Continuations, threads, and LLVM

Kavon Farvardin
John Reppy

University of Chicago

June 2018

Motivation

- Compilers for concurrent and parallel languages can benefit from having an *Intermediate Representation* (IR) that supports operations on lightweight user-space threads.
- Such an IR can then represent the runtime-system mechanisms for concurrency/parallelism.
- Inlining of runtime-system code into the application code then enables cross-layer optimizations.
- Our *Parallel ML* (PML) compiler, which is part of the Manticore project, follows this approach.
- We are exploring the tradeoffs between several different runtime representations of threads in our compiler using LLVM. (**Work in progress.**)

Representing threads in an IR

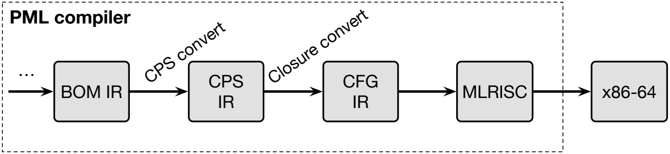
- How should thread state and operations on threads be represented in an IR for a concurrent or parallel language?
- One principled approach is to represent a suspended thread as a **continuation**.
- There is a long history of using surface-language continuations (`callcc`) to implement multithreading.

There are a number of different approaches to incorporating continuations in a compiler's IR.

- Appel-style CPS representation — all continuations are explicit
- Kelsey-style CPS representation – explicit continuations with annotations
- ANF with continuation binders – select continuations are reified

Continuations in an IR

- ANF+Continuations works well for writing runtime code and can be easily converted to the other representations or directly compiled to target code.
- Our PML compiler uses an ANF-style IR extended with continuation operations called BOM.



Representing threads in the BOM IR *(continued ...)*

$$\begin{array}{rcl}
\langle \mathit{exp} \rangle & ::= & \texttt{let}\ (x_1, ..., x_n) = \langle \mathit{prim} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
& | & \texttt{fun}\ f\ (x_1, ..., x_n) = \langle \mathit{exp} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
& | & \texttt{cont}\ k\ (x_1, ..., x_n) = \langle \mathit{exp} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
& | & \texttt{if}\ x\ \texttt{then}\ \langle \mathit{exp} \rangle\ \texttt{else}\ \langle \mathit{exp} \rangle \\
& | & \texttt{apply}\ f\ (x_1, ..., x_n) \\
& | & \texttt{throw}\ k\ (x_1, ..., x_n)
\end{array}$$

$$\begin{array}{rcl}
\langle \mathit{prim} \rangle & ::= & \texttt{create\_thread}\ (f) \\
& | & \textit{other primitive operations and values}
\end{array}$$

Representing threads in the BOM IR *(continued ...)*

$$
\begin{array}{rcl}
\langle \textit{exp} \rangle & ::= & \texttt{let}\ (x_1, ..., x_n) = \langle \textit{prim} \rangle\ \texttt{in}\ \langle \textit{exp} \rangle \\
& | & \texttt{fun}\ f\ (x_1, ..., x_n) = \langle \textit{exp} \rangle\ \texttt{in}\ \langle \textit{exp} \rangle \\
& | & \texttt{cont}\ k\ (x_1, ..., x_n) = \langle \textit{exp} \rangle\ \texttt{in}\ \langle \textit{exp} \rangle \\
& | & \texttt{if}\ x\ \texttt{then}\ \langle \textit{exp} \rangle\ \texttt{else}\ \langle \textit{exp} \rangle \\
& | & \texttt{apply}\ f\ (x_1, ..., x_n) \\
& | & \texttt{throw}\ k\ (x_1, ..., x_n)
\end{array}
$$

$$
\begin{array}{rcl}
\langle \textit{prim} \rangle & ::= & \texttt{create\_thread}\,(f) \\
& | & \textit{other primitive operations and values}
\end{array}
$$

Three forms for continuations:

