R}_{\Sigma, b\sim\tau@\rho} b \mathbin{?} v \text{ action}} \qquad \frac{\cdot \vdash^{R}_{\Sigma, b\sim\tau@\rho} v : \tau}{\vdash^{R}_{\Sigma, b\sim\tau@\rho} !v \text{ action}}$$

Fig. 13. Static semantics for actions.

now also include the "return" action $!v$, indicating that the thread returns the value v. Rule DT-Sync matches this with a corresponding sync action and performs 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. 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 $\nu\Sigma\{a_1 \underset{\rho_1}{\hookrightarrow} m_1 \uplus \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} m_n\}$ 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. This is expressed by the judgment $\mu \overset{A}{\underset{P}{\Rightarrow}} \mu'$, which allows all of the threads in the set A to step silently in parallel. The only rule for this judmgent is DT-Par, which steps any number of threads in a nondeterministic fashion. 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 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 13 and require that the value returned or received match the type of the thread.

Theorem 1 (Progress). (1) *If* $\cdot \vdash^{R}_{\Sigma} e : \tau$, *then either* e $\texttt{val}_{\Sigma}$ *or* $e \longrightarrow e'$.

(2) *If* $\cdot \vdash^{R}_{\Sigma} m \mathrel{\dot\sim} \tau \mathbin{@} \rho$, *then either* $m = \texttt{ret}\ e$ *where* e $\texttt{val}_{\Sigma}$ *or* $m \overset{\alpha}{\underset{\Sigma}{\longmapsto}} (\Sigma', m', \mu)$ *where* $\vdash^{R}_{\Sigma} \alpha$ 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* $\mu \equiv \nu\Sigma''\{a_1 \underset{\rho_1}{\hookrightarrow} m_1 \uplus \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} m_n\}$ *and for all* $i \in [1, n]$, *we have* $\mu \overset{a_i/\alpha}{\underset{\Sigma,\Sigma'}{\Longrightarrow}} \mu'$ *and* $\vdash^{R}_{\Sigma,\Sigma'} \alpha$ action.

See the supplementary material for the proof.

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 2 (Preservation). (1) *If* $\cdot \vdash^R_\Sigma e : \tau$ *and* $e \longrightarrow e'$, *then* $\cdot \vdash^R_\Sigma e' : \tau$.

(2) *If* $\cdot \vdash^R_\Sigma m \div \tau \;@\; \rho$ *and* $m \overset{\alpha}{\underset{\Sigma}{\longmapsto}} (\Sigma', m', \mu')$ *and* $\vdash^R_\Sigma \alpha$ action *then* $\cdot \vdash^R_{\Sigma,\Sigma'} m' \div \tau \;@\; \rho$ *and* $\vdash^R_\Sigma \mu' : \Sigma'$.

(3) *If* $\vdash^R_\Sigma \mu : \Sigma'$ *and* $\mu \overset{\alpha}{\underset{\Sigma}{\Longrightarrow}} \mu'$ *then* $\vdash^R_\Sigma \mu' : \Sigma'$

(4) *If* $\vdash^R_\cdot \mu : \Sigma$ *and* $\mu \overset{A}{\underset{P}{\Longrightarrow}} \mu'$ *then* $\vdash^R_\cdot \mu' : \Sigma$.

See the supplementary material for the proof.

Theorem 3 (Type Safety). *If* $\vdash^R_\cdot a_0 \underset{\rho_0}{\hookrightarrow} m_0 : a_0{\sim}\tau_0@\rho_0$ *and* $a_0 \underset{\rho_0}{\hookrightarrow} m_0 \underset{P}{\Longrightarrow}^* \mu'$, *then* $\mu' \equiv \nu a_1{\sim}\tau_1@\rho_1, \ldots, a_n{\sim}\tau_n@\rho_n\{a_0 \underset{\rho_0}{\hookrightarrow} m'_0 \uplus \ldots \uplus a_n \underset{\rho_n}{\hookrightarrow} m'_n\}$ *and for all* $i \in [1, n]$, *we have* $\mu' \overset{a_i/\alpha}{\underset{\cdot}{\Longrightarrow}} \mu''$.

Proof. By inductive application of Theorem 1 and Theorem 2. □

4 COST SEMANTICS

So far, we have presented a core calculus for writing parallel programs and expressing responsiveness requirements using priority annotations. We have not yet discussed how these requirements are met and what guarantees can be made. Doing so is the main theoretical contribution of the remainder of the paper. We will show how to derive cost bounds (both for computation time and response time) for λ^4 programs and then show that, under reasonable assumptions about scheduling, these bounds hold for the dynamic semantics of Section 3.2. We first (Section 4.1) develop a cost model for parallel programs with partially ordered thread priorities. The model comes equipped with bounds on computation times and response times. We then use this model (Sections 4.2 and 4.3) to reason about λ^4 programs.

4.1 A Cost Model for Prioritized Parallel Code

Parallel programs admit an elegant technique for reasoning about their execution time, in the form of Directed Acyclic Graph, or DAG models [Blelloch and Greiner 1995, 1996]. Such a model captures the dependences between threads in a program and, conversely, what portions may be parallelized. In DAG models of parallel programs, vertices represent units of sequential computation and edges represent sequential dependences. For example, an edge (u_1, u_2) indicates that the computation u_1 must run before u_2. If there is no directed path between u_1 and u_2, the two computations may run in parallel. Without loss of generality, it is typically assumed that each vertex represents a computation taking a single indivisible unit of time (perhaps a single processor clock cycle). These are the units in which we will measure execution time and response time.

Because threads play such an important role in the design of PriML (and λ^4) programs, it will be helpful for us to distinguish in the DAG model between edges that represent continuations of threads and edges that represent synchronizations between threads. In our model, a thread is a sequence of vertices $\vec{u} = u_1 \cdot u_2 \cdot u_3 \cdot \ldots \cdot u_n$, written [] when $n = 0$, representing a sequence of unit-time operations that are connected by a series of edges $(u_1, u_2), (u_2, u_3), \ldots, (u_{n-1}, u_n)$ representing sequential dependences. These are referred to as *thread edges* and ensure that the operations of a thread are performed in the proper sequence.

