Push/enter vs eval/apply

Simon Marlow
Simon Peyton Jones

The question

Consider the call (f x y). We can either

- **Evaluate** f, and then **apply** it to its arguments, or
- **Push** x and y, and **enter** f
- Both admit fully-general tail calls
- Which is better?

Push/enter for (f x y)

- Stack of "pending arguments"
- Push y, x onto stack, enter (jump to) f
- f knows its own arity (say 1). It checks there is at least one argument on the stack.
- Grabs that argument, and executes its body, then enters its result (presumably a function) which consumes y

Eval/apply for (f x y)

- Caller evaluates f, inspects result, which must be a function.
- Extracts arity from the function value. (Say it is 1.)
- Calls fn passing one arg (exactly what it is expecting)
- Inspects result, which must be a function.
- Extracts its arity… etc

Known functions

- Often f is a known function

  let f x y = ... in ...(f 3 5)....
- In this case, we know f's arity statically; just load the arguments into registers and call f.
- This "known function" optimisation applies whether we are using push/enter or eval/apply
- So we only consider unknown calls from now on.

Program	Uneval (%)	Unknown (%) <	Unknown (%) =	Unknown (%) >	Known (%) <	Known (%) =	Known (%) >
anna	0.8	0.0	25.5	0.0	0.6	73.8	0.0
cacheprof	0.3	0.0	25.2	0.0	0.2	74.5	0.0
compress	0.0	0.0	1.6	0.0	0.0	98.4	0.0
fem	0.0	0.0	5.4	0.0	0.0	94.6	0.0
fulsom	0.4	0.0	25.0	0.0	0.2	74.8	0.0
hidden	0.1	0.0	13.8	0.0	0.0	86.1	0.1
infer	0.1	0.0	18.8	0.0	0.1	81.1	0.0
scs	0.5	0.0	17.3	0.0	0.0	82.5	0.2
circsim	0.0	0.0	14.5	0.0	0.0	85.5	0.0
fibheaps	5.1	5.8	8.3	0.0	0.0	85.3	0.6
typecheck	0.5	0.0	27.3	0.0	0.5	72.2	0.0
simple	0.0	0.0	49.2	0.0	0.0	50.8	0.0
Min	0.0	0.0	0.0	0.0	0.0	21.2	0.0
Max	18.7	8.3	78.8	1.1	3.9	100.0	1.6
Average	1.0	0.4	20.3	0.0	0.2	79.0	0.1

Figure 6. Anatomy of calls

Uncurried functions

- If f is an uncurried function:

  f :: (Int,Int) -> Int

  ….f (3,4)…
- Then a call site must supply exactly the right number of args
- So matching args expected by function with args supplied by call site is easier (=1).
- But we want efficient *curried* functions too
- And, in a lazy setting, can't do an efficient n-arg call for an unknown function, because it might not be strict.

Push/enter vs eval/apply

When calling an unknown function:

- the call site knows how many args are supplied
- the function knows how many args it is expecting

- Push/enter: function inspects data structure describing arguments
- Eval/apply: call site inspects data structure describing function

Push/enter vs eval/apply

- Both are reasonable for both strict and lazy evaluators
- Traditionally, strict languages have used eval/apply (Lisp interpreter), while lazy ones have used push/enter (G-machine, TIM..)
- Push/enter does handle currying particularly elegantly
- GHC has always used push/enter

But no one knows which better

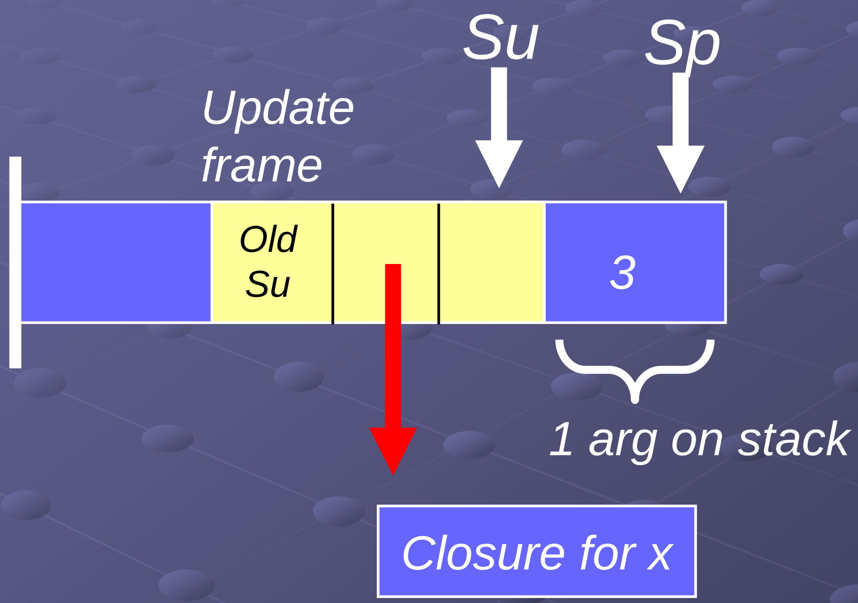
- Typically built rather deeply into an implementation
- Hence, hard to implement both
- Hence no good way to compare the two
- So implementors just stick their finger in the air
- We aim to close the question