- `cont` bindings
- `throw` expressions
- `create_thread` operator

Representing threads in the BOM IR *(continued ...)*

$$\begin{array}{lcl}
\langle \mathit{exp} \rangle & ::= & \texttt{let}\ (x_1, ..., x_n) = \langle \mathit{prim} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
 & \mid & \texttt{fun}\ f\ (x_1, ..., x_n) = \langle \mathit{exp} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
 & \mid & \texttt{cont}\ k\ (x_1, ..., x_n) = \langle \mathit{exp} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
 & \mid & \texttt{if}\ x\ \texttt{then}\ \langle \mathit{exp} \rangle\ \texttt{else}\ \langle \mathit{exp} \rangle \\
 & \mid & \texttt{apply}\ f\ (x_1, ..., x_n) \\
 & \mid & \texttt{throw}\ k\ (x_1, ..., x_n)
\end{array}$$

$$\begin{array}{lcl}
\langle \mathit{prim} \rangle & ::= & \texttt{create\_thread}\ (f) \\
 & \mid & \textit{other primitive operations and values}
\end{array}$$

Three forms for continuations:

- `cont` bindings
- `throw` expressions
- `create_thread` operator

Representing threads in the BOM IR *(continued ...)*

$$
\begin{array}{rcl}
\langle \mathit{exp} \rangle & ::= & \mathtt{let}\ (x_1, ..., x_n) = \langle \mathit{prim} \rangle\ \mathtt{in}\ \langle \mathit{exp} \rangle \\
 & | & \mathtt{fun}\ f\ (x_1, ..., x_n) = \langle \mathit{exp} \rangle\ \mathtt{in}\ \langle \mathit{exp} \rangle \\
 & | & \mathtt{cont}\ k\ (x_1, ..., x_n) = \langle \mathit{exp} \rangle\ \mathtt{in}\ \langle \mathit{exp} \rangle \\
 & | & \mathtt{if}\ x\ \mathtt{then}\ \langle \mathit{exp} \rangle\ \mathtt{else}\ \langle \mathit{exp} \rangle \\
 & | & \mathtt{apply}\ f\ (x_1, ..., x_n) \\
 & | & \mathtt{throw}\ k\ (x_1, ..., x_n) \\
 & & \\
\langle \mathit{prim} \rangle & ::= & \mathtt{create\_thread}\ (f) \\
 & | & \textit{other primitive operations and values}
\end{array}
$$

Three forms for continuations:

- `cont` bindings
- `throw` expressions
- `create_thread` operator

Representing threads in the BOM IR *(continued ...)*

$$\begin{array}{rcl}
\langle \mathit{exp} \rangle & ::= & \texttt{let}\ (x_1, \ldots, x_n) = \langle \mathit{prim} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
 & | & \texttt{fun}\ f\ (x_1, \ldots, x_n) = \langle \mathit{exp} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
 & | & \texttt{cont}\ k\ (x_1, \ldots, x_n) = \langle \mathit{exp} \rangle\ \texttt{in}\ \langle \mathit{exp} \rangle \\
 & | & \texttt{if}\ x\ \texttt{then}\ \langle \mathit{exp} \rangle\ \texttt{else}\ \langle \mathit{exp} \rangle \\
 & | & \texttt{apply}\ f\ (x_1, \ldots, x_n) \\
 & | & \texttt{throw}\ k\ (x_1, \ldots, x_n)
\end{array}$$

$$\begin{array}{rcl}
\langle \mathit{prim} \rangle & ::= & \texttt{create\_thread}\,(f) \\
 & | & \textit{other primitive operations and values}
\end{array}$$

Three forms for continuations:

- `cont` bindings
- `throw` expressions
- `create_thread` operator

Example: thread creation

Thread creation

```
fun fork f =
      fun f' () = (
            apply f ();
            throw Sched.dequeue ())
      let childK = thread_create f'
      in
        apply Sched.enqueue childK
```

Example: thread creation

Thread creation

```
fun fork f =
      fun f' () = (
            apply f ();
            throw Sched.dequeue ())
      let childK = thread_create f'
      in
        apply Sched.enqueue childK
```

We can also run the child thread first

```
fun fork f = cont parentK = ()
      in
        fun f' () = (
              apply f ();
              throw Sched.dequeue ())
        let childK = thread_create f'
        in
          apply Sched.enqueue parentK;
          throw childK ()
```

Example: context switch

Coroutine style explicit context switch.

```
fun yield () = cont k() = ()
      in
        Sched.enqueue k;
        throw Sched.dequeue ()
```

Example: context switch

Coroutine style explicit context switch.

```
fun yield () = cont k() = ()
      in
        Sched.enqueue k;
        throw Sched.dequeue ()
```

We can build all kinds of concurrency and parallelism mechanisms with this IR:

- locks and condition variables
- CML events / message-passing mechanisms
- work-stealing fork-join
- futures

Implementing continuations

Given an IR with continuations; we have to decide on a semantics for continuations and a supporting runtime model.

