Dynamic and First-class Priorities

MARELLE LEÓN, Illinois Institute of Technology, USA
STEFAN MULLER (ADVISOR), Illinois Institute of Technology, USA

Interactive parallel programs have varying responsiveness requirements for tasks of differing urgency, which has been met with the solution of thread priorities to determine the tasks' allocation of processor time. Previous priority-based language models limit the span of entire threads to a single priority. Given an approaching real-time deadline, tasks are unable to shift to a higher priority in order to match the changing requirements. We design a type system that enforces thread priorities and allows dynamic prioritization, treating priorities as first-class to reduce code complexity. We create a dependency graph-based cost model for our system and define well-formedness to exclude all priority inversions. We prove that programs under our type system produce well-formed graphs, and can be guaranteed an efficient response-time bound.

ACM Reference Format:
Marelle León and Stefan Muller (Advisor). 2023. Dynamic and First-class Priorities. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Parallel programs with tasks that greatly outnumber the processors face a problem of choosing which tasks to execute first. Treating threads equally may result in background tasks blocking user interactions. Prior work has resolved this issue by ordering competing tasks by priority. In the PriML [1] language, parallel programs define a set of priorities and assign each thread a priority matching its task's importance. Prompt scheduling then allocates more processor time to higher priority threads than low-priority competitors, creating responsiveness. However, this may result in priority inversions: a high priority thread waits to sync on a lower priority thread. If all processors are assigned to higher priority threads, neither thread will continue executing unless the lower is completed first, violating priority. Though PriML [1] has implemented priorities without inversions, it has limitations on writing thoroughly responsive programs. We design a new, more flexible type system.

```
let val priorities_in_order = [p1; p2; p3; p4; p5]
    val msecs_to_deadline = deadline - curr_time ()
in
    if msecs_to_deadline < 5 then
        change[List.nth priorities_in_order (floor msecs_to_deadline)]
    else
        ret ()
```

Fig. 1. Changing priority as deadline approaches

Say we spawn a thread for some low priority task. We need this task to finish before a deadline, and as this deadline approaches, it becomes more urgent. We introduce *change* so we can dynamically increase the priority of its thread. Now, the thread's current priority in the schedule is no longer bound to the part of its task that is already complete.

This exposes another inflexibility of the former system: requiring that we hard-code priority labels onto commands. This constraint makes programming dynamically prioritized threads sufficiently impractical, as seen in the rigid code in 2 2. Our solution frees priorities to be first-class, allowing them to inhabit a versatile data structure.

```
let val int_msecs_to_deadline = floor (deadline - curr_time ())
in
    if      (int_msecs_to_deadline) = 4 then change[p5]
    else if (int_msecs_to_deadline) = 3 then change[p4]
    else if (int_msecs_to_deadline) = 2 then change[p3]
    else if (int_msecs_to_deadline) = 1 then change[p2]
    else if (int_msecs_to_deadline) = 0 then change[p1]
    else ret ()
```

Fig. 2. Changing priority without first-class priorities

2 TYPE SYSTEM

We design a type system that tracks the priorities of threads and only allows programs that use them safely (i.e., have no priority inversions). Because priorities are now first class, we also need what priorities an expression may evaluate to. *Refinements* are typing constraints on values. We use refinements to capture priorities; for example, we refine the type of "if n < 4 then *medium* else *high*" with *medium* and *high* as its only possible priorities.

Threads run commands, which perform thread operations including *sync*, *spawn*, and *change*, and can be sequenced together. Since *sync* is the only operation that may result in a priority inversion, it determines the priority information needed for commands and threads. Due to *change*, threads may take on multiple priorities. To avoid inversions, we must have all priorities of the thread synced on be not-less-than the current priority of the thread synced from. We use a refinement to type a thread with all priorities it could be at according to its command.

However, a valid *sync* requires more information. The priorities of the thread synced on depend on the priority it starts at and those it changes to. We must also know what priority we are syncing from, which must be restricted to match the priority the previous command has ended at. We then give commands a type with three refinements: one for what priorities it's allowed to start at, one for what priorities it could have at any point, and one for what priorities it may have ended at.

3 TYPE SAFETY AND COST MODEL

We proved that our system correctly evaluates all well-typed programs and that each expression's type is preserved throughout the entire program. Preservation required a special clause: since priorities may change, *change* commands reset the starting priority for the following command sequence, necessitating a new starting refinement. Parallel programs can be represented as directed acyclic graphs (DAGs), each vertex representing a prioritized operation. We define well-formedness to constrain DAG priorities such that no operation depends on another with lower priority, and being strongly well-formed (SWF) as a local property on joins; both ensuring no priority inversions. We designed a cost model which produces strongly well-formed DAGs from programs our type system allows. We showed that all SWF DAGs are well-formed and, when given a prompt schedule, produce efficient computation times bound by the processors and operations at priorities not-less-than each thread operation's priority.

RELATED WORK

PriML [1] is the basis of our priority-based design for competitive threading. LiquidHaskell [2] explores refinements on a variety of types including recursive functions.

ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under grant number NSF-2150500.

REFERENCES

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

[2] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. *SIGPLAN Not.* 49, 9 (aug 2014), 269–282. https://doi.org/10.1145/2692915.2628161