We then combine threads into a DAG, $g = (\mathcal{T}, E_s, E_j)$, in which $\mathcal{T}$ is a mapping from thread names to a pair consisting of that thread's priority and its sequence of operations. We write an element of the mapping as $a \underset{\rho}{\hookrightarrow} \vec{u}$, and we define $Prio_g(u)$ as the priority of the thread to which u belongs. The other two components of a DAG are the sets of *spawn edges*, E_s, and *join edges*, E_j. A spawn edge (u, a) indicates that a vertex u spawned a thread a. It may be considered an edge from u to the first vertex of a. A join edge (a, u) indicates that vertex u syncs (joins) with thread a. It may be considered an edge from the last vertex of a to vertex u.

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 will define shorthands for a graph with the (proper) ancestors and descendants of a vertex u removed:

$$\not\uparrow u \triangleq g \setminus \{u' \neq u \mid u' \sqsupseteq u\}$$
$$\not\downarrow u \triangleq g \setminus \{u' \neq u \mid u \sqsupseteq u'\}$$

The *competitor work*, $\not\updownarrow a$, 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 $g = (a \underset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t, E_s, E_j)$, then

$$\not\updownarrow a \triangleq g \setminus \{u \neq s \mid u \sqsupseteq s\} \setminus \{u \neq t \mid t \sqsupseteq u\}$$

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

The Prompt Scheduling Principle. A *schedule* of a DAG simulates the execution of a parallel program on a given number of processors. The execution proceeds in time steps, each one time unit in length. At each time step, the schedule designates some number of vertices to be executed, bounded by the available number of processors, P. A schedule may only execute a vertex if it is *ready*, that is, if all of its ancestors in the DAG have been executed.

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 [Eager et al. 1989], but greediness is insufficient to place bounds on response time. To provide such bounds, a schedule must take into account the thread priorities. A *prompt* schedule [Muller et al. 2017] is a greedy schedule that prioritizes vertices according to their priority, with high priorities preferred over low. Prompt schedules have previously only been used in languages with two priorities, so more care is required to apply them to an arbitrary partial order. At each step, we will 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 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.

Response Time. Our goal is to show a bound on the response time of threads in prompt schedules. In a given schedule, the response time of a thread a, written $T(a)$, is defined as the number of steps from when s is ready (exclusive) to when t is executed (inclusive). If our definitions of priority are set up correctly, the response time of a thread a at priority ρ should depend only on the amount of work at priorities greater than, equal to, or unrelated to ρ in the partial order. Were the response time of a high-priority thread to depend on the amount of low-priority work in the computation, there would be a priority inversion in the schedule, a condition to be avoided.

Well-formed DAGs. To prove a bound on response time that depends only on work at priorities not less than ρ, we will need to place an additional restriction on DAGs. Consider a DAG with two threads, $a \underset{\rho_a}{\hookrightarrow} u_1 \cdot \ldots \cdot u \cdot \ldots \cdot u_n$ and $b \underset{\rho_b}{\hookrightarrow} \vec{u}_b$, where $\rho_b \prec \rho_a$. Suppose there is a join edge (b, u) from b to a. Thread a will need to wait for b to complete, so the response time of a depends on the length of thread b. The type system given earlier is designed to rule out such inversions in programs; we must impose a similar restriction on computation DAGs.

[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.

A DAG is *well-formed* if no thread depends on lower-priority work along its critical path. More precisely, if a thread a consists of operations $u_1 \cdot \ldots \cdot u_n$, no vertex that may be executed after u_1 and must be executed before u_n may have a priority less than that of a.

Definition 1. A DAG $g = (\mathcal{T}, E_s, E_j)$ is well-formed if for all threads $a \underset{\rho}{\hookrightarrow} u_1 \cdot \ldots \cdot u_n \in \mathcal{T}$, if $u \sqsupseteq u_n$ and $u \not\sqsupseteq u_1$ then $\rho \preceq \mathit{Prio}_g(u)$.

We will show that the well-formedness restriction on DAGs and the type restrictions imposed on λ^4 programs coincide in that well-typed programs give rise only to well-formed DAGs. 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 2. A DAG $g = (\mathcal{T}, E_s, E_j)$ is *strongly well-formed* if for all $(a, u) \in E_j$, if $a \underset{\rho_a}{\hookrightarrow} \vec{u}, b \underset{\rho_b}{\hookrightarrow} \vec{u}_1 \cdot u \cdot \vec{u}_2 \in \mathcal{T}$, we have that

(1) $\rho_b \preceq \rho_a$ and
(2) If $(u', a) \in E_s$, then there exists a path from u' to u where the first edge is a thread edge.

Lemma 4.1. *If g is strongly well-formed, then g is well-formed.*

Proof. Let $a \underset{\rho}{\hookrightarrow} u_1 \cdot \ldots \cdot u_n \in \mathcal{T}$ and let $u \sqsupseteq u_n$. We need to show that either $u \sqsupseteq u_1$ or $\rho \preceq \mathit{Prio}_g(u)$. 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 $\rho \preceq \mathit{Prio}_g(u')$. If $u' \sqsupseteq u_1$ for any such u', then $u \sqsupseteq u_1$, so consider the case where $\rho \preceq \mathit{Prio}_g(u')$ for all such u'. Consider the outgoing edges of u which lead to u' such that $u' \sqsupseteq u_n$. If any is a thread or join edge, then we have $\rho \preceq \mathit{Prio}_g(u') \preceq \mathit{Prio}_g(u)$. Suppose the only such edge is a spawn edge (u, b), 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 thread edge, but this is a contradiction because the spawn edge (u, b) was assumed to be the only outgoing edge from u to an ancestor of u_n. 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. □

Bounding Response Time. We are now ready to bound the response time of threads in prompt schedules using cost metrics that we now define.