- first-class continuations
- one-shot continuations (may only be thrown to once)
- escape-continuations (essentially `setjmp`/`longjmp`)

First-class continuations are the most expressive and do not require any restrictions on their use in the IR
For example, we do not need to define **`create_thread`** as a primitive.

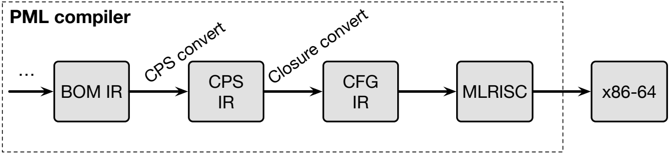
```
fun create_thread f =
      cont thdK () = (
            apply f ();
            throw Sched.dequeue ())
      in
        thdK
```

Implementing continuations *(continued ...)*

- Implementing first-class continuations on a traditional stack, however, is quite challenging.
- Early Scheme compilers used environment analysis to map continuations to stack-allocated frames (*e.g.*, Rabbit and Orbit). Note that Kelsey's IR encodes this analysis.
- Stack copying would be used to implement captured continuations.
- Segmented stacks were introduced (Chez Scheme) as a way to implement **`callcc`** more efficiently.
- Heap-allocated continuations (SML/NJ) provided a very simple implementation that abandoned the stack.

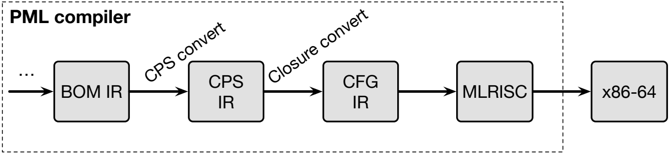
Choosing an approach

- Heap-allocated continuations provide a simple implementation of CPS, but giving up the stack has potentially significant performance costs.
- Previous empirical comparisons of runtime models are controversial [Appel-Shao '96] or dated [Clinger *et al.* '88 & '99].
- We are comparing four different runtime representations for continuations techniques using the LLVM code generator framework.



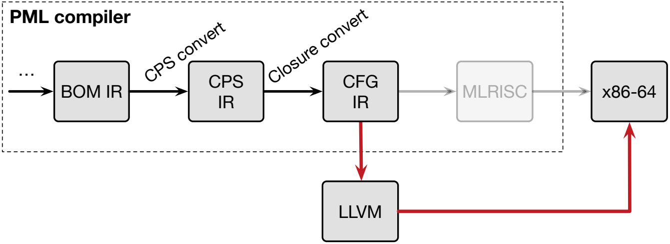
Choosing an approach

- Heap-allocated continuations provide a simple implementation of CPS, but giving up the stack has potentially significant performance costs.
- Previous empirical comparisons of runtime models are controversial [Appel-Shao ’96] or dated [Clinger *et al.* ’88 & ’99].
- We are comparing four different runtime representations for continuations techniques using the LLVM code generator framework.

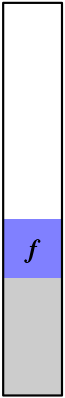


Choosing an approach

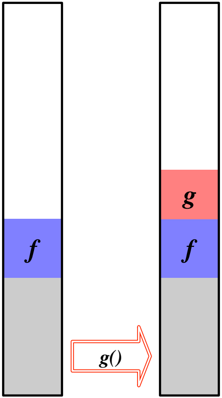
- Heap-allocated continuations provide a simple implementation of CPS, but giving up the stack has potentially significant performance costs.
- Previous empirical comparisons of runtime models are controversial [Appel-Shao ’96] or dated [Clinger *et al.* ’88 & ’99].
- We are comparing four different runtime representations for continuations techniques using the LLVM code generator framework.



