ChipotLang

Final Progress Report

William Long (wl359)
Benjamin Posnick (bmp53)

Vision.

The vision of our project is a functional systems programming language that allows users to easily use synchronization primitives to control accesses to shared memory. ChipotLang is interpreted into OCaml code and uses continuations (by translating expressions into CPS) to support simulated concurrency using threads. ChipotLang makes it extremely easy for users to write concurrent programs as any expression can be passed into a thread and the lock, unlock, and join primitives are all built-in operators. ChipotLang is intended to be an educational tool to help bridge the gap between functional and systems/concurrent programming. It is not intended to be a high system performance language like C. This vision has changed from previous sprints (particularly the Alpha phase) in that instead of utilizing OCaml's libraries for threads, we instead implemented them from scratch with continuations.

Summary of Progress.

During this sprint, we implemented a CPS translation for every expression type in our language. We also implemented recursion with continuations by applying the recursion removal trick to a function definition *before* translating that function definition to CPS. Furthermore, we implemented currying in order to support the notion of multi-argument functions. Lastly, we modified our interpreter to support simulated concurrency with a scheduler that we control. We spent time writing documentation and factoring out helper functions as well.

The scheduling of threads is random and each thread is given a random number of steps to take before a context switch occurs. When such a context switch does occur, the current continuation and current environment are stored on a queue of running threads. When that thread is eventually rescheduled, the computation continues using the aforementioned continuation and store. A program terminates as soon as the main thread is done running, and individual threads terminate once they evaluate to a value. We modified all of our synchronization primitives (lock, unlock, join, etc) to work with our new scheduler. We support non-blocking by immediately context switching upon unsuccessfully acquiring a lock or by joining on a thread that is still running.

Activity Breakdown.

Both members of the team participated in *all* parts of the project as we used pair programming to complete all phases of the assignment. We found pair programming to be a very effective tool for our group throughout the entire project. Especially as we implemented CPS, a fairly complex language feature (that is incredibly difficult to debug), having two sets of eyes looking at the code written was very helpful.

Productivity Analysis.

Our group was extremely productive this sprint and completed by far the most complex aspects of our project. We certainly underestimated the difficulty of implementing CPS, but we were still able to complete it nonetheless. Our language is now able to do some fairly complex computations with concurrency now that we have continuations and a custom scheduler.

Grade.

Our team would give ourselves a scope grade of excellent. Implementing CPS for every expression in our language proved to be a very difficult task. It required reading some research papers as well as consulting the CS 6110 course materials. Both members of our group now understand CPS much better as a result. Additionally, a large portion of our interpreter had to be rewritten to support evaluation of the CPS translation that allowed for concurrency and context switching. Additionally, we had to rewrite all of our concurrency expression evaluation code as we had to create our own logic for doing operations (e.g. such as join) instead of simply relying on OCaml libraries.

Next Steps For Project.

Future goals for the project would be to expand some of the core features of the language by including features such as while loops, records, pattern matching / case statements, and some built-in functions (e.g. map, fold, filter). Another feature we would like to implement is message passing between threads. An implementation of this could utilize pi-calculus.