The *priority work* $W_{\not\prec\rho}(g)$ of a graph g at a priority ρ is defined as the number of vertices in the graph at priorities not less than ρ:

$$W_{\not\prec\rho}(g) \triangleq |\{u \in g \mid \mathit{Prio}_g(u) \not\prec \rho\}|$$

The *a-span* $S_a(g)$ of a graph $g \ni a \underset{\rho}{\hookrightarrow} s \cdot \ldots \cdot t$, is the length of the longest path in g ending at t.

Theorem 4 bounds the response time of a thread based on these quantities which depend only on the work and span of high-priority threads. Because they deal with scheduling DAGs which are known ahead of time, results of this form are often known as *offline scheduling bounds.* Later in the section, we will apply this result to executions of the λ^4 dynamic semantics as well.

Theorem 4. *Let g be a well-formed DAG with a thread $a \underset{\rho}{\hookrightarrow} \vec{u} \in g$. For any prompt schedule of g on P processors,*

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

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 is 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 B_h; otherwise, place a token in the "low" bucket B_l. 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 buckets after t is executed.

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

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

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

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 path ending at t consisting of vertices of $g \setminus Exec(j)$, the path starts at a vertex that is ready at the beginning of step j. By the definition of well-formedness, this vertex must have priority greater than ρ and is therefore executed in step j by the prompt principle. Thus, the length of the path decreases by 1 and so $S_a(g \setminus Exec(j+1)) = S_a(g \setminus Exec(j)) - 1$. The maximum number of such steps is thus $S_a(g \setminus Exec(0))$, and so $B_l \leq P \cdot S_a(g \setminus Exec(0))$. Because $\uparrow s \supset g \setminus Exec(0)$, any path excluding vertices in $Exec(0)$ is contained in $\uparrow s$, and $S_a(g \setminus Exec(0)) \leq S_a(\uparrow s)$, so $B_l \leq P \cdot S_a(\uparrow s) = P \cdot S_a(\updownarrow a)$. □

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 [Eager et al. 1989]. Indeed, the priority work and a−span reduce to the overall work and span, respectively, so the bound given by Theorem 4 coincides with the expected bound on computation time.

4.2 Cost Semantics for λ^4

We develop a cost semantics that evaluates a program, producing a value and a DAG of the form described in Section 4.1. Unlike the operational semantics of Section 3.2, this is an evaluation semantics that does not fully specify the order in which threads are evaluated. Figure 14 shows the cost semantics for λ^4 using three judgments. The judgment for expressions is $e \downarrow v; \vec{u}$, indicating that expression e evaluates to value v and produces thread $\vec{u}$. 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 this judgment are quite straightforward: subexpressions are evaluated to produce sequences of operations, which are then composed sequentially. The judgment $\sigma; \Sigma; m \downarrow_{(a,\rho)} v; g; \sigma'; \Sigma'$ indicates that m evaluates to `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 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 rule C-Sync to capture the value

C-Val

$$\frac{}{v \downarrow v; []}$$

C-Let

$$\frac{e_1 \downarrow v_1; \vec{u}_1 \qquad [v_1/x]e_2 \downarrow v; \vec{u}_2 \qquad u\ \mathtt{fresh}}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \downarrow v; \vec{u}_1 \cdot u \cdot \vec{u}_2}$$

C-Ifz-NZ

$$\frac{[\overline{n}/x]e_2 \downarrow v; \vec{u} \qquad u\ \mathtt{fresh}}{\mathtt{ifz}\ \overline{n+1}\ \{e_1; x.e_2\} \downarrow v; u \cdot \vec{u}}$$

C-Ifz-Z

$$\frac{e_1 \downarrow v; \vec{u} \qquad u\ \mathtt{fresh}}{\mathtt{ifz}\ \overline{0}\ \{e_1; x.e_2\} \downarrow v; u \cdot \vec{u}}$$

C-App

$$\frac{[v/x]e \downarrow v'; \vec{u} \qquad u\ \mathtt{fresh}}{(\lambda x.e)\ v \downarrow v'; u \cdot \vec{u}}$$

C-Pair

$$\frac{u\ \mathtt{fresh}}{(v_1, v_2) \downarrow \langle v_1, v_2 \rangle; u}$$

C-Fst

$$\frac{u\ \mathtt{fresh}}{\mathtt{fst}\ \langle v_1, v_2 \rangle \downarrow v_1; u}$$

C-Snd

$$\frac{u\ \mathtt{fresh}}{\mathtt{snd}\ \langle v_1, v_2 \rangle \downarrow v_2; u}$$

C-InL

$$\frac{u\ \mathtt{fresh}}{\mathtt{inl}\ v \downarrow \mathtt{l} \cdot v; u}$$

C-InR

$$\frac{u\ \mathtt{fresh}}{\mathtt{inr}\ v \downarrow \mathtt{r} \cdot v; u}$$

C-Case-L

$$\frac{[v/x]e_1 \downarrow v'; \vec{u} \qquad u\ \mathtt{fresh}}{\mathtt{case}\ \mathtt{l} \cdot v\ \{x.e_1; y.e_2\} \downarrow v'; u \cdot \vec{u}}$$

C-Case-R

$$\frac{[v/y]e_2 \downarrow v'; \vec{u} \qquad u\ \mathtt{fresh}}{\mathtt{case}\ \mathtt{r} \cdot v\ \{x.e_1; y.e_2\} \downarrow v'; u \cdot \vec{u}}$$

C-Output

$$\frac{u\ \mathtt{fresh}}{\mathtt{output}\ v \downarrow \langle\rangle; u}$$

C-Input

$$\frac{u\ \mathtt{fresh}}{\mathtt{input} \downarrow \overline{n}; u}$$

C-PrApp

$$\frac{[\rho/\pi]e \downarrow v; \vec{u} \qquad u\ \mathtt{fresh}}{(\Lambda\pi : C.e)\ \rho \downarrow v; u \cdot \vec{u}}$$