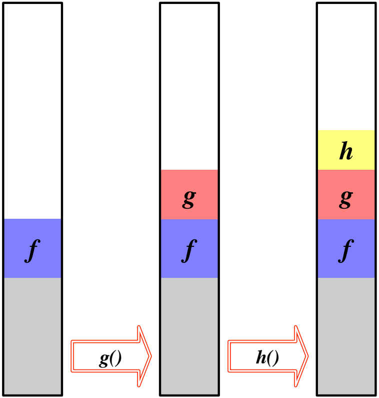
Contiguous stacks



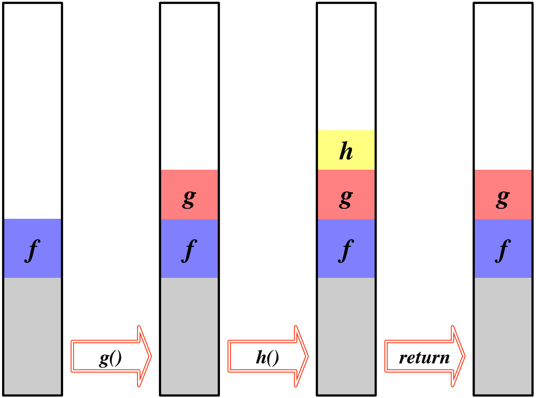
Contiguous stacks



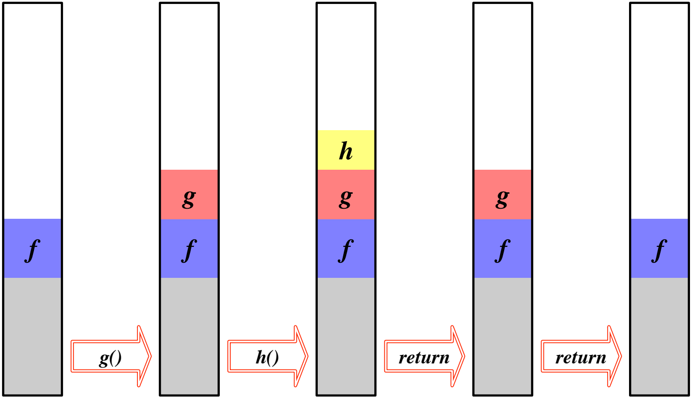
Contiguous stacks



Contiguous stacks



Contiguous stacks

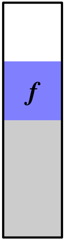


Contiguous stacks

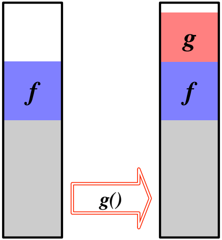
Pros and cons:

- \+ natural LLVM model
- \+ good locality across call/return
- \+ hardware optimized for return branch prediction
- \- stack overflow is a problem
- \- GC interface is more complicated and expensive
- \- potential race conditions when switching stacks
- \- thread creation and space overhead is high

