Type-Safe Multithreaded Programming with Priorities

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

Many multithreaded programs are naturally interactive and use priorities to ensure responsiveness. Programming with priorities, however, is challenging because existing techniques using a fixed set of priorities do not suffice to express rich priority relations and the lack of language support for the correct usage of priorities can lead to delays due to priority inversion. In this paper, we take an important step toward providing language support for multithreaded programming with priorities. We consider a language that provides the classic spawn-sync (a.k.a., fork-join) interface for threads and that allows the programmer to define priorities and their relationship by defining a partial order relation between them. We then equip the language with a modal type system analogous to S4 modal logic, where the accessibility relation between worlds is assumed to be symmetric and transitive. The type system ensures that priorities are used correctly without priority inversions, thus precluding a whole class of important run-time problems statically. We show that our proposed techniques are realistic by extending the Standard ML language with the necessary primitives to support multithreading and priorities and providing a prototype compiler for the extended language.

1 INTRODUCTION

Interactive applications, which interact with the user in some way, constitute an important and widely-used class of programs. Interactive applications typically aim to complete latency-sensitive computations as quickly as possible by maximizing responsiveness or minimizing latency. Latency-sensitive computations usually consist of those that have an effect observable from the outside, e.g., communication with a user or another piece of software through a network.

To ensure responsiveness, interactive programs rely on competitive threading: the programmer creates threads to perform certain computations and assigns high priority to latency-sensitive computations (e.g., [16, 21, 33]). The threading system then schedules the threads with the goal of maximizing the responsiveness (minimizing the latency) of high-priority threads, using preemption to suspend lower-priority threads as necessary. For example, in an email client that simultaneously performs several tasks of varying importance, there could be several different levels of priorities: the main event loop that drives user interactions can be given medium priority, asynchronously sending an email could be given low priority, and displaying an alert if the sending fails could be given high priority.

Although priorities appear conceptually simple, programming with them can be challenging, especially with regard to modularity and correctness. Existing languages typically define priorities as a totally ordered set with a fixed range. This has the advantage of capturing the intuitive notion of priority and is straightforward to implement, but it is not modular. For example, imagine coding an email client in a system with priority levels labeled LOW, MED and HI. To send an email, we may want to use a function that is provided by an external library, which assigns the function

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnn

Suppose that the same library uses priority HI to display an alert if the send operation fails. But now, there is no priority to assign to the main event (interaction) loop, whose priority should be higher than the send operation but lower than an alert. For the same reason, there is no way to define an alert with even higher priority than the “Failed to Send” alert, e.g., a security alert.

One way to solve this modularity problem could be to increase the range of priorities to support more than just the three we have. Early pioneering work on Cedar uses 7 levels [21], Java uses 10, and the POSIX threads (pthreads) API exposes several scheduling policies with as many as 100 levels. But then two important questions arise. First, how many levels should there be? With any fixed level of priorities, we can always imagine an application that needs more. For example, consider an application that uses two libraries with \( n \) and \( m \) priorities respectively. If the application wishes to give priority to all functions of the first library over the other, then we would need a total of \( n + m \) priorities. The second problem is that, as the range increases, reasoning with priorities becomes difficult: how should the programmer reason about the difference between priorities 6 and 7, or between 41 and 42? In fact, prior work shows that programmers had trouble using just the 7 priorities of Cedar [21]. To overcome these difficulties, several researchers have argued that priorities should be partially ordered for several reasons, including intuitive use and expressiveness [4, 15]. Partially ordered priorities are rarely supported. We only know of the occam language, which provides the programmer with primitives to prioritize certain parallel branches. These primitives, however, provide insufficient expressive power, leaving the potential for ambiguities [15].

In addition to being challenging to use modularly, incorrect use of priorities can also lead unacceptable and even fatal delays by causing a high-priority thread to wait for low-priority ones, a problem called priority inversion (e.g., [4, 27]). Because they lead to serious problems, priority inversions should ideally be avoided. Without programming language support, however, this is challenging, because the programmer must reason about the effects of priorities globally, considering all the ways the threads in a concurrent system might interact. For example, “Mars Pathfinder”, the robotic craft that landed on Mars on 4 July 1997, suffered from a software bug that caused the craft to reset itself periodically soon after landing. The bug, which was found using a replica of the system and fixed remotely, was traced to a priority inversion. Although some researchers have discussed language support for preventing priority inversion in languages such as Ada [12, 30], we are aware of no static language mechanisms for preventing priority inversions. Instead, researchers and practitioners have proposed techniques for handling priority inversions at run time as they arise. These techniques include priority inheritance [27] and more sophisticated resource allocation strategies [4]. These techniques at least make an attempt to avoid undesirable outcomes but can complicate the analysis of a system. For example, priority inheritance can promote the priority of a thread if the thread is involved in a priority inversion. This prevents the inversion but at the cost of raising, unbeknownst to the programmer, the priority of a thread that was intended to be executed at a lower priority.

In this paper, we propose language-based techniques for writing high-level multithreaded code with priorities. Our techniques guarantee modularity and prevent priority inversions statically. To this end, we allow the programmer to define priorities and the ordering relation between them and to assign priorities to threads in ways that respect the priority orderings and that don’t cause priority inversions. To support expressiveness and flexibility, we allow priorities to be partially ordered, relieving thus the programmer from specifying a total order, which is counter-intuitive and unnecessary [4]. To support threading, we provide for first-class asynchronous threads, with two operations, \texttt{spawn} and \texttt{sync}. This classic thread model, while simple, is expressive and popular, allowing for example the implementation of a range of interactive and parallel applications.
We specify a type system that ensures that priorities are used consistently. 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. [34], and Jia and Walker [25] 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 tracks the interactions between threads (through synchronization) at run time. This information allows us to show, in addition to standard type safety results, that well-typed programs do not have priority inversions at runtime. 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.

This paper shows that a reasonably broad class of multithreaded programs can safely use priorities. Our specific contributions include the following.

- A calculus $\lambda^4$, consisting of a type system and dynamic semantics that supports expressive and safe primitives for multithreading programming with priorities.
- Proofs of type safety and freedom from priority inversions.
- An extension of the Parallel ML language, called PriML, with language constructs for expressing priorities and the partial ordering between them.
- An elaboration translation from PriML to $\lambda^4$ showing that the calculus preserves the expressivity of the language.
- An implementation of the compiler and the runtime system for PriML as an extension of the Parallel MLton compiler.
- Example programs, such as a simple web server, written using our implementation, which give preliminary qualitative justification that our proposals are practical.

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 (as described briefly in the introduction), 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) and describes how they may be realized in practice (Section 5).

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
```
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 that we expect 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 safe use of priorities, PriML provides a modal type system that tracks priorities. The types of PriML include the standard types of functional programming including sums, products, function types, 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 \( \pi \) is a newly bound priority variable, and \( C \) is a set of constraints of the form \( p_1 \leq p_2 \) (where \( p_1 \) and \( p_2 \) are priority constants or variables, one of which will in general be \( \pi \)), which bounds the allowable instantiations of \( \pi \).

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 run asynchronously with the rest of the program.

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

The `spawn` command takes a command to run in the new thread and returns a handle to the spawned thread. In the above code, this handle is ignored, but it can also be bound to a variable and used later to synchronize with the thread (wait for it to complete).

```ml
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. Since sorting is a basic function and may be used at many priorities, we
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.

would want the code for qsort to be polymorphic over priorities, allowing it to be executed at all suitable 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, then it partitions the sequence by using a pivot, chosen to be middle element of the sequence. It then returns a command that sorts in parallel the sub-sequences consisting of the elements that are less than and greater than the pivot, and concatenates the sorted sequences to produce the result. To perform the two recursive calls in parallel, the function spawns two threads, using the command spawn, and specifies that the threads operate at priority p.

**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. Since syncing with that

\(^1\)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.
thread immediately afterward (line 13) causes the remainder of the event loop (high-priority) to
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.