C-Fix

$$\frac{[v/x]e \downarrow v'; \vec{u} \qquad u\ \mathtt{fresh}}{\mathtt{fix}\, x{:}\tau\ \mathtt{is}\ e \downarrow v'; u \cdot \vec{u}}$$

C-Bind

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

C-Spawn

$$\frac{b\ \mathtt{fresh} \qquad \sigma; \Sigma; m \downarrow_{(b,\rho')} v; (\mathcal{T}, E_s, E_j); \sigma, \sigma'; \Sigma, \Sigma' \qquad u\ \mathtt{fresh}}{\sigma; \Sigma; \mathtt{spawn}[\rho'; \tau]\ \{m\} \downarrow_{(a,\rho)} \mathtt{tid}[b]; (a \underset{\rho}{\hookrightarrow} u \uplus \mathcal{T}, E_s \cup \{(u,b)\}, E_j); \sigma, \sigma', b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'}$$

C-Sync

$$\frac{e \downarrow \mathtt{tid}[b]; \vec{u} \qquad u\ \mathtt{fresh}}{\sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho'; \mathtt{sync}\ e \downarrow_{(a,\rho)} v; (a \underset{\rho}{\hookrightarrow} \vec{u} \cdot u, \emptyset, \{(b,u)\}); \sigma, b \hookrightarrow (v, \Sigma'); \Sigma, b{\sim}\tau@\rho', \Sigma'}$$

C-Ret

$$\frac{e \downarrow v; \vec{u}}{\sigma; \Sigma; \mathtt{ret}\ e \downarrow_{(a,\rho)} v; (a \underset{\rho}{\hookrightarrow} \vec{u}, \emptyset, \emptyset); \sigma; \Sigma}$$

CT-Thread

$$\frac{\sigma; \Sigma; m \downarrow_{(a,\rho)} v; g; \sigma'; \Sigma'}{\sigma, a \hookrightarrow (v, \Sigma'); \Sigma; a \underset{\rho}{\hookrightarrow} m \downarrow g; \sigma'}$$

CT-Extend

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

CT-Concat

$$\frac{\sigma; \Sigma; \mu \downarrow (\mathcal{T}, E_s, E_j); \sigma_1 \qquad \sigma; \Sigma; \mu' \downarrow (\mathcal{T}', E_s', E_j'); \sigma_2}{\sigma; \Sigma; \mu \uplus \mu' \downarrow (\mathcal{T} \uplus \mathcal{T}', E_s \cup E_s', E_j \cup E_j'); \sigma_1, \sigma_2}$$

Fig. 14. Cost semantics of λ^4

of the target thread b, which must be returned by the sync operation. The rule also captures the signature of threads transitively spawned by b, which it adds 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 section, 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:

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

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

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

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.

Finally, the judgment $\sigma; \Sigma; \mu \downarrow g; \sigma'$ 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 $a \underset{\rho}{\hookrightarrow} m$ evaluates to the same graph as the command m. Rule CT-Concat evaluates both parts of the thread pool and composes the graphs, giving each access to the thread records of the other.

Lemma 4.2 shows that the evaluation judgment on expressions preserves typing. The equivalent property for commands will be shown as part of Lemma 4.4.

Lemma 4.2. *If* $\cdot \vdash^R_\Sigma e : \tau$ *and* $e \downarrow v; \vec{u}$, *then* $\cdot \vdash^R_\Sigma v : \tau$.

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

One more technical result we will need in Section 4.3 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 4.3. (1) *If* $\cdot \vdash^R_\Sigma m \div \tau \ @ \ \rho$ *and* $\sigma, c \hookrightarrow (v_c, \Sigma_c); \Sigma; m \downarrow_{(a,\rho)} v; g; \sigma', c \hookrightarrow (v_c, \Sigma_c); \Sigma'$ *and* $c \notin dom(\Sigma)$, *then* $\sigma; \Sigma; m \downarrow_{(a,\rho)} v; g; \sigma'; \Sigma'$.

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

Proof. (1) By induction on the derivation of $\sigma, c \hookrightarrow (v_c, \Sigma_c)\rho'; \Sigma; m \downarrow_{(a,\rho)} v; g; \sigma', c \hookrightarrow (v_c, \Sigma_c); \Sigma'$.

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

□

We now show that well-typed programs produce strongly well-formed cost graphs. 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 3. We say that a graph $g = (\mathcal{T}, E_s, E_j)$ is compatible with a signature Σ at a if

(1) $a \underset{\rho_a}{\hookrightarrow} \vec{u}_a \cdot t_a \in \mathcal{T}$
(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.

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 posit 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".

Lemma 4.4. *If* $\cdot \vdash^R_\Sigma m \div \tau \ @ \ \rho$ *and* $\vdash^R_\Sigma \sigma$ *and* $\sigma; \Sigma; m \downarrow_{(a,\rho)} v; g; \sigma'; \Sigma'$ *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) $g = (a \underset{\rho}{\hookrightarrow} \vec{u} \uplus \mathcal{T}, E_s, E_j)$
(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 $\sigma; \Sigma; m \downarrow_{(a,\rho)} v; g; \sigma'; \Sigma'$. □

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 1. *If* $\cdot \vdash^R_\cdot m \div \tau \ @ \ \rho$ *and* $\cdot; \cdot; m \downarrow_{(a,\rho)} v; g; \sigma; \Sigma$, *then* g *is well-formed.*

Proof. Because $\emptyset$ is strongly well-formed and compatible with $\cdot$, Lemma 4.4 shows that g is strongly well-formed, and is thus well-formed by Lemma 4.1. □

4.3 Response Time Bound for Operational Semantics

Thus far in this section, 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-typed programs admit the cost bound of Theorem 4, we may then directly appeal to that theorem for cost bounds on programs. The argument proceeds as follows:

(1) Lemmas 4.5 and 4.6 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) Lemma 4.7 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.
(3) Lemma 4.8 combines the above results to establish a correspondence between an execution of a λ^4 program and a schedule of its cost graph.
(4) Finally, we use Theorem 4 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 4.5 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 a ready 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. Parts (3) and (4) extend part (2) to thread pools. 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.