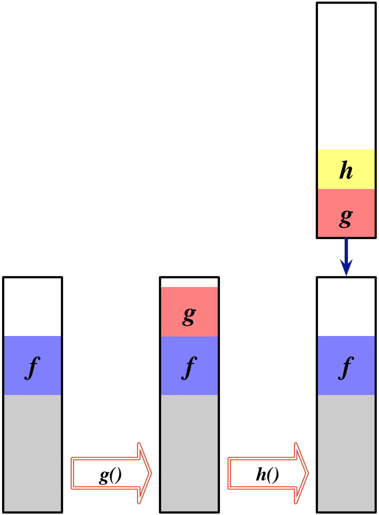
Segmented stacks



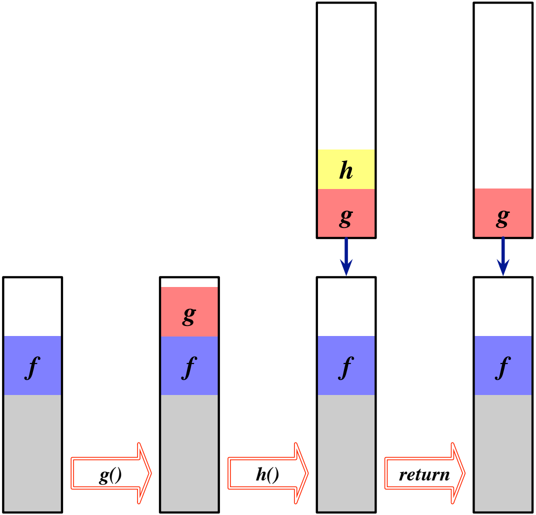
Segmented stacks



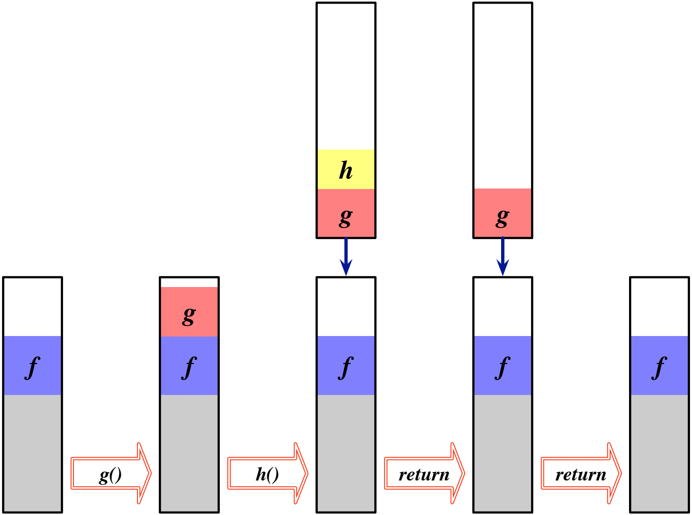
Segmented stacks



Segmented stacks



Segmented stacks

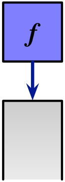


Segmented stacks

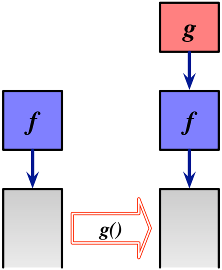
Pros and cons:

- + close to natural LLVM model
- + good locality across call/return
- + hardware optimized for return branch prediction
- + better space overhead than contiguous stacks
- - GC interface is more complicated and expensive
- - potential race conditions when switching stacks
- - thread creation overhead is high
- - additional calling overhead/complexity

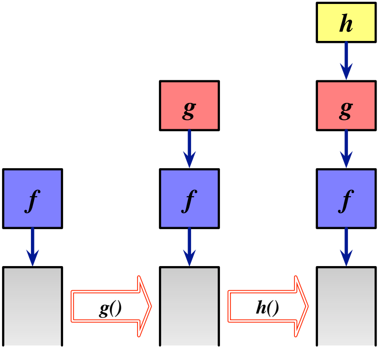
Heap-allocated stack stacks



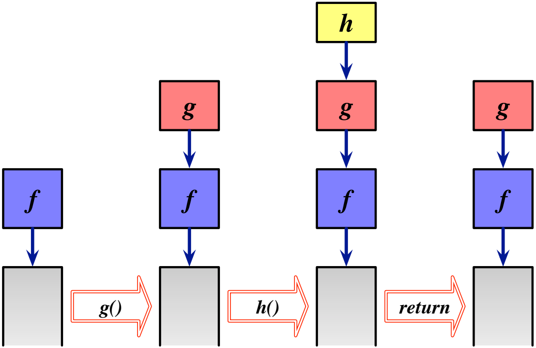
Heap-allocated stack stacks



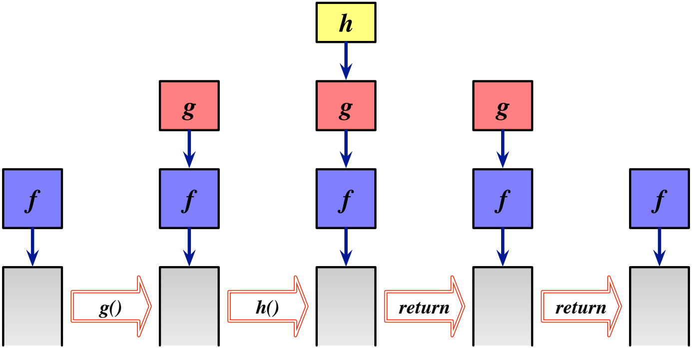
Heap-allocated stack stacks



Heap-allocated stack stacks



Heap-allocated stack stacks

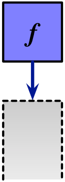


Heap-allocated stack frames

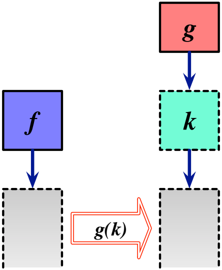
Pros and cons:

- \+ good locality across call/return
- \+ hardware optimized for return branch prediction
- \+ better space overhead than contiguous stacks
- \+ low thread creation overhead
- \- GC interface is more complicated and expensive
- \- potential race conditions when switching stacks
- \- additional calling overhead/complexity