While 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” is a common idiom in programming with futures, but this attempt has gone awry because the thread at priority display_p is waiting on the lower-priority sorting thread. Because of priority polymorphism, it may not be immediately clear where exactly the priority inversion occurs, and yet this code will still be correctly rejected by the type system. The type error is on line 10. This sync operation is passed a thread of priority p (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 20 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 20 would become ill-typed, as it should since 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).
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. fun[p] disp (t : email seq thread[p]) : unit cmd[display_p] =
   cmd[display_p] {
     l <- sync t;
     ret (display_ordered l)
   }
7. fun loop emails : unit cmd[loop_p] =
   case next_event () of
     SORT_BY_DATE =>
       cmd[loop_p] {
         t <- spawn[sort_p] { do ([sort_p]qsort date emails));
         spawn[display_p] { do ([sort_p]disp t) }
       }
   | ...

Fig. 3. An ill-typed attempt at chaining threads together. The high-priority display thread is waiting on the lower-priority sorting thread.

\[
\begin{align*}
\text{Types} & \quad \tau ::= \text{unit} \mid \tau \text{thread}[p] \mid \tau \text{cmd}[p] \mid \text{nat} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \mid \forall \pi : C.\tau \\
\text{Priorities} & \quad p ::= \bar{p} \mid \pi \\
\text{Constrs.} & \quad C ::= p \leq p \mid C \land C \\
\text{Exprs.} & \quad e ::= x \mid \langle \rangle \mid \text{tid}[a] \mid \overline{n} \mid \text{ifz} \ e \{e;x,e\} \mid \lambda x.e \mid e \ e \mid \langle e,e \rangle \\
& \quad \quad \quad \mid \text{fst} \ e \mid \text{snd} \ e \mid \text{inl} \ e \mid \text{inr} \ e \mid \text{case} \ e \{x.e;y.e\} \\
& \quad \quad \quad \mid \text{output} \ e \mid \text{input} \ \text{cmd}[p] \{m\} \mid \Lambda \pi : C.e \mid e[p] \mid \text{fix} \ x: \tau \ \text{is} \ e \\
\text{Commands} & \quad m ::= x \leftarrow e; m \mid \text{spawn}[p;\tau] \{m\} \mid \text{sync} \ e \mid \text{ret} \ e
\end{align*}
\]

Fig. 4. Syntax of \( \lambda^4 \)

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

3 THE \( \lambda^4 \) CALCULUS

In this section, we define a core calculus \( \lambda^4 \) 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 \( \lambda^4 \). In addition to the unit type, a type of natural numbers, functions, product types and sum types, \( \lambda^4 \) has three special types. The type \( \tau \text{ thread}[p] \) is used for a handle to an asynchronous thread running at priority \( p \) and returning a value of type \( \tau \). The type \( \tau \text{ cmd}[p] \) 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 \( \Lambda \pi : \pi \leq \overline{p}. e \) can only be instantiated with priorities \( \overline{p'} \) for which \( \overline{p'} \leq \overline{p} \).

A priority \( \overline{p} \) can be either a priority constant, written \( \overline{p} \), or a priority variable \( \pi \). 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 \( \lambda^4 \) and PriML, in which the program can define new priority constants. We will discuss in Section 4 how the priority definitions of PriML may be hoisted out of the program to produce \( \lambda^4 \) programs.

Expressions and commands are distinguished syntactically. Expression variables (which will have pure values substituted for them at runtime) are expressions, as are unit values \( \langle \rangle \). Numerals \( \overline{n} \) are the runtime representation of the natural number \( n \). They are eliminated with the if-zero conditional \( \text{ifz } e \{ e_1; x.e_2 \} \). Expressions also include anonymous functions \( \lambda x.e \), function application, pairs, projection, injection, and the case construct case \( e \{ x.e_1; y.e_2 \} \). Finally, we include encapsulated commands \( \text{cmd}[p] \{ m \} \), priority-level abstractions \( \Lambda \pi : C.e \) and priority instantiation \( e[p] \).

As in PriML, the language constructs that involve priorities are factored out into the command layer. 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 command \( \text{spawn}[p; \tau] \{ m \} \) is parametrized by both a priority \( p \) and the type \( \tau \) of the return value of \( m \) for convenience in defining the dynamic semantics.

### 3.1 Static Semantics

The type system of \( \lambda^4 \) carefully tracks the priorities of threads as they wait for each other and enforces that a program is free of priority inversions. Even if such a strong static guarantee is not desired, tracking the priorities of threads statically gives programmers the ability to reason about priority inversions and where they might occur.

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 \( \Sigma \) containing the types and priorities of running threads. A signature has entries of the form \( a \sim \tau @ p \) indicating that thread \( a \) is running at priority \( p \) and will return a value of type \( \tau \). The signature is needed to check the types of thread handles.

The expression typing judgment is \( \Gamma \vdash^P e : \tau \), indicating that under signature \( \Sigma \), a partial order \( P \) of priority constants and context \( \Gamma \), expression \( e \) has type \( \tau \). As usual, the variable context \( \Gamma \) maps variables to their types. The rules for this judgment are shown in Figure 5. The variable rule \( \text{VAR} \), 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 \( \text{tid} [a] \) looks up the thread \( a \) in the signature. The rule for encapsulated commands \( \text{cmd}[p] \{ m \} \) requires that the command \( m \) be well-typed and runnable at priority \( p \), using the typing judgment for commands, which will be defined below. Rule \( \forall I \) extends the context with both the priority variable \( \pi \) and the constraint \( C \).
\[
\begin{array}{lcl}
\text{VAR} & \quad & \text{unitl} \\
\Gamma, x : \tau & \vdash^P & x : \tau \\
\text{natl} & \quad & \text{TID} \\
\Gamma \vdash^P \text{pid} : \text{nat} & \quad & \Gamma \vdash^P \text{pid} \, \text{ifz e} \langle e_1, e_2 \rangle : \tau \\
\Gamma \vdash^P \text{pid} \, \text{e} : \tau & \quad & \Gamma \vdash^P \text{pid} \, \text{e} \langle e_1, e_2 \rangle : \tau \\
\Gamma \vdash^P \text{pid} \, \text{e} : \tau & \quad & \Gamma \vdash^P \text{pid} \, \text{e} : \tau \\
\end{array}
\]

\[
\begin{array}{lcl}
\text{IF} & \quad & \text{cmdl} \\
\Gamma \vdash^P \text{pid} \, \text{e} : \tau & \quad & \Gamma \vdash^P \text{pid} \, \text{e} \langle e_1, e_2 \rangle : \tau \\
\Gamma \vdash^P \text{pid} \, \text{e} : \tau & \quad & \Gamma \vdash^P \text{pid} \, \text{e} : \tau \\
\Gamma \vdash^P \text{pid} \, \text{e} : \tau & \quad & \Gamma \vdash^P \text{pid} \, \text{e} : \tau \\
\end{array}
\]

\[
\begin{array}{lcl}
\text{BIND} & \quad & \text{SPAWN} \\
\Gamma \vdash^P \text{e} : \tau & \quad & \Gamma \vdash^P \text{e} : \tau \\
\Gamma, x : \tau \vdash^P m \sim \tau' @ p & \quad & \Gamma, x : \tau \vdash^P m \sim \tau' @ p \\
\Gamma \vdash^P x \leftarrow e; m \sim \tau' @ p & \quad & \Gamma \vdash^P \text{spawn}[p'; \tau] \{ m \} \sim \tau \text{ thread}[p'] @ p \\
\end{array}
\]

\[
\begin{array}{lcl}
\text{SYNC} & \quad & \text{RETR} \\
\Gamma \vdash^P \text{e} : \tau \text{ thread}[p'] & \quad & \Gamma \vdash^P \text{e} : \tau \\
\Gamma \vdash^P m \sim \tau @ p & \quad & \Gamma \vdash^P \text{ret e} \sim \tau @ p \\
\end{array}
\]

\begin{figure}
\caption{Expression typing rules.}
\end{figure}

\begin{figure}
\caption{Thread typing rules.}
\end{figure}
Rule $\forall E$ handles priority instantiation. When instantiating the variable $\pi$ with priority $p'$, the rule requires that the constraints hold with $p'$ substituted for $\pi$ (the constraint typing judgment $\Gamma \vdash^P C$ will be discussed below). The rule also performs the corresponding substitution in the return type.