Lemma 4.5. (1) *If* $\cdot \vdash^R_\Sigma e : \tau$ *and* $e \downarrow v; \vec{u}$, *then* $e \longrightarrow e'$ *for some* e' *if and only if* $\vec{u}$ *is nonempty.*

(2) *If* $\cdot \vdash^R_\Sigma m \div \tau \mathbin{@} \rho$ *and* $\sigma; \Sigma; m \downarrow_{(a,\rho)} v; g; \sigma'; \Sigma'$, *then* $g = (a \underset{\rho}{\hookrightarrow} \vec{u} \uplus \mathcal{T}, E_s, E_j)$, *and* g *has no spawn edges to threads in* Σ *and has no join edges to active threads other than* a, *and one of the following is true:*

(a) *There exists* m' *such that* $m \underset{\Sigma}{\overset{\epsilon}{\mapsto}} m'$ *and* $\vec{u} = u \cdot \vec{u}'$ *and* u *is ready in* g.

(b) *There exists* v *such that* v $\mathtt{val}_\Sigma$ *and* $m = \mathtt{ret}\ v$ *and* $\vec{u} = []$.

(c) *There exist* v *and* m' *such that* $m \xmapsto[\Sigma]{b?v} m'$ *and* $\vec{u} = u \cdot \vec{u}'$ *and there exists an edge* $(b, u) \in g$, *which is the only in-edge of* u.

(3) *If* $\vdash^R_\Sigma \mu : \Sigma', a{\sim}\tau@\rho$ *and* $\sigma; \Sigma; \mu \downarrow g; \sigma'$ *where* $g = (\mathcal{T}, E_s, E_j)$ *and* $\mu \xRightarrow[\Sigma]{a/\alpha} \mu'$, *then* g *has no spawn or join edges to threads not in* $\mathcal{T}$ *and*

(a) *If* $\alpha = \epsilon$, *then* $a \underset{\rho}{\hookrightarrow} u \cdot \vec{u} \in \mathcal{T}$ *and* u *is ready in* g.

(b) *If* $\alpha = !v$, *then* $a \underset{\rho}{\hookrightarrow} [] \in \mathcal{T}$.

(c) *If* $\alpha = b\,?\,v$, *then* $a \underset{\rho}{\hookrightarrow} u \cdot \vec{u} \in \mathcal{T}$ *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* $\sigma; \Sigma; \mu \downarrow g; \sigma'$ *and the first vertex of* a *is ready in* g, *then there exists* μ' *such that* $\mu \xRightarrow[\cdot]{a/\epsilon} \mu'$.

The full proof is available in the supplementary material.

Parts (3) and (4) of Lemma 4.5 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 4.6 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 4.6. *Let* $R = \{a \mid a \underset{\rho}{\hookrightarrow} u \cdot \vec{u} \in g, u \text{ is ready in } g\}$. *If* $\vdash^R_\Sigma \mu :$ *and* $\sigma; \Sigma; \mu \downarrow g; \sigma'$, *then*

(1) *For any subset $\{a_1, \ldots, a_n\}$ of R, we have $\mu \xRightarrow[P]{\{a_1,\ldots,a_n\}} \mu'$.*

(2) *If $\mu \xRightarrow[P]{\{a_1,\ldots,a_n\}} \mu'$, then $\{a_1, \ldots, a_n\} \subset R$.*

Proof. (1) By Theorem 3, we have $\mu \equiv \nu\Sigma'\{a_1 \underset{\rho_1}{\hookrightarrow} m_1 \uplus \ldots \uplus a_m \underset{\rho_m}{\hookrightarrow} m_m\}$. For all $a_i \in R$, we have that by Lemma 4.5, $\mu \xRightarrow[\Sigma]{a_i/\epsilon} \mu'_i$. A straightforward induction on $\mu \xRightarrow[\Sigma]{a_i/\epsilon} \mu'_i$ shows that $\mu'_i \equiv \nu\Sigma''\{a_1 \underset{\rho_1}{\hookrightarrow} m_1 \uplus \ldots \uplus a_i \underset{\rho_i}{\hookrightarrow} m'_i \uplus \mu''_i \uplus \ldots \uplus a_m \underset{\rho_m}{\hookrightarrow} m_m\}$. Applying this reasoning to all $a_i \in \{a_1, \ldots, a_n\}$ allows us to apply rule DT-Par.

(2) Let $i \in [1, n]$. By inversion on rule DT-Par, $\mu \xRightarrow[\cdot]{a_i/\epsilon} \mu'_i$. By Lemma 4.5, $a_i \in R$.

□

We now move on to showing that a parallel transition corresponds to a step of a schedule. At a more precise level, Lemma 4.7 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).

$$\begin{aligned}
& [\vec{u}]\overline{\oplus}_a(a \underset{\rho}{\hookrightarrow} \vec{u}' \uplus a_1 \underset{\rho_1}{\hookrightarrow} \vec{u}_1 \cdots \uplus a_n \underset{\rho_n}{\hookrightarrow} \vec{u}_n, E_s, E_j) \\
\triangleq\ & (a \underset{\rho}{\hookrightarrow} \vec{u} \cdot \vec{u}' \uplus a_1 \underset{\rho_1}{\hookrightarrow} \vec{u}_1 \cdots \uplus a_n \underset{\rho_n}{\hookrightarrow} \vec{u}_n, E_s \cup \{(u, a_1), \ldots, (u, a_n)\}, E_j)
\end{aligned}$$