Implementing push/enter

- Two entry points for each function:
  - "fast" for known calls
  - "slow" for unknown calls
- “Su” register points to deepest pending argument; so Sp-Su gives # of pending args
- Save/restore Su when pushing an update frame

Push/enter example

let x = f 3 in ...
where f has arity 2



f sees that there is only one argument on stack, so it

- *Updates the closure for x with (f 3)*
- *Removes the update frame*
- *And looks for further arguments*

Implementing eval/apply

- For unknown (f x y), jump to RTS code

  apply2(f,x,y)

  passing x,y in registers
- The RTS code evaluates f, tests arity etc
- RTS apply code is mechanically generated for many common patterns (apply2, apply3 etc)
- Exception cases by repeated calls

Call patterns (unknown calls)

Program	Argument pattern (% of all unknown calls) v	p	pv	pp	ppv	ppp	pppv	pppp	ppppp	OTHER
anna	0.0	29.6	0.0	69.3	0.0	1.1	0.0	0.0	0.0	0.0
cacheprof	0.0	91.6	0.0	8.1	0.0	0.3	0.0	0.0	0.0	0.0
compress	0.4	73.9	0.0	12.9	0.0	12.7	0.0	0.0	0.0	0.0
fem	0.0	91.3	0.0	8.1	0.0	0.6	0.0	0.0	0.0	0.0
fulsom	0.0	17.5	0.0	82.5	0.0	0.0	0.0	0.0	0.0	0.0
hidden	0.2	48.7	0.0	14.3	0.0	36.8	0.0	0.0	0.0	0.0
infer	0.0	51.8	0.0	48.1	0.0	0.1	0.0	0.0	0.0	0.0
scs	1.4	19.6	0.0	79.0	0.0	0.0	0.0	0.0	0.0	0.0
circsim	0.0	70.2	0.0	8.6	0.0	21.2	0.0	0.0	0.0	0.0
fibheaps	0.0	43.2	13.7	43.1	0.0	0.0	0.0	0.0	0.0	0.0
typecheck	0.0	89.5	0.0	10.5	0.0	0.0	0.0	0.0	0.0	0.0
simple	0.0	20.1	0.0	79.9	0.0	0.0	0.0	0.0	0.0	0.0
Min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Max	58.6	100.0	13.7	100.0	15.5	98.9	6.2	11.3	0.3	0.1
Average	5.2	54.4	0.3	34.4	0.3	5.2	0.1	0.1	0.0	0.0