The command typing judgment is $\Gamma \vdash^P m \sim \tau \circ @ p$ and includes both the return type $\tau$ and the priority $p$ 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 $\tau$, and then extends the context with a variable $x$ of type $\tau$ in order to type the remaining command. The rule for $\text{spawn}[p';\tau'] \{m\}$ requires that $m$ be runnable at priority $p'$ and return a value of type $\tau'$. The spawn command returns a thread handle of type $\tau' \text{thread}[p']$, and may do so at any priority. The $\text{sync} e$ command requires that $e$ have the type of a thread handle of type $\tau$, and returns a value of type $\tau$. 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 $\text{sync}$ commands that would cause priority inversions. Finally, if $e$ has type $\tau$, then the command $\text{ret} e$ returns a value of type $\tau$, 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 $\text{HYP}$) or the partial order (rule $\text{ASSUME}$) or if it can be concluded from reflexivity or transitivity (rules $\text{REFL}$ and $\text{TRANS}$, respectively). Finally, the conjunction $C_1 \land C_2$ 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 ($[p/[\pi]]e$), commands ($[p/[\pi]]m$), constraints ($[p/[\pi]]C$), contexts ($[p/[\pi]]T$), 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 $\Gamma, x : \tau \vdash^p \begin{array}{c} \text{e}_1 : \tau' \text{ and } \Gamma \vdash^p \begin{array}{c} \text{e}_2 : \tau \end{array} \end{array}$, then $\Gamma \vdash^p \begin{array}{c} \text{[e}_2/\text{x}]\text{e}_1 : \tau' \end{array}$.
2. If $\Gamma, x : \tau \vdash^p \begin{array}{c} m \sim \tau' \circ @ p \text{ and } \Gamma \vdash^p \begin{array}{c} \text{e} : \tau \end{array} \text{, then } \Gamma \vdash^p \begin{array}{c} \text{[e}/\text{x} \text{]m} \sim \tau' \circ @ p \end{array}$.
3. If $\Gamma, \pi \text{ pri o} \vdash^p \begin{array}{c} \text{e} : \tau, \text{ then } \begin{array}{c} \text{[p}/\text{[\pi]} \text{]} \Gamma \vdash^p \begin{array}{c} \text{[p}/\text{[\pi]} \text{]} \text{[e}/\text{x} \text{] \sim [p}/\text{[\pi]} \text{]} \end{array} \end{array} \text{.}$
4. If $\Gamma, \pi \text{ pri o} \vdash^p \begin{array}{c} m \sim \tau \circ @ p, \text{ then } \begin{array}{c} \begin{array}{c} \text{[p}/\text{[\pi]} \text{]} \Gamma \vdash^p \begin{array}{c} \text{[p}/\text{[\pi]} \text{]} \text{[p}/\text{[\pi]} \text{]} \text{[m} \sim \tau' \circ @ p \text{]} \text{.} \end{array} \end{array} \end{array}$
5. If $\Gamma, \pi \text{ pri o} \vdash^p C, \text{ then } \begin{array}{c} \text{[p}/\text{[\pi]} \text{]} \Gamma \vdash^p \begin{array}{c} \text{[p}/\text{[\pi]} \text{]} \text{.} \end{array} \end{array}$

**Proof.** (1) By induction on the derivation of $\Gamma, x : \tau \vdash^p \begin{array}{c} \text{e}_1 : \tau' \end{array}$.

- $\forall E$. Then $e = e_0[p']$. By inversion, $\Gamma, x : \tau \vdash^p \begin{array}{c} e_0 : \forall \pi : C. \tau_0 \text{ and } \tau' = [p' \circ \pi] \tau_0. \text{ By induction, } \Gamma \vdash^p \begin{array}{c} \text{[e}_2/\text{x} \text{]e}_0 : \forall \pi : C. \tau_0. \text{ Apply } \forall E.$

(2) By induction on the derivation of $\Gamma, x : \tau \vdash^p \begin{array}{c} m \sim \tau' \circ @ p \end{array}$.

- $\text{BIND}$. Then $m = y \leftarrow e_0, m_0$. By inversion, $\Gamma, x : \tau \vdash^p \begin{array}{c} e_0 : \tau'' \circ \text{cmd}[p] \text{ and } \Gamma, x : \tau, y : \tau'' \vdash^p \begin{array}{c} m_0 \sim \tau' \circ @ p \end{array}. \text{ By weakening, } \Gamma, y : \tau'' \vdash^p \begin{array}{c} e : \tau. \text{ By induction, } \Gamma \vdash^p \begin{array}{c} \text{[e}/\text{x} \text{]e}_0 : \tau''. \text{ cmd}[p] \text{ and } \Gamma, y : \tau'' \vdash^p \begin{array}{c} \text{[e}/\text{x} \text{]m}_0 \sim \tau' \circ @ p \end{array}. \text{ Apply } \text{BIND.}$
We will also define a syntax and dynamic semantics for 

We define a small-step operational semantics for

\(\lambda\)

Type-Safe Multithreaded Programming with Priorities 1:11

contained command) and priority abstractions.

formed. For clarity, we will often use metavariables

and variants to range over values, even though they are in the same syntactic class as expressions. The values are the unit value, thread handles contained in \(\Sigma\), numerals, functions, pairs of values, injections of values, encapsulated commands (since the command is encapsulated and not run, these are values regardless of the contained command) and priority abstractions.

3.2 Dynamic Semantics

We define a small-step operational semantics for \(\lambda^4\). Since expressions are pure and do not depend on priorities or other running threads, the semantics for expressions need not take these into account. The semantics for commands will be more complex, since it must include other threads. We will also define a syntax and dynamic semantics for thread pools, which are collections of all of the currently running threads.

The dynamic semantics for expressions, shown in Figure 10, consists of two judgments. The judgment \(e \mathit{val}_\Sigma\) states that \(e\) is a value under the signature \(\Sigma\), that is, it is irreducible and well-formed. For clarity, we will often use metavariables \(v\) and variants to range over values, even though they are in the same syntactic class as expressions. The values are the unit value, thread handles contained in \(\Sigma\), numerals, functions, pairs of values, injections of values, encapsulated commands (since the command is encapsulated and not run, these are values regardless of the contained command) and priority abstractions.

The transition relation for expressions is fairly straightforward for a left-to-right, call-by-value lambda calculus. For space reasons, we include only selected rules as examples. The judgment, $e \rightarrow_{\Sigma} e'$, indicates that $e$ steps to $e'$ under the signature $\Sigma$, which does not change during expression evaluation and is used solely to determine whether thread handles are well-formed values. The ifz and case constructs evaluate the argument $e$ to a value in order to examine it. The ifz construct conditions on the value of the numeral $n$ to which $e$ evaluates. 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 $e$ evaluates to a left or right injection, and steps to $e_1$ (resp. $e_2$), substituting the injected value for $x$ (resp. $y$). Since the semantics is call-by-name, function applications $e_1 e_2$ are not performed until $e_1$ is evaluated to a lambda abstraction and $e_2$ is evaluated to a value, which is then substituted for the bound variable. Pairs are evaluated left-to-right, with the first component being fully evaluated before the second, and both are evaluated eagerly before a projection is performed. Priority instantiation $e[p']$ evaluates $e$ to an abstraction, and then performs the substitution.

Define a thread pool $\mu$ to be a mapping of thread symbols to threads: $a \leftrightarrow m$ indicates a thread $a$ running $m$. The concatenation of two thread pools is written $\mu_1 \oplus \mu_2$. Thread pools can also introduce new thread names: the thread pool $\nu \Sigma \{\mu\}$ allows the thread pool $\mu$ to use thread names bound in the signature $\Sigma$. Thread pools are not ordered; we identify thread pools up to commutativity and


associativity of $\cup^2$. We also introduce the additional congruence rules of Figure 9, which allow for thread name bindings to freely change scope within a thread pool. We will use this congruence relation to establish a normal form for thread pools, in which all thread names have been moved to the outside: for every thread pool $\mu$, there exists a thread pool $v\Sigma\{\mu'\} \equiv \mu$, where $\mu'$ has no nested thread name bindings.

