Inlined Code Generation for Smalltalk

Dave Mason and Daniel Franklin
Toronto Metropolitan University

©2024 Dave Mason

Why inline?

- obviate need for sends/calls
- access variables more efficiently
- use registers more effectively - modern hardware
- larger body of code for optimizers
- hierarchy of need:
    - static procedural
    - static vtables
    - dynamic dispatch - Java interfaces, dynamic languages

Why inline Smalltalk?

- dynamic dispatch, and also:
  - very small methods are encouraged
  - inheritance is heavily used
  - control structures are message sends
  - non-local returns are common

Smalltalk Method Dispatch

1. set T to the class of receiver, set S to selector
2. look up S in T methods - if found execute the method
3. else if T has a superclass, set T to the superclass and continue from step 2
4. else set T to the class of receiver, set S to `doesNotUnderstand:` with the original message as the parameter, and continue from step 2

- same as Java except DNU and offsets are statically detected (except interfaces)

What's known?

- Deutsch-Schiffman dispatch has been used since the 1980s
- about a dozen “special” control-flow selectors are inlined
- caches ameliorate the cost - Polymorphic Inline Cache
- Self in the early 1990s (and subsequently Javascript) compiled JITed code using per-target dispatch tables and inlining
- inlining almost exclusively in native-code JITs

What's new?

- Zag is new Smalltalk implementation
- two interchangeable execution modes - threaded (for debugging), and native (for speed)
- compiles methods to concrete classes (like Self & Javascript) - so `self` and `super` are usually exact
- compilation is "on demand" rather than JIT or AOT (only receiver class considered) - tiered:
    - threaded, no inlining
    - threaded, inlining
    - native version of previous

... Inlining

- no *ad hoc* selector inlining
- inlining is semantic - independent of target code - debugging is of the same code
- inlining is done breadth-first
- can be more intentional about inlining
- can be filtered to manage code explosion, possible strategies:
    - n-levels
    - maximum number of inlinings
    - inline until all possible block closures have been inlined
    - inline only sends with block closure parameters
    - incrementally inline - additional rounds

Inlining Opportunities

- Send to `self`, `super` or known type
- Send to `self`, `super` or known type when can’t inline
- Recursive send to `self`, `super` or known type
- Recursive tail-call send to `self`, `super` or known type
- Send to `self`, `super` or known type where the method is primitive
- Send `value`, `value:`, etc. to a literal `BlockClosure` - *safe*
- Send where there are few implementations of a method - *not safe*
- Send where the target is the result of a comparison primitive - *safe*

Conclusions

- there are advantages to semantic inlining
- we will report back when we have some actual benchmark results
- `https://github.com/Zag-Research/Zag-Smalltalk`

Questions?

we are hiring and have (many) open faculty positions in Toronto

dmason@torontomu.ca