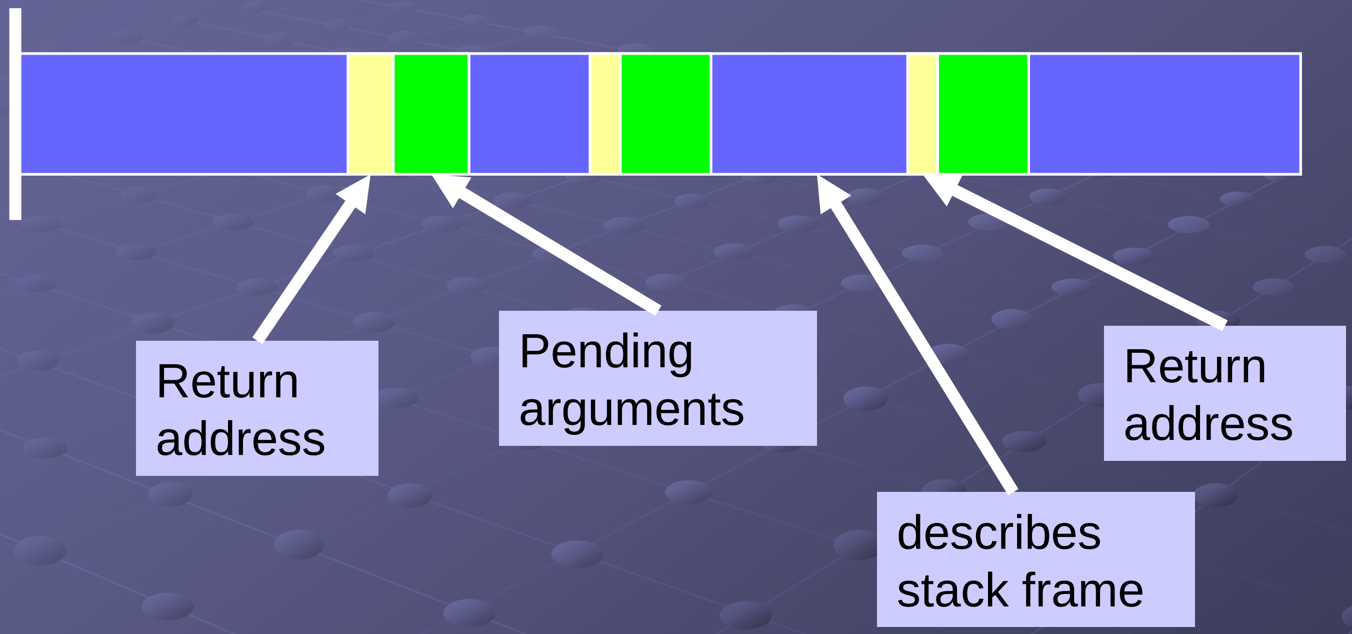
Figure 6. Argument patterns

Subtle costs

Push/enter has non-obvious costs

- Difficulties with stack/walking
- Difficult to compile to C--
- Burns a register (Su) to maintain current pending-arg count (+ need for save/restore in each update frame)
- Two entry points tiresome when hand-writing RTS built-ins

Stack-walking in push/enter



- **Problem: distinguishing pending args from return addresses**

Distinguishing return addresses

- Distinguish unboxed pending args with tags
- Could also do that with pointer args, but expensive (2 words/arg)
- We never found a satisfactory way of distinguishing return addresses from pending pointer args
- Address based schemes fail with dynamic linking; and OS fragility

Compiling to C--

- We'd like to compile to C--
- But the push/enter stack discipline is alien to C-- (because of the pending args)
- Unable to find a decent abstraction for C-- that accommodates pending args.
- Unsatisfactory fall-backs:
  - separate pending arg stack
  - ignore C-- stack, manage stack by hand

Qualitative conclusion

With deep reluctance I am forced to declare that

Eval/apply is a significantly simpler implementation technology for high-performance compilers

But how does it perform?

Program	Eval/apply change (Δ%)					
	Code size	Alloc	Instrs	Memory reads	Memory writes	Run-time
anna	-5.1	+1.7	+2.0	+2.5	-3.2	-0.7
cacheprof	-4.0	-0.0	+10.7	+10.3	+0.3	+4.1
circsim	+0.2	+0.0	+0.2	+1.0	-9.4	-4.7
compress	+2.2	-0.0	+1.8	+3.1	+3.6	+1.8
fem	-0.8	+0.0	-5.5	-3.2	-7.7	-
fibheaps	+1.0	+0.9	+3.3	+4.5	-3.1	-
fulsom	-2.1	+0.1	-2.5	-2.3	-7.9	-3.6
hidden	-2.4	+0.0	+3.3	+4.0	-6.1	+2.0
infer	-1.6	+0.2	+2.4	+2.4	-0.9	-
scs	-2.3	+0.0	+0.6	+1.4	-2.4	-3.7
simple	-1.8	+0.0	+3.5	+2.5	-4.7	+1.4
typecheck	+4.6	+1.2	+6.8	+6.6	-4.7	+3.0
Min	-5.1	-2.7	-10.1	-8.0	-13.6	-23.1
Max	+7.6	+2.9	+11.6	+20.8	+21.4	+6.8
G. Mean	+1.8	+0.1	+0.0	+1.0	-4.8	-2.4

Figure 8. Space and time

Program	Eval/apply change (%)		
	Tot alloc	PAPs	
	Words	Number	Words
anna	+1.7%	+934.7%	+761.2%
cacheprof	-0.0%	+9516.1%	+8326.7%
circsim	+0.0%	+15.4%	+9.1%
compress	-0.0%	+41.8%	+26.2%
fem	+0.0%	+1807.1%	+1622.6%
fibheaps	+0.9%	+12.7%	+12.7%
fulsom	+0.1%	+31576.0%	+35546.8%
hidden	+0.0%	+34.2%	+33.2%
infer	+0.2%	+184.6%	+164.3%
scs	+0.0%	+3454.5%	+3051.9%
simple	+0.0%	+2720.0%	+2543.8%
typecheck	+1.2%	+2840000.0%	+3036915.5%
Min	-2.7%	-12.0%	-11.5%
Max	+2.9%	+2840000.0%	+3036915.5%
Geometric Mean	+0.1%	+173.8%	+158.2%

Figure 7. Change in allocation

Conclusions

- Eval/apply does not change performance much either way
- But it's significantly simpler to think about and implement
- Complexity is located in one place (the RTS apply code), which can be hand tuned
- Less complexity elsewhere
- The balance is probably different for an interpreter
- Paper at http://research.microsoft.com/~simonpj