Figure 8 gives the typing rules for thread pools. The typing judgment is $\tau\Sigma\mu$ and the rules are straightforward: the empty thread pool $\emptyset$ is always well-typed, individual threads are well-typed if their commands are, and concatenations are well-typed if their components are. Bindings $v\Sigma'\{\mu\}$ introduce the additional bindings in $\Sigma'$ in checking whether $\mu$ is well-typed.

The transition judgment for commands is $m \rightarrow^\alpha \Sigma (\Sigma', m', \mu')$, indicating that under signature $\Sigma$, command $m$ steps to $m'$. The transition relation carries a label $\alpha$, indicating the “action” taken by this step. At this point, actions can be the silent action $\epsilon$ or the sync action $b ? v$, indicating

---

Fig. 11. Dynamic rules for commands.

![Diagram showing the rules for commands](image)

Fig. 12. Dynamic rules for thread pools.

---

Since 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 $\lambda^2$. This is outside the scope of this paper.
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 (\( \mu' \)) and the signature (\( \Sigma' \)). 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 \( \text{cmd}[p] \{ m_1 \} \), then evaluate this command to a return value \( \text{ret} v \) before substituting \( v \) for \( x \) in \( m_2 \). The spawn command \( \text{spawn}[p; \tau] \{ m \} \) does \textit{not} evaluate \( m \), but simply spawns a fresh thread \( b \) to execute it, and returns a thread handle \( \text{tid}[b] \).

The sync command \( \text{sync} e \) evaluates \( e \) to a thread handle \( \text{tid}[b] \), and then takes a step to \( \text{ret} v \) labeled with the action \( b \ ? \ v \). Note that, since 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, \( \text{ret} e \) evaluates \( e \) to a value.

We define an additional transition judgment for thread pools, which nondeterministically allows a thread to step (without regard to priority, since the concern of the semantics is safety, not responsiveness; a prioritized implementation is discussed in Section 5). Thus, the semantics can produce all possible interleavings of threads. Our safety theorems will then be quantified over all such interleavings. The judgment \( \mu \xrightarrow{\Sigma} \mu' \) is again labeled with an action, which now also includes the \textit{return} action \( b \ ! v \), indicating that thread \( b \) returns \( v \). The third rule matches these two actions and performs the synchronization. If a thread in \( \mu_1 \) wishes to sync with \( b \) and a thread \( b \) in \( \mu_2 \) wishes to return its value, then the thread pool \( \mu_1 \uplus \mu_2 \) can step silently, performing the synchronization. Without loss of generality, \( \mu_1 \) can come first since thread pools are identified up to ordering. The last two rules allow threads to step when concatenated with other threads and under bindings.

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, since 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, \textit{every} thread is either complete or 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.

\[ \vdash \Sigma \ e \ \text{action} \]

\[ \vdash \Sigma, b \rightarrow p \ b ? \ v \ \text{action} \]

\[ \vdash \Sigma, b \rightarrow p \ b ! \ v \ \text{action} \]

\[ \text{Fig. 13. Static semantics for actions.} \]
Proof. (1) By induction on the derivation of $\cdot \vdash^P e : \tau$. The cases are standard.

(2) By induction on the derivation of $\cdot \vdash^P m \sim \tau @ p$.
- **Bind.** Then $m = x \leftarrow e; m_2$. By inversion, $\cdot \vdash^P e : \tau' \cmd[p]$ and $x : \tau \vdash^P m_2 \sim \tau @ p$. By induction, either $e \val^\Sigma$ or $e \rightarrow e'$. In the second case, $m$ steps by rule **Bind1.** In the first case, by canonical forms, $e = \cmd[p] \{m_1\}$ and, by inversion on the expression typing rules, $\cdot \vdash^P m_1 \sim \tau' @ p$. By induction, either $m_1 = \ret \ e$ where $e \val^\Sigma$ or $m_1$ takes a step. In both cases, $m$ takes a step (**Bind3** or **Bind2**).
- **Spawn.** Apply rule **Spawn**.
- **Sync.** Then $m = \sync e$. By inversion, $\cdot \vdash^P e : \tau \thread[p']$. By induction, either $e \val^\Sigma$ or $e \rightarrow e'$. In the second case, $m$ steps by rule **Sync1.** In the first case, by canonical forms, $e = \tid[b]$. Apply rule **Sync2**.
- **Ret.** Then $m = \ret e$. By inversion, $\cdot \vdash^P e : \tau$. By induction, either $e \val^\Sigma$ or $e \rightarrow e'$. In the second case, $m$ steps by rule **Ret.** In the first case, the conclusions are trivially satisfied.

(3) By induction on the derivation of $\vdash^P \alpha \mu$. We consider the interesting cases.
- **Concat.** By inversion, $\cdot \vdash^P \mu_1$ and $\cdot \vdash^P \mu_2$. By induction, $\mu_1 \equiv v^\Sigma_1 \{\mu'_1\}$ and $\mu_2 \equiv v^\Sigma_2 \{\mu'_2\}$, where $\mu'_1$ and $\mu'_2$ meet the conditions of the theorem. We have $\mu_1 \uplus \mu_2 \equiv v^\Sigma_1, \Sigma_2 \{\mu'_1 \uplus \mu'_2\}$, and $\mu'_1 \uplus \mu'_2$ meets the conditions.
- **Extend.** Then $\mu = v^\Sigma \{\mu'\}$ and by inversion, $\cdot \vdash^P \Sigma' \{\mu''\}$. By induction, $\mu'' \equiv v^\Sigma'' \{\mu'''\}$ and $\mu'''$ meets the conditions of the theorem. We have $\mu \equiv v^\Sigma, \Sigma'' \{\mu''\}$.

\[ \Box \]

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

Theorem 2 (Preservation).

1. If $\cdot \vdash^P e : \tau$ and $e \rightarrow e'$, then $\cdot \vdash^P e' : \tau$.
2. If $\cdot \vdash^P m \sim \tau @ p$ and $m \alpha \rightarrow^\Sigma (\Sigma', m', \mu')$ and $\vdash^P \alpha$ action then $\cdot \vdash^P m' \sim \tau @ p$ and $\vdash^P \Sigma', \mu'$.
3. If $\vdash^P \mu$ and $\mu \alpha \mu'$ then $\vdash^P \mu'$

Proof. (1) By induction on the derivation of $\cdot \vdash^P e$. The cases are standard.

(2) By induction on the derivation of $m \alpha \rightarrow^\Sigma (\Sigma', m', \mu')$.
- **Bind1.** By inversion on the typing rules, $\cdot \vdash^P e : \tau' \cmd[p']$ and $x : \tau' \vdash^P m \sim \tau @ p$. By induction, $\cdot \vdash^P e' : \tau' \thread[p']$. Apply rule **Bind**.
- **Bind2.** By inversion on the typing rules, $\cdot \vdash^P \cmd[p] \{m_1\} : \tau' \cmd[p]$ and $\cdot \vdash^P m_1 \sim \tau' @ p$ and $x : \tau' \vdash^P m_2 \sim \tau @ p$. By induction, $\cdot \vdash^P \cmd[p], m_1 \sim \tau' @ p$ and $\vdash^P \Sigma, \mu'$. By **Cmd1.** By inversion on the action typing rules, $b \sim \tau @ p' \in \Sigma$ and $\cdot \vdash^P \Sigma, b \sim \tau @ p'$. Apply rules **TID** and **Ret**.
- **Sync1.** By inversion on the typing rules, $\cdot \vdash^P e : \tau \thread[p']$ and $p \leq p'$. By induction, $\cdot \vdash^P e' : \tau \thread[p']$. Apply rule **Sync**.
- **Sync2.** By inversion on the typing rules, $\cdot \vdash^P \tid[b] : \tau \thread[p']$ and $p \leq p'$. By inversion on the action typing rules, $b \sim \tau @ p' \in \Sigma$ and $\cdot \vdash^P \Sigma, v : \tau$. Apply rule **Ret**.