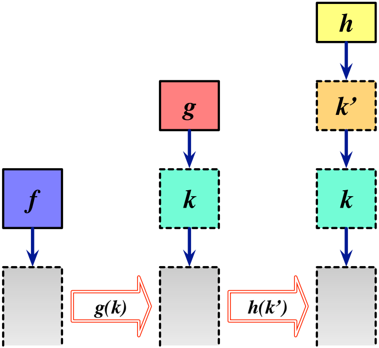
Heap-allocated continuation closures



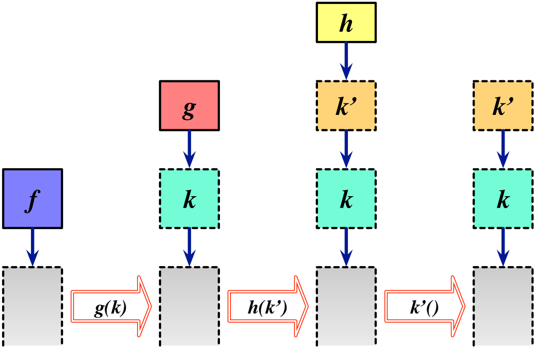
Heap-allocated continuation closures



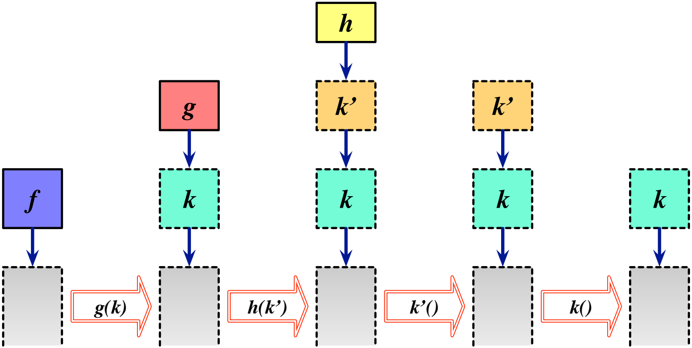
Heap-allocated continuation closures



Heap-allocated continuation closures



Heap-allocated continuation closures

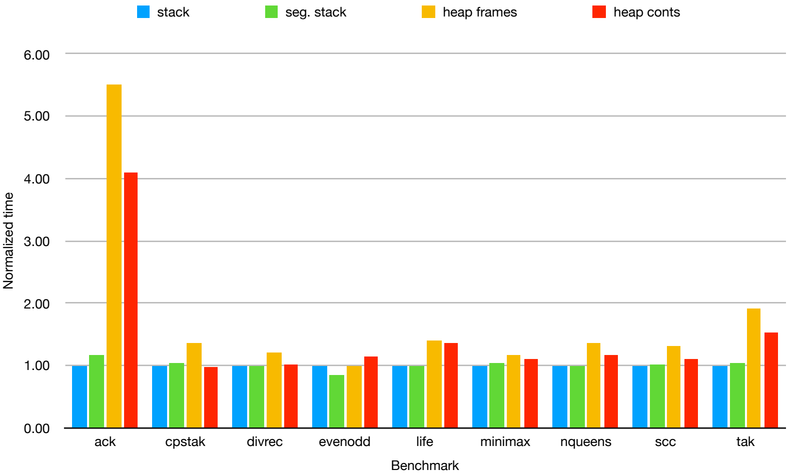


Heap-allocated continuation closures

Pros and cons:

- \+ simple implementation
- \+ simple GC interface
- \+ minimal space overhead
- \+ fast thread creation
- \+ no race conditions when context switching
- \- loses locality between calls and returns
- \- increased allocation rate
- \- cannot take advantage of return-branch prediction

Sequential costs



Concurrency costs

- We do not have complete numbers for threading experiments yet (because of some GC issues in the heap-allocated frame implementation).
- Previous experiments showed that heap-allocated continuations were significantly faster than stacks for thread creation.
- Segmented stacks performed poorly, but we have since improved the implementation and so we need to re-run the experiments.

Conclusion and Future Work

We need to complete our experiments before drawing firm conclusions, but here are some pre

- the overhead of linked frames appears to outweigh the locality benefits of reusing the frame
- segmented stacks may be the best choice if sequential performance is a high priority (although they were abandoned by Rust and Go because of poor implementation).
- the cost of heap-allocated continuations is low enough that the ease of implementation makes them a good choice.
- need more experiments to complete the study.