Lemma 4.7. (1) *If $e' \downarrow v; \vec{u}$ and $e \to e'$, then $e \downarrow v; u \cdot \vec{u}$.*

(2) *If $\sigma; \Sigma; a \underset{\rho}{\hookrightarrow} m' \uplus \mu' \downarrow g; \sigma''$ and $m \overset{\alpha}{\underset{\Sigma}{\mapsto}} (\Sigma', m', \mu')$, then $\sigma; \Sigma; m \downarrow_{(a,\rho)} v; g_0; \sigma'; \Sigma'$, where g_0 is isomorphic to $[u]\overline{\oplus}_a g$.*

(3) *If $\sigma; \Sigma; \mu' \downarrow g'; \sigma'$ and $\mu \xRightarrow[P]{\{a_i/\epsilon,\ldots,a_n/\epsilon\}} \mu'$, then g' can be decomposed into $g_0 \uplus g'_1 \uplus \cdots \uplus g'_n$, and $\sigma; \Sigma; \mu \downarrow g; \sigma'$, where g is isomorphic to $g_0 \uplus ([u_1]\overline{\oplus}_{a_1} g_1) \uplus \cdots \uplus ([u_n]\overline{\oplus}_{a_n} g_n)$.*

See the supplementary material for the proof, which is by induction on the transition derivation.

We can now repeatedly apply the above results to show a step-by-step correspondence between arbitrary 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 "prompt" manner by stepping as many threads as possible and prioritizing high-priority threads, then the corresponding schedule is prompt. Specifying how to pick the threads in a parallel transition is out of the scope of this paper, though we briefly discuss an appropriate scheduling algorithm at an implementation level in Section 6.

Lemma 4.8. *Suppose $\vdash^R_{\cdot} \mu :$ and $\mu \underset{P}{\Rightarrow}^* \mu'$ where $\cdot; \cdot; \mu' \downarrow \emptyset; \cdot$ and thread a is active for T transitions and at each transition, threads are chosen in a prompt manner. Then $\cdot; \cdot; \mu \downarrow g; \cdot$ and there exists a prompt schedule of g in which $T(a) = T$.*

Proof. By induction on the derivation of $\mu \underset{P}{\Rightarrow}^* \mu'$. If $\mu = \mu'$, then the result is clear. Suppose $\mu \xRightarrow[P]{\{a_1,\ldots,a_n\}} \mu'' \underset{P}{\Rightarrow}^* \mu'$, and a is active for T transitions of the latter execution. By induction, $\cdot; \cdot; \mu'' \downarrow$

$g''; \cdot$ and there exists a prompt schedule of g'' where $T(a) = T$. By Lemma 4.7, g'' is isomorphic to $g_0 \uplus g'_1 \dots g'_n$ and $\cdot; \cdot; \mu \downarrow g; \cdot$, where g is isomorphic to $g_0 \uplus ([u_1]\overline{\oplus}_{a_1} g'_1) \uplus \cdots \uplus ([u_n]\overline{\oplus}_{a_n} g'_n)$. By Lemma 4.6, these threads are ready in g, so the schedule that executes $u_1, \dots, u_n$ in step 1 and then follows the schedule of g'' is a valid schedule of g. Because (also by Lemma 4.6), 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. If $a \in dom(\mu)$, then by Lemma 4.6, 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 4 to bound the response time of prompt schedules, and therefore of the corresponding executions of the operational semantics.

Theorem 5. *If* $\cdot \vdash^R_{\cdot} m \mathrel{\dot{\sim}} \tau \mathbin{@} \rho$ *and* $a \underset{\rho}{\hookrightarrow} m \underset{P}{\Longrightarrow}^* \mu'$, *where* $\cdot; \cdot; \mu' \downarrow \emptyset; \cdot$ *and thread a is active for T transitions and at each transition, threads are chosen in a prompt manner, then there exists a graph g such that* $\cdot; \cdot; m \downarrow_{(a,\rho)} v; g; \sigma; \Sigma$ *and*

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

Proof. By Lemma 4.8, there exists such a g and a prompt schedule of g where $T(a) = T$. By Lemma 4.4, g is well-formed. Thus, the result follows from Theorem 4. □

5 STARVATION AND FAIRNESS

Throughout this paper, we assume that higher-priority threads should always be given priority over lower-priority ones. This is the desired semantics in many applications, but not all: sometimes, it is important to be *fair* and devote a certain fraction of cycles to lower-priority work. Fairness raises a number of interesting theoretical and practical questions the full treatment of which are beyond the scope of this paper. We note, however, that fairness is largely orthogonal to our results and it is not difficult to extend our results (e.g., those in Section 4.1) to devote a fraction L of processor cycles to lower-priority work. This simply inflates the response time bounds by a factor of $\frac{1}{1-L}$ to account for time not devoted to being prompt. A discussion of cost bounds accounting for fairness can be found in an appendix in the attached supplementary material.

6 IMPLEMENTATION

We have developed a prototype implementation of PriML. Our implementation compiles PriML to `mlton-parmem` [Raghunathan et al. 2016], a parallel extension of Standard ML which is derived from the work of Spoonhower [2009]. We have also developed a parallel scheduler for PriML programs, which plugs into the `mlton-parmem` runtime. The implementation allows programmers to use almost all of the features of Standard ML, including datatype declarations, higher-order functions, pattern matching, and so on. While PriML itself does not have a module system and expects all PriML code to be in one file (a limitation we inherit from the compiler on whose elaborator we build), our implementation is designed so that code may freely interface with the Standard ML basis library and SML modules defined elsewhere.

We will describe the two components of the implementation (compilation to parallel ML and the scheduler) separately.

6.1 Compilation to Parallel ML

Our compiler modifies the parser and elaborator of ML5/pgh [Murphy 2008], 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. At the same time, the elaborator collects the priority and ordering declarations into a set of worlds and a set of ordering constraints (raising a type error if inconsistent ordering declarations ever cause a cycle in the ordering relation).