(3) By induction on the derivation of $\mu \alpha \mu'$. We show the interesting case.
• **THREADT.** By induction on the typing rules, \( \vdash^P \alpha \rightarrow \tau \rightarrow p, \Sigma \) \( m \sim \tau \rightarrow p \) and induction, \( \vdash^P \alpha \rightarrow \tau \rightarrow p, \Sigma, \Sigma' \) \( m' \sim \tau' \rightarrow p \) and \( \vdash^P \alpha \rightarrow \tau \rightarrow p, \Sigma, \Sigma' \) \( \mu' \). Apply rules ONE_THREAD, CONCAT and EXTEND.

In order to show the absence of priority inversions, we also show that a well-typed command can only take a step labeled with a sync action if the sync would not cause a priority inversion.

**Lemma 2.** If \( \vdash^P \alpha \rightarrow \tau \rightarrow p, \Sigma, a \rightarrow m \xrightarrow{b ? \nu} \Sigma, a \rightarrow \mu' \rightarrow p \) and \( a \rightarrow m \xrightarrow{b ? \nu} \Sigma, a \rightarrow \mu' \rightarrow p \), then \( b \sim \tau' \rightarrow p' \in \Sigma \) and \( p \leq p' \).

**Proof.** By induction on the derivation of \( a \rightarrow m \xrightarrow{b ? \nu} \Sigma, a \rightarrow \mu' \rightarrow p \). The only thread pool rule that applies is THREADT, so continue by induction on the sub-derivation \( m \xrightarrow{\alpha} \Sigma, \alpha \rightarrow \tau \rightarrow p, \Sigma, \Sigma' \) \( \mu' \). The only applicable non-inductive case is Sync2. In this case, \( m = \text{sync tid[b]} \). By induction on the typing rules, we have \( b \sim \tau' \rightarrow p' \in \Sigma \) and \( p \leq p' \).

We can now combine the results of this section into a theorem that shows both standard type safety and the additional result that no sync step causes a priority inversion.

**Theorem 3 (Type Safety and Priority Inversions).** If \( \vdash^P \alpha \rightarrow \tau \rightarrow p, \Sigma, \nu \Sigma, a \rightarrow m \xrightarrow{\Sigma} \mu' \rightarrow \Sigma \), then \( \mu' \equiv \nu \Sigma' \{ a_i \leftarrow m_1; \vdots ; a_n \rightarrow m_n \} \) where for all \( i \in [1, n] \),

1. Either \( m_i = \text{ret e} \) where \( e \equiv \text{val}_\Sigma \) or \( m_i \xrightarrow{\alpha} \Sigma, \Sigma' \rightarrow (\Sigma', m_i', \mu_i) \) and \( \vdash \Sigma, \Sigma' \rightarrow \alpha \) action.
2. For some \( \tau_i \) and \( p_i \), we have \( a_i \sim \tau_i \rightarrow p_i \in \Sigma, \Sigma' \).
3. If \( a_i \leftarrow m_i \xrightarrow{b ? \nu} \Sigma, \Sigma' \rightarrow \mu_i' \) then \( b \sim \tau' \rightarrow p' \in \Sigma, \Sigma' \) and \( p_i \leq p' \).

**Proof.** By inductive application of Theorem 1 and Theorem 2. Part (3) follows from Lemma 2.

To show, as the culmination of this section, that executions of well-typed programs are free of priority inversions, we use the formalization of priority inversions provided by Babaoglu et al. [4]. They define a composite system graph consisting of waits-for edges \((a, b)\) for all threads \(a, b\) such that \(a\) waits for \(b\) and priority edges \((a, b)\) where \(a < b\). In this formulation, a priority inversion corresponds to a \(\pi\)-cycle in the composite system graph, that is, a cycle consisting of exactly one priority edge. We can show by Theorem 3 and a simple argument that such a cycle cannot exist.

**Corollary 1.** If \( \vdash^P \alpha \rightarrow \tau \rightarrow p, \Sigma, \nu \Sigma, a \rightarrow m \xrightarrow{\Sigma} \mu' \rightarrow \Sigma \), then the execution of the program thus far has been free of priority inversions.

**Proof.** A priority inversion corresponds to a \(\pi\)-cycle \((a_1, a_2), (a_2, a_3), \ldots, (a_n, a_1)\), where without loss of generality, the edge \((a_n, a_1)\) is the priority edge. Each edge \((a_i, a_{i+1})\) corresponds to a transition \( a_i \leftarrow m_i \xrightarrow{a_{i+1} ? \nu} \Sigma_i' \rightarrow (\Sigma_i', m_i', \mu_i') \) at some point during the derivation, and by Theorem 3, we have \( a_i \sim \tau_i \rightarrow p_i, a_{i+1} \sim \tau_{i+1} \rightarrow p_{i+1} \in \Sigma' \) and \( p_i \leq p_{i+1} \), so \( p_1 \leq p_2 \leq \ldots \leq p_n \). But the presence of the priority edge \((a_n, a_1)\) means that \( p_n < p_1 \), which, by transitivity, gives \( p_1 < p_1 \), which is a contradiction.

### 4 ELABORATION OF PRIML TO \(\lambda^4\)

In this section, we define an elaboration from PriML to \(\lambda^4\). This can be seen as giving a formal semantics to PriML programs by placing them in correspondence with \(\lambda^4\) programs. We begin by formally defining the syntax of PriML, in Figure 14. We use the metavariables \(\hat{e}\) and \(\hat{m}\) (and variants) to refer to PriML expressions and commands, respectively, in order to distinguish them.
from their $\lambda^4$ counterparts. We also add syntactic classes for instructions $i$, priority annotations $\rho$, declarations $d$, toplevel declarations $o$ and programs $G$. Expressions are more or less the same as in $\lambda^4$, with the addition of let bindings, which introduce zero or more declarations into the scope of an expression (the notation $\hat{d}$ stands for a list of declarations). The command layer of PriML is split into two syntactic classes: commands and instructions, where a command is a sequence of instructions, each binding its return value, ending in a return. Instructions now include do$(\hat{e})$, which runs an encapsulated command (expressible in PriML as $x \leftarrow e; \text{ret } x$). Declarations $d$, which may be included in let bindings, introduce expressions, functions (which bind one or more variables, indicated by the notation $\hat{x}^\pi$), or priority-level functions (which bind one or more priority variables and one or more expression variables). All of these declarations can appear at the top level, as can priority and order declarations. These are the separate class of toplevel declarations. Finally, a program is zero or more toplevel declarations followed by a command to run as the “main” thread.

For simplicity of the formal definition of elaboration, a number of features of PriML are omitted from the formalism. For example, we include natural numbers and unit as the only base type, while PriML has integers, booleans, strings, etc. We also omit many useful features of a language such as algebraic datatypes, mutually recursive functions and exceptions. These are present in our implementation (although we have not implemented mutually recursive priority-level functions). The elaboration of these features is standard (e.g. [20]).

The main work of elaboration is converting declarations in PriML to (possibly recursive) functions (both expression- and priority-level) in $\lambda^4$, converting let bindings to function application, and hoisting priority and order declarations out of the code so that they may be presented as a pre-defined partially-ordered set, as required by the $\lambda^4$ semantics. The overall goal of elaboration then is to convert a PriML program $G$ into a $\lambda^4$ command $\hat{m}$ together with a partial order $P$ of priorities. This proceeds in a number of mutually recursive stages, with one elaboration judgment for each syntactic class of PriML.

