Lecture 25

CIS 341: COMPILERS
Announcements

• HW 7: Optimization & Experiments
  – Available now
  – Due: April 29th

• My office hours today are cancelled.

• Final Exam:
  – Thursday, May 7th
  – 9:00AM
  – Moore 216
MARK & SWEEP GC
Example Object Graph

- Pointers in the stack, registers, and globals are *roots*
Mark and Sweep Garbage Collection

• Classic algorithm with two phases:

• Phase 1: Mark
  – Start from the roots
  – Do depth-first traversal, marking every object reached.

• Phase 2: Sweep
  – Walk over all allocated objects and check for marks.
  – Unmarked objects are reclaimed.
  – Marked objects have their marks cleared.
  – Optional: compact all live objects in heap by moving them adjacent to one another. (needs extra work & indirection to “patch up” pointers)
Results of Marking Graph

Unreachable blocks are garbage
Costs & Implications

- Need to generalize to account for objects that have multiple outgoing pointers.
- Depth-first traversal terminates when there are no children pointers or all children are already marked.
  - Accounts for cycles in the object graph.
- The Deutsch-Schorr-Waite algorithm breaks objects during the traversal.
  - All computation must be halted during the mark phase. (Bad for concurrent programs!)
- Mark & Sweep algorithm reads all memory in use by the program (even if it’s garbage!)
  - Running time is proportional to the total amount of allocated memory (both live and garbage).
  - Can pause the programs for long times during garbage collection.
Copying Garbage Collection

- Like mark & sweep: collects all garbage.
- Basic idea: use two regions of memory
  - One region is the memory in use by the program. New allocation happens in this region.
  - Other region is idle until the GC requires it.

- Garbage collection algorithm:
  - Traverse over live objects in the active region (called the “from-space”), copying them to the idle region (called the “to-space”).
  - After copying all reachable data, switch the roles of the from-space and to-space.
  - All dead objects in the (old) from-space are discarded en masse.
  - A side effect of copying is that all live objects are compacted together.
Cheney’s Algorithm (1)

- Idea: maintain two pointers into the to-space
  - Scan – points to the next piece of data to be examined
  - Free – points to the next available word of memory
  - Invariant: data pointed to by values between the scan and free pointers might need to be copied to the to-space
  - Leave behind “forwarding pointers” to the new copies.

- Crucial subroutine: (note implicit use of type information)

```c
pointer copy-forward(pointer p)
```

- If structure pointed to by p has already been copied, return the corresponding forwarding pointer.
- Otherwise:
  - Copy the structure pointed to by p into the to-space. (Incrementing the free pointer)
  - Mark the structure in from-space as copied and put a forwarding pointer in from-space to the copy in to-space
  - Return the pointer to the new copy in to-space
Cheney’s Algorithm (2)

• When garbage collection is triggered:
  – Initialize the free pointer to be beginning of to-space

• For each root R containing a pointer ptr:
  Set ptr’ = copy-forward(ptr)
  Set R := ptr’
  Set the scan pointer to ptr’.
  While (scan != free)
    – Increment the scan pointer (element-wise according to types of the fields in the underlying structure)
    – If the scan pointer points to a pointer ptr
      • Set *scan := copy-forward(ptr)
Run of Cheney's Algorithm

Memory at the point that GC is triggered.

CIS 341: Compilers
Run of Cheney's Algorithm

From-space

Roots

A
B
D

To-space

A'

A' = Marked as forwarded

= Copied, not yet scanned

= Copied & scanned

copy-forward on the root pointer.
Run of Cheney's Algorithm

Scan the first element of A' in to-space, copying B and modifying the pointer in the datastructure.

A' \[\text{scan}\]
B \[\text{free}\]
B' \[\text{Marked as forwarded}\]
C \[\text{Copied, not yet scanned}\]
D \[\text{Copied & scanned}\]
E \[\text{Marked as forwarded}\]

- \(\text{Marked as forwarded}\)
- \(\text{Copied, not yet scanned}\)
- \(\text{Copied & scanned}\)
Scan the second element of $A'$ in to-space, copying $C$ and modifying the pointer in the datastructure.
Run of Cheney's Algorithm

Scan the first element of B’ in to-space, copying D and modifying the pointer in the datastructure.