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 SML with the parallel extensions of `mlton-parmem`. Priority names and variables are converted into ordinary SML variables. 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 (so as to preserve the semantics that they are delayed). We compile threads using Spoonhower's original implementation of parallel futures: spawn commands spawn a future, and sync commands force the future.

The AST generated by the above process is then prefaced by a series of declarations which register all of the priorities and ordering constraints with the runtime, and bind the priority names to the generated priorities. The compiler finally generates Standard ML code from the AST, and passes it to `mlton-parmem` for compilation to an executable.

6.2 Runtime and Scheduler

The runtime for PriML is written in Standard ML as a scheduler for `mlton-parmem`. As described above, before executing the program code, PriML programs call into the runtime to register the necessary priorities and orderings. The runtime then uses Warshall's transitive closure algorithm to build the full partial order and stores the result, so that checking the ordering on two priorities at runtime is a constant-time operation. It then performs a topological sort on the priorities to convert the partial order into a total order which is compatible with all of the ordering constraints. Once this is complete, the program runs.

In our scheduling algorithm, each processor has a private deque [Acar et al. 2013] of tasks for each priority, ordered by the total order computed above. Each processor works on its highest-priority task (in the total order, which guarantees it has no higher-priority task in the partial order). A busy processor q_1 will periodically preempt its work and pick another "target" processor q_2 at random. Processor q_1 will send work to q_2 at an arbitrarily chosen priority, if q_2 has no work at that priority. It will then start the process over by finding its highest-priority task (which may have changed if another processor has sent it work) and working on it.

6.3 Examples

We have implemented five sizable programs in PriML. These include the email client of Section 2 and a bank example inspired by an example used to justify partially-ordered priorities [Babaoğlu et al. 1993]. We have also adapted the Fibonacci server, streaming music and web server benchmarks of our prior work [Muller et al. 2017]. 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 2. 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. [1993] 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. Like the server of the music service, 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`) periodically traverses the request logs and analyzes them (currently, the analysis consists of calculating the number of views per page, together with a large Fibonacci computation to simulate a larger job). Both `accept_p` and `serve_p` are higher-priority than `stat_p`, but the ordering between them is unspecified.

6.3.1 Evaluation. While a performance evaluation is outside the scope of this paper, we have completed a preliminary performance evaluation of the scheduler described above. We have evaluated the performance of the web server benchmark described above, as well as a number of smaller benchmarks which allow for more controlled experiments and comparisons to prior work. In all cases, we have observed good performance and scaling. The web server, for example, scales easily to 50 processors and 50 concurrent requests while keeping response times to 6.5 milliseconds.

7 RELATED WORK

In this section, we review some of the most closely related papers from fields such as multithreading and modal type systems, and discuss their relationship with our work.

Multithreading and priorities. Multithreaded programming goes back to the early days of computer science, such as the work on Mesa [Lampson and Redell 1980], Xerox's STAR [Smith et al. 1982], and Cedar [Swinehart et al. 1986]. These systems allow the programmer to create ("fork") threads, and synchronize ("join") with running threads. The programmer can assign priorities, generally chosen from a fixed set of natural numbers (e.g., 7 in Cedar), to threads, allowing those that execute latency-sensitive computations to have a greater share of resources such as the CPU.

Our notion of priorities is significantly richer than those considered in prior work, because we allow the programmer to create as many priorities as needed, and impose an arbitrary partial order on them. Several authors have observed that partial orders are more expressive and more desirable for programming with priorities than total orders [Babaoğlu et al. 1993; Fidge 1993]. There is little prior work on programming language support for partially ordered priorities. The only one we know of is the occam language, whose expressive power is limited, leaving the potential for ambiguities in priorities [Fidge 1993].

Some languages, such as Concurrent ML [Reppy 1999], don't expose priorities to the programmer, but give higher priority at runtime to threads that perform certain (e.g. interactive) operations.

Priority Inversion. Priority inversion is a classic problem in multithreading systems. Lampson and Redell [1980] appear to be the first to observe it in their work on Mesa. Their original description of the problem uses three threads with three different priorities, but the general problem can be restated using just two threads at different priorities (e.g. [Babaoğlu et al. 1993]).

Babaoğlu, Marzullo, and Schneider provide a formalization of priority inversions and describe protocols for preventing them in some settings, e.g. transactional systems.

Parallel Computing. Although earlier work on multithreading was driven primarily by the need to develop interactive systems [Hauser et al. 1993], multithreading has also become an important paradigm for parallel computing. In principle, a multithreading system such as pthreads can be used to perform parallel computations by creating a number of threads and distributing the work of the computation among them. This approach, sometimes called "flat parallelism," has numerous disadvantages and has therefore given way to a higher-level approach, sometimes called "implicit threading", in which the programmer indicates the computations that can be performed in parallel using constructs such as "fork" and "join". The language runtime system creates and manages the threads as needed. In addition to the focus on throughput rather than responsiveness, cooperative systems differ from competitive systems in that they typically handle many more, lighter-weight threads. The ideas of implicit and cooperative threading go back to early parallel programming languages such as Id [Arvind and Gostelow 1978] and Multilisp [Halstead 1985], and many languages and systems have since been developed.

Cost Semantics. Cost semantics, broadly used to reason about resource usage [Rosendahl 1989; Sands 1990], have been deployed in many domains. We build in particular on cost models that use DAGs to reason about parallel programs [Blelloch and Greiner 1995, 1996; Spoonhower et al. 2008]. These models summarize the parallel structure of a computation in the cost metrics of *work* and *span*, which can then be used to bound computation time. While finding an optimal schedule of a DAG is NP-hard [Ullman 1975], Brent [1974] showed that a "level-by-level" schedule is within a factor of two of optimal. Eager et al. [1989] extended this result to all greedy schedules.