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

The elaboration judgments are:

\[
\begin{align*}
\text{Expressions} \quad & \hat{e} ::= x | \langle \rangle | \pi | \text{idz } \hat{e} \{ \hat{e}; x.\hat{e} \} | \lambda x.\hat{e} | \hat{e} \{ \hat{e}, \hat{e} \} | \text{fst } \hat{e} | \text{snd } \hat{e} \\
& \quad | \text{inl } \hat{e} | \text{inr } \hat{e} | \text{case } \hat{e} \{ x.y.\hat{e} \} | \text{output } \hat{e} | \text{input } \\
\text{Instructions} \quad & i ::= \text{do}(\hat{e}) | \text{spawn}[p; r] \{ \hat{m} \} | \text{sync } \hat{e} | \text{ret } \hat{e} \\
\text{Commands} \quad & \hat{m} ::= x \leftarrow i; \hat{m} | i \\
\text{Prior. Annotations} \quad & \rho ::= \pi : C \\
\text{Declarations} \quad & d ::= \text{val } x = \hat{e} | \text{fun } f(\hat{x}^\pi) = \hat{e} | \text{fun } [\rho^\pi] f(\hat{x}^\pi) = \hat{e} \\
\text{Toplevel Decls.} \quad & o ::= d | \text{priority } \rho | \text{order } \rho < \rho^\pi \\
\text{Programs} \quad & G ::= \text{main } \{ \hat{m} \} | o G \\
\end{align*}
\]
\( \Gamma \vdash \hat{e} \leadsto e : \text{nat} \)

\( \Gamma \vdash \text{output } \hat{e} \leadsto e : \text{unit} \)

\( \Gamma \vdash \text{input } \hat{e} \leadsto e : \text{nat} \)

\( \Gamma \vdash \hat{m} \leadsto m \vdash \tau \mathrel{\triangleleft} p \)

\( \Gamma \vdash \text{cmd}[p] \{ \hat{m} \} \leadsto \tau \text{ cmd}[p] \{ m \} : \tau \text{ cmd}[p] \)

\( \Gamma, \pi \text{ prio}, C \vdash \hat{e} \leadsto e : \tau \)

\( \Gamma \vdash \lambda \pi \vdash C. \hat{e} \leadsto e \quad \Gamma \vdash \forall \pi : C. \tau \quad \Gamma \vdash [p'/\pi]C \)

\( \Gamma \vdash \hat{e} \leadsto e : \tau \quad \Gamma \vdash \text{do } (x, e') : \tau' \quad \Gamma, x : \tau' \vdash \text{let } d \text{ in } \hat{e} \text{ end } \leadsto e : \tau \)

\( \Gamma \vdash \text{let in } \hat{e} \text{ end } \leadsto e : \tau \quad \Gamma \vdash \text{let } d \text{ in } \hat{e} \text{ end } \leadsto (\lambda x. e) e' : \tau \)

\( \Gamma \vdash \hat{e} \leadsto e : \tau \text{ cmd}[p] \)

\( \Gamma \vdash \text{do } (\hat{e}) \leadsto_i x \leftarrow e ; \text{ret } x \vdash \tau \mathrel{\triangleleft} p \)

\( \Gamma \vdash \text{sync } \hat{e} \leadsto_i \text{ sync } e \vdash \tau \mathrel{\triangleleft} p \)

\( \Gamma \vdash \hat{m} \leadsto m \vdash \tau \mathrel{\triangleleft} p' \)

\( \Gamma \vdash \text{spawn}[p'; \tau] \{ \hat{m} \} \leadsto_i \text{ spawn}[p'; \tau] \{ m \} \vdash \tau \text{ thread}[p'] \mathrel{\triangleleft} p \)

\( \Gamma \vdash \hat{e} \leadsto e : \tau \)

\( \Gamma \vdash \text{ret } \hat{e} \leadsto_i \text{ ret } e \vdash \tau \mathrel{\triangleleft} p \)

\( \Gamma \vdash i \leadsto_i m \vdash \tau \mathrel{\triangleleft} p \)

\( \Gamma, x : \tau \vdash \hat{m}' \leadsto m' \vdash \tau' \mathrel{\triangleleft} p \)

\( \Gamma \vdash i \leadsto_i m \vdash \tau \mathrel{\triangleleft} p \)

\( \Gamma \vdash x \leftarrow i ; m' \leadsto m \leftarrow \text{cmd}[p] \{ m \} : m' \mathrel{\triangleleft} \tau' \mathrel{\triangleleft} p \)

\( \Gamma \vdash i \leadsto_i m \vdash \tau \mathrel{\triangleleft} p \)

\[
\begin{align*}
\text{Fig. 15. Elaboration of expressions(selected rules).} \\
\Gamma \vdash \hat{e} \leadsto e : \tau \text{ cmd}[p] & \quad \text{x fresh} & \quad \Gamma \vdash \hat{e} \leadsto e : \tau \text{ thread}[p'] & \quad \Gamma \vdash p \leq p' \\
\Gamma \vdash \text{do } (\hat{e}) \leadsto_i x \leftarrow e ; \text{ret } x \vdash \tau \mathrel{\triangleleft} p & \quad \Gamma \vdash \text{sync } \hat{e} \leadsto_i \text{ sync } e \vdash \tau \mathrel{\triangleleft} p \\
\Gamma \vdash \hat{m} \leadsto m \vdash \tau \mathrel{\triangleleft} p' & \quad \Gamma \vdash \text{spawn}[p'; \tau] \{ \hat{m} \} \leadsto_i \text{ spawn}[p'; \tau] \{ m \} \vdash \tau \text{ thread}[p'] \mathrel{\triangleleft} p \\
\Gamma \vdash \hat{e} \leadsto e : \tau & \quad \Gamma \vdash \text{ret } \hat{e} \leadsto_i \text{ ret } e \vdash \tau \mathrel{\triangleleft} p \\
\Gamma \vdash i \leadsto_i m \vdash \tau \mathrel{\triangleleft} p &\quad \Gamma, x : \tau \vdash \hat{m}' \leadsto m' \vdash \tau' \mathrel{\triangleleft} p \\
\Gamma \vdash i \leadsto_i m \vdash \tau \mathrel{\triangleleft} p & \quad \Gamma \vdash i \leadsto_i m \vdash \tau \mathrel{\triangleleft} p \\
\Gamma \vdash x \leftarrow i ; m' \leadsto m \leftarrow \text{cmd}[p] \{ m \} : m' \mathrel{\triangleleft} \tau' \mathrel{\triangleleft} p & \quad \Gamma \vdash i \leadsto_i m \vdash \tau \mathrel{\triangleleft} p \\
\end{align*}
\]

\[
\begin{align*}
\text{Fig. 16. Elaboration of instructions and threads.} \\
\text{• } \Gamma \vdash \hat{e} \leadsto e : \tau & \text{ converts a PriML expression to a } \lambda^4 \text{ expression.} \\
\text{• } \Gamma \vdash i \leadsto_i e \vdash \tau \mathrel{\triangleleft} p & \text{ converts a PriML instruction to a } \lambda^4 \text{ command. These will be sequenced} \\
\text{together using the bind operator of } \lambda^4. \\
\text{• } \Gamma \vdash \hat{m} \leadsto m \vdash \tau \mathrel{\triangleleft} p & \text{ converts a PriML command to a } \lambda^4 \text{ command.} \\
\text{• } \Gamma \vdash d \leadsto_d (x, e) : \tau & \text{ converts a PriML declaration to the variable it binds together with the } \lambda^4 \\
\text{expression bound to it.} \\
\text{• } P ; \Gamma \vdash G \leadsto_G m ; P' & \text{ converts a PriML program to a } \lambda^4 \text{ command, together with a partially} \\
\text{ordered set of priorities. The existing partial order } P \text{ is extended by } P'. \\
\text{• } \text{The elaboration rules for expressions are defined in Figure 15. These rules are mostly straightforward,} \\
\text{and closely follow the typing rules, so we show only a few rules as examples. The rules of} \\
\text{interest are those for let bindings. For each declaration in sequence, the declaration is elaborated} \\
\text{into a pair } (x, e'). \text{ The body of the let binding (together with the elaboration of the remaining} \\
\text{bindings) is wrapped in a } \lambda\text{-abstraction binding the variable } x, \text{ which is applied to } e'. \\
\text{The elaboration rules for instructions and commands are defined in Figure 16. The elaboration} \\
\text{rule for do instructions evaluates the expression, checks that it has the type of an encapsulated} \\
\text{command, and elaborates it to a command that uses the bind operation of } \lambda^4 \text{ to bind the command’s} \\
\end{align*}
\]
return value to a fresh variable. Other instructions elaborate to the corresponding command (with any sub-components recursively elaborated). This uniformly produces a command for each instruction, which can be sequenced together using binding in the first command elaboration rule. The second command elaboration rule, for \texttt{ret}, simply elaborates the final instruction into a command. The \(m_1; m_2\) form of sequencing we used in Section 2, which does not bind the return value of \(m_1\), can easily be desugared to \(x \leftarrow m_1; m_2\), where \(x\) does not appear free in \(m_2\). To keep the formalism simple, this syntactic sugar is not included in the rules.

The elaboration rules for declarations and programs appear in Figure 17. The rule for \texttt{val} declarations simply elaborates the expression and returns a pair of the bound variable and elaborated expression. Function declarations (both expression-level and priority-level) are recursive, and so elaborating them involves finding a fixed point. The elaboration of priority-monomorphic \texttt{fun} declarations elaborates the body in a context that includes all of the arguments, as well as the function itself. This is then placed inside \(n\) nested \(\lambda\)-abstractions to bind the required arguments. Finally, this expression is nested inside a fixed-point operator to introduce the recursive binding of the function name. Something similar occurs for priority-polymorphic \texttt{fun} declarations. The body is nested inside \(m\ \lambda\)-abstractions to introduce the expression variables, followed by \(n\) priority-level abstractions to introduce the priority variables. Finally, this whole expression is wrapped in a fixed-point operator.

The last elaboration judgment covers both toplevel declarations and programs, and produces a command corresponding to the entire PriML program. The first three rules for this judgment elaborate each form of toplevel declaration followed by the remainder of the program, and the final rule evaluates the “main” command that terminates the program. Priority and order declarations are erased from the program; their elaboration is simply the elaboration of the remainder of the
program. However, the priorities and ordering relations introduced by these declarations are collected into the partial order that will be returned at the end of elaboration. Each time a new ordering constraint $\vec{p}_1 < \vec{p}_2$ is introduced, we check that $\vec{p}_2 \not\leq \vec{p}_1$. This premise checks that the new constraint will not induce a cycle in the priority relation and ensure that the generated relation is a valid partial order. Ordinary declarations (val and fun) at the top level are elaborated using the declaration elaboration rules. Before elaborating these declarations, the current set of worlds and ordering constraints is loaded into the context as follows, assuming $P$ contains priorities $\{\vec{p}_1, \ldots, \vec{p}_n\}$ with ordering relation $\{(\vec{p}_1, \vec{p}_1'), \ldots, (\vec{p}_m, \vec{p}_m')\}$.

$$\Gamma^P = \vec{p}_1 \text{ prio}, \ldots, \vec{p}_n \text{ prio}, \top \leq \vec{p}_1, \ldots, \top \leq \vec{p}_n, \vec{p}_1 \leq \vec{p}_1', \ldots, \vec{p}_m \leq \vec{p}_m'$$

The elaborated expression is wrapped in a command and then introduced using binding. Loading the bindings into the context types the same constraints, expressions and commands as providing the partial order as a parameter to the judgment.

**Lemma 3.**
1. If $\Gamma^P, \Gamma \vdash C$, then $\Gamma \vdash^P C$.
2. If $\Gamma^P, \Gamma \vdash e : \tau$, then $\Gamma \vdash^P e : \tau$.
3. If $\Gamma^P, \Gamma \vdash m \sim \tau @ p$, then $\Gamma \vdash^P m \sim \tau @ p$.

**Proof.** We show part (1) by induction on the derivation of $\Gamma^P, \Gamma \vdash C$. The interesting case is $\Gamma', \vec{p}_1 \leq \vec{p}_2, \Gamma \vdash C$, where $\vec{p}_1 < \vec{p}_2 \in P$. Apply rule assume. Parts (2) and (3) are then straightforward inductions, the only interesting cases of which are applications of (1).

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

**Theorem 4 (Correctness of Elaboration).**
1. If $\Gamma \vdash \hat{e} \sim^e e : \tau$, then $\Gamma \vdash e : \tau$.
2. If $\Gamma \vdash i \sim^l m \sim \tau @ p$, then $\Gamma \vdash m \sim \tau @ p$.
3. If $\Gamma \vdash \hat{m} \sim^m m \sim \tau @ p$, then $\Gamma \vdash m \sim \tau @ p$.
4. If $\Gamma \vdash d \sim^d (x, e) : \tau$, then $\Gamma \vdash e : \tau$.
5. If $P; \Gamma \vdash G \sim^G m; P'$, then $\Gamma \vdash^{P \cup P'} m \sim \tau @ \bot$.

**Proof.**
1. By induction on the derivation of $\Gamma \vdash \hat{e} \sim^e e : \tau$.
2. By induction on the derivation of $\Gamma \vdash i \sim^l m \sim \tau @ p$.
3. By induction on the derivation of $\Gamma \vdash \hat{m} \sim^m m \sim \tau @ p$.
4. By induction on the derivation of $\Gamma \vdash d \sim^d (x, e) : \tau$.
   - $d = \text{val } x = e$. By inversion, $\Gamma \vdash \hat{e} \sim^e e : \tau$. By induction, $\Gamma \vdash e : \tau$.
   - $d = \text{fun } f(x_1 \ldots x_n) = \hat{e}$. By inversion, $\Gamma, f : \tau_f, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \hat{e} \sim^e e : \tau$. By induction, $\Gamma, f : \tau_f, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$. By $n$ applications of $\rightarrow \text{I}$, we have $\Gamma, f : \tau_f \vdash \lambda x_1, \ldots, \lambda x_n. e : \tau_f$. Finally, apply rule fix.
   - fun$[\pi_1 : C_1, \ldots, \pi_n : C_n] f(x_1 \ldots x_m) = \hat{e}$. By inversion,
     $$\Gamma, f : \tau_f, \pi_1 \text{ prio}, C_1, \ldots, \pi_n \text{ prio}, C_n, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash \hat{e} \sim^e e : \tau$$
     By induction, $\Gamma, f : \tau_f, \pi_1 \text{ prio}, C_1, \ldots, \pi_n \text{ prio}, C_n, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash e : \tau$. By $m$ applications of $\rightarrow \text{I}$, we have
     $$\Gamma, f : \tau_f, \pi_1 \text{ prio}, C_1, \ldots, \pi_n \text{ prio}, C_n, \lambda x_1, \ldots, \lambda x_m. e : \tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \tau$$
     By $n$ applications of $\forall \text{I}$, we have $\Gamma, f : \tau_f \vdash \Lambda \pi_1 : C_1 \ldots \Lambda \pi_n : C_n. \lambda x_1, \ldots, \lambda x_m. e : \tau_f$. Finally, apply rule fix.
5. By induction on the derivation of $P; \Gamma \vdash G \sim^G m; P'$.
   - $G = \text{priority } \vec{p} G'$. By inversion, $P \cup \{\vec{p}\}; \Gamma \vdash G' \sim^G m; P'$. By induction, $\Gamma \vdash^{P \cup P' \cup \{\vec{p}\}} m \sim \tau @ \bot$. 

PriML

We have developed a prototype implementation of PriML with extra arguments for the priorities, and their instantiations become function applications.

At the same time, the elaborator collects the priority and ordering declarations into a set of worlds 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.

Our compiler modifies the parser and elaborator of ML5/pgh [35], which also extends Standard ML with modal constructs, although for a quite different purpose. Elaboration converts the abstract syntax tree to a typed intermediate language, and type checks the code in the process. For PriML, 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.

5 IMPLEMENTATION

We have developed a prototype implementation of PriML. Our implementation compiles PriML to mlton-parmem [38], a parallel extension of Standard ML which is derived from the work of Spoonhower [43]. 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.