While these models have historically been applied to cooperatively threaded programs, in recent work we have extended them to handle latency-incurring operations [Muller and Acar 2016], and presented a DAG model which enables reasoning about responsiveness in addition to computation time [Muller et al. 2017]. This prior work introduced the idea of a prompt schedule, but considers only two priorities. Our cost semantics in this paper applies to programs with a partially ordered set of priorities.

Modal and Placed Type Systems. A number of type systems have been based on various modal logics, many of them deriving from the judgmental formulation of Pfenning and Davies [2001].

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 [2003] 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 [Jia and Walker 2004; Murphy et al. 2007] 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 $A \ \mathtt{at} \ w$, 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 $\tau \ \mathtt{cmd}[\rho]$ and $\tau \ \mathtt{thread}[\rho]$ 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. [Davies 1996; Feltman et al. 2016]). 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 [Muller et al. 2017] we adapted these ideas to a type system for prioritized computation with 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" of LTL), such systems would be limited to totally ordered priorities.

Place-based systems (e.g. [Chandra et al. 2008; Charles et al. 2005; Yelick et al. 1998]), 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.

8 CONCLUSION

We present techniques for writing parallel interactive programs where threads can be assigned partially ordered priorities. A type system ensures proper usage of priorities by precluding priority inversions and a cost model enables predicting the responsiveness and completion time properties for programs. We implement these techniques by extending the Standard ML language and show a number of example programs. Our experiments provide preliminary evidence that the proposed techniques can be effective in practice.

ACKNOWLEDGEMENTS

The authors would like to thank Frank Pfenning and Tom Murphy VII for their helpful correspondence on related work.

This work was partially supported by the National Science Foundation under grant number CCF-1629444.

REFERENCES

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *PPoPP '13*.

Arvind and K. P. Gostelow. 1978. *The Id Report: An Asychronous Language and Computing Machine.* Technical Report TR-114. Department of Information and Computer Science, University of California, Irvine.

Özalp Babaoğlu, Keith Marzullo, and Fred B. Schneider. 1993. A Formalization of Priority Inversion. *Real-Time Systems* 5, 4 (1993), 285–303.

Guy Blelloch and John Greiner. 1995. Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA '95).* ACM, 226–237.

Guy E. Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming.* ACM, 213–225.

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.* 21, 1 (1994), 4–14.

Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206.

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data parallel Haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007.* 10–18.

Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. 2008. 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).* ACM, New York, NY, USA, 11–22.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. 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).* ACM, 519–538.

Dennis Cornhill and Lui Sha. 1987. Priority Inversion in Ada. *Ada Letters* VII, 7 (Nov. 1987), 30–32.

Rowan Davies. 1996. A Temporal-Logic Approach to Binding-Time Analysis. In *LICS.* 184–195.

Derek L. Eager, John Zahorjan, and Edward D. Lazowska. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing* 38, 3 (1989), 408–423.

Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. 2016. Automatically Splitting a Two-Stage Lambda Calculus. In *Proceedings of the 25 European Symposium on Programming, ESOP.* 255–281.

C. J. Fidge. 1993. A Formal Definition of Priority in CSP. *ACM Trans. Program. Lang. Syst.* 15, 4 (Sept. 1993), 681–705.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (2011), 1–40.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI.* 212–223.

Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7 (1985), 501–538.

Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. 1993. Using Threads in Interactive Systems: A Case Study. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93).* ACM, New York, NY, USA, 94–105.

Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. 2010. The Design Rationale for Multi-MLton. In *ML '10: Proceedings of the ACM SIGPLAN Workshop on ML.* ACM.

Limin Jia and David Walker. 2004. Modal Proofs as Distributed Programs. In *13th European Symposium on Programming, ESOP 2004,* David Schmidt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 219–233.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10).* 261–272.

Butler W. Lampson and David D. Redell. 1980. Experience with Processes and Monitors in Mesa. *Commun. ACM* 23, 2 (1980), 105–117.

G. Levine. 1988. The Control of Priority Inversion in Ada. *Ada Lett.* VIII, 6 (Nov. 1988), 53–56.

Jonathan Moody. 2003. *Modal Logic as a Basis for Distributed Computation.* Technical Report CMU-CS-03-194. School of Computer Science, Carnegie Mellon University.

Stefan K. Muller and Umut A. Acar. 2016. 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.* 71–82. https://doi.org/10.1145/2935764.2935793

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2017. Responsive Parallel Computation: Bridging Competitive and Cooperative Threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* ACM, New York, NY, USA, 677–692.

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. 2004. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS).* IEEE Press, 286–295.

Tom Murphy, VII. 2008. *Modal Types for Mobile Code.* Ph.D. Dissertation. Carnegie Mellon. Available as technical report CMU-CS-08-126.

Tom Murphy, VII, Karl Crary, and Robert Harper. 2007. Type-safe Distributed Programming with ML5. In *Trustworthy Global Computing 2007*.

Frank Pfenning and Rowan Davies. 2001. A Judgmental Reconstruction of Modal Logic. *Mathematical Structures in Computer Science* 11 (2001), 511–540.

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 392–406.

John H. Reppy. 1999. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA.

Mads Rosendahl. 1989. Automatic complexity analysis. In *FPCA '89: Functional Programming Languages and Computer Architecture*. ACM, 144–156.

David Sands. 1990. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*. Springer-Verlag, London, UK, 361–376.

David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Eric Harslem. 1982. Designing the Star User Interface. *BYTE Magazine* 7, 4 (1982), 242–282.

Daniel Spoonhower. 2009. *Scheduling Deterministic Parallel Programs*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA.

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2008. Space Profiling for Parallel Functional Programs. In *International Conference on Functional Programming*.

Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. 1986. A Structural View of the Cedar Programming Environment. *ACM Trans. Program. Lang. Syst.* 8, 4 (Aug. 1986), 419–490.

J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384 – 393.

Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. 1998. Titanium: A high-performance Java dialect. In *In ACM 1998 Workshop on Java for High-Performance Network Computing (New*. ACM, Ed., ACM Press.