5.1 Compilation to Parallel ML

Our compiler modifies the parser and elaborator of ML5/pgh [35], which also extends Standard ML with modal constructs, although for a quite different purpose. Elaboration converts the PriML abstract syntax tree to a typed intermediate language, and type checks the code in the process. For the PriML extensions, elaboration proceeds broadly along the lines of the rules defined in Section 4.

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

The main result is the correctness of the elaboration of an entire PriML program in an empty context with only the priority ⊥ initially defined. This is a simple application of the last part of Theorem 4.

**Corollary 2.** If \( \{ \bot \} \); \( \vdash G \leadsto_G m \); \( P \), then \( \vdash \{ \bot \} \cup \{ P \} \).
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.

5.2 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.

The scheduling algorithm is based on the design of the two-priority work stealing algorithm of Muller et al. [33]. Each processor has a private deque [1] 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) and sets a public flag indicating the priority of the task on which it is working. A busy processor $q_1$ will periodically preempt its work and pick another “target” processor $q_2$ at random. If $q_1$ has a task of higher priority (in the programmer-defined partial order) than $q_2$’s task, it will send that task. 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.

5.3 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 [4]. We have also adapted the Fibonacci server, streaming music and web server benchmarks of prior work [33]. 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. [4] 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). The threads are prioritized using the three priorities described above.

**Fibonacci Server.** The Fibonacci server runs a foreground loop at 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, `smallfib`, `medfib` and `largefib`, depending on the size of the computation. The priorities are ordered `largefib < medfib < smallfib < fg`, so smaller computations will be prioritized but the input loop will always be responsive. 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 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. All requests are logged. The logs are periodically traversed by a background thread (priority stat_p), which analyzes the logs (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 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 [27], Xerox’s STAR [41], and Cedar [44]. 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.

In this paper, we use similar primitives for creating and synchronizing with threads as in these classic papers. We do not, however, support richer forms of synchronization between threads such as barriers, locks, and monitors. 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 [4, 15]. 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 [15].

Some languages, such as Concurrent ML [39], 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 appear to be the first to observe it in their work on Mesa [27]. 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, as for example defined by Babaoğlu, Marzullo,
and Schneider [4]. Since priority inversions can lead to undesirable program behavior, there has been much work on techniques for preventing them. The priority inheritance technique allows the priority of low-priority threads to be increased at run-time in order prevent higher priority threads from being delayed [27, 40]. Babaoğlu, Marzullo, and Schneider deem the complexity of the analysis of priority inheritance techniques to be a disadvantage [4]. They then provide a formal definition of priority inversions and describe protocols for preventing them, e.g., in a database transaction system where transactions could be aborted [4].

Beyond its complexity, priority inheritance can promote a low-priority thread, which can then take a significant time to complete. As a result, reasoning about the cost of a program becomes challenging. If instead, programmers are alerted statically to priority inversions, they could fix them and guarantee the desired properties. In this paper, this is the approach that we take. Instead of run-time heuristics, we present a type system that rejects programs with priority inversions, guaranteeing thus that type-safe programs are “well behaved.”

**Parallel Computing.** Although earlier work on multithreading was driven primarily by the need to develop interactive systems [21], 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. Even though this approach uses essentially the same threading primitives as traditional (competitive) multithreading, there are several important differences. First, a typical parallel program can create many more threads than the number of processors, rather than the small numbers of threads (typically on the order of the number of processors) used in traditional multithreading. Second, threads in parallel computing are used to maximize throughput by minimizing the completion time of the whole program, and are therefore scheduled cooperatively, i.e., without preemption [1, 2, 5, 7].

The ideas of implicit and cooperative threading go back to early parallel programming languages such as Id [3] and Multilisp [19], and many programming languages and language extensions for parallel systems have since been designed and implemented, including NESL [6], OpenMP, Cilk [18], Fork/Join Java [28], X10 [11], TBB [23], TPL [29], parallel Haskell [9, 26], parallel ML [17, 24, 38] and Habanero Java [22].

Although cooperative threading and competitive threading have historically been studied mostly separately, the widespread use of parallel hardware has motivated the need for bridging them. In a recent paper, Muller et al. show that this can be done by describing a language and a cost model that aims to unify the two models [33]. Our threading constructs are similar to theirs, allowing the expressions of interactive and parallel programs. They don’t have, however, consider the richer forms of priorities as we do here, relying instead on a fixed two-level priority domain.

**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 [37]. 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 [31] 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 [25, 36]...
used an explicit worlds formulation of S5, in which the “possible worlds” of the modal logic are made explicit in typing judgment. Worlds are interpreted as nodes in the distributed system, and an expression that is well-typed at a world is a computation that may be run on that node. Both type systems also include a “hybrid” connective $A \at w$, expressing the truth of a proposition $A$ at a world $w$. They interpret proofs of such a proposition as encapsulated computations that may be sent to $w$ to be run. Our type system uses a form of both of these features; priorities are explicit, and the types $\tau \text{cmd}[p]$ and $\tau \text{thread}[p]$ 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. [13, 14]). In these systems, the “next” modality of LTL is interpreted as a type of computations that may occur at the next stage of computation. Muller et al. [33] adapt these ideas to a type system for prioritized computation, but their system only considers two priorities: background and foreground. In principle, a priority type system based on LTL could be generalized to more than two priorities, but (because of the “linear” of LTL), such systems would be limited to totally ordered priorities.

Place-based systems (e.g. [10, 11, 45]), 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.

**Security Type Systems.** In this work, we track inter-thread communication (through synchronization) and require that high-priority threads do not wait for low priority threads. A similar problem occurs in the area of information flow control for concurrent programs (e.g. [8, 42]), in which systems must ensure that communication cannot cause data to pass from high-security threads to low-security threads. Type systems for ensuring these properties are generally finer-grained than ours, since they are concerned with the security of individual memory locations and not just overall properties of threads. Muller and Chong [32], however, use a place-based language (see above) derived from X10 [11] to increase the granularity of this tracking by effectively reducing inter-thread communication to message passing and ensuring that this communication is secure. Their work might therefore be a good basis for expanding our analysis to other forms of inter-thread communication (see Section 7).

7 DISCUSSION AND FUTURE WORK

In this paper, we consider a high-level language with a simple thread abstraction and one form of communication between threads: sync. In principle, the analysis could be extended to consider richer multithreading primitives such as locks and barriers, and other forms of inter-thread communication. The key would be to carefully track thread priorities through any such communication. For example, even innocuous-looking global state could be used to make an ad-hoc spinlock (e.g. while (flag == 0) {}), so a type system to prevent a high-priority thread from waiting on a low-priority thread in this way would have to track the priorities of threads that access the state. This kind of fine-grained tracking is common in security type systems, discussed briefly in Section 6, and such systems could provide inspiration for future work in this area.

The present work also imposes a strong restriction, in that it prevents thread synchronization except when it can be statically shown that this will not cause a priority inversion. Sometimes, programmers might prefer to use a combination of static techniques and dynamic techniques such as priority inheritance to prevent performance problems. It would therefore be interesting to design a system based on our analysis that enables the restriction to be relaxed in certain cases. Another
useful extension would be to make priorities first class, so that the priority of a thread could be chosen at runtime. This would make a static priority inversion analysis more difficult, and increase the relevance of dynamic techniques.

A final direction for future work would be to develop techniques for reasoning about, and ideally bounding, the responsiveness of programs that use the PriML approach to thread priorities, and to design and analyze more robust scheduling algorithms that can ensure that these responsiveness goals are met.

8 CONCLUSION

In this paper, we propose techniques for writing interactive multithreaded programs that support an expressive notion of partially ordered priorities and that statically preclude priority inversions in such programs. We show that these techniques are realistic through a number of example programs, written in our extension of the Standard ML language and compiled using the compiler we have presented. It is our hope that this work can form the basis of a new paradigm for writing multithreaded interactive programs.

ACKNOWLEDGEMENTS

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

REFERENCES