= Marked as forwarded
= Copied, not yet scanned
= Copied & scanned

CIS 341: Compilers
Run of Cheney's Algorithm

From-space

Roots

A

B

C

D

E

Scan the second element of B’ in to-space – it’s not a pointer.

To-space

A’

B’

C’

D’

Scan the second element

= Marked as forwarded

= Copied, not yet scanned

= Copied & scanned

CIS 341: Compilers
Run of Cheney's Algorithm

Scan the third element of B' in to-space, copying E and modifying the pointer in the datastructure.
Scan the first element of C’ in to-space, it has already been forwarded so just update the pointer.
Run of Cheney's Algorithm

Scan the second element of C' in to-space, it has already been forwarded so just update the pointer.

- Marked as forwarded
- Copied, not yet scanned
- Copied & scanned
Run of Cheney’s Algorithm

From-space

Roots

A’
A
C’
C
B’
B
D’
D
E’
E

Structures D and E have no pointers.

To-space

A’
B’
C’
D’
E’

= Marked as forwarded
= Copied, not yet scanned
= Copied & scanned

scan
free
Run of Cheney's Algorithm

Roots

From-space

To-space

Structures D and E have no pointers.

= Marked as forwarded
= Copied, not yet scanned
= Copied & scanned
Run of Cheney's Algorithm

From-space

Roots

A' A
B B'
C C'
D D'
E E'

To-space

A' B' C' D' E'

= Marked as forwarded
= Copied, not yet scanned
= Copied & scanned

Structures D and E have no pointers.
Run of Cheney's Algorithm

From-space

Roots
A
B
C
D
E

A' B' C' D' E'

To-space

A' B' C' D' E'

Structures D and E have no pointers.
Free = Scan, so we're finished with this root.

free = scan
= Marked as forwarded
= Copied, not yet scanned
= Copied & scanned
Run of Cheney's Algorithm

Roots

To-space

From-space

A'  B'  C'  D'  E'

free = scan
Tradeoffs of Copying Collection

• Benefits:
  – Simple, no stack space needed to implement the algorithm.
  – Running time is proportional to the number of reachable objects (not all allocated objects)
  – Automatically eliminates fragmentation by compacting memory during copy phase.
  – `malloc(n)` is implemented by `free := free + n`

• Drawbacks:
  – Twice as much memory is needed
  – Lots of memory traffic
  – Precise pointer/type information is required for traversal
  – Still can have long pauses
Baker’s Concurrent GC

- Variant of copying collection in which the program and the garbage collector run concurrently.
- Program holds only pointers to to-space
- On field-fetch operation, if the pointer is in from-space, run \texttt{copy-forward} instead of directly fetching.
  - Moves the structure to to-space to maintain the invariant
  - Incrementally garbage collects as the program touches data.
- When the to-space fills up, swap to/from by copying the roots and fixing up the stack and registers.

- Avoids long pauses due to copying
Generational Garbage Collection

- Observation: If an object has been reachable for a long time, it is likely to remain so.
- In long-running programs, mark & sweep and copying collection waste time and cache by scanning/copying old objects.
- Idea: Assign objects to different generations $G_0$, $G_1$, $G_2$, …
  - Generation $G_0$ contains newest objects, most likely to become garbage (< 10% live)
  - Younger generations scanned for garbage much more frequently than older generations.
  - New object eventually given tenure (promoted to the next generation) if they last long enough.
  - Roots of garbage collection for $G_0$ include objects in $G_1$
- Remembered sets:
  - Avoid scanning all tenured objects by keeping track of pointers from old objects to new objects. Compiler emits extra code to keep track of such pointer updates.
  - Pointers from old generations to new generations are uncommon
GC in Practice

• Combination of generational and incremental GC techniques reduce delay
  – Millisecond pause times
• Very large objects (e.g. big arrays) can be copied in a “virtual” fashion without doing a physical copy
  – Complicates the book keeping
• Some systems combine copying collection (for young data) with mark & sweep (for old data)
• Challenging to scale to server-scale systems with terabytes of memory
• Interactions with OS matter a lot
  – It can be cheaper to do GC than it is to start paging
• GC is here to stay (thanks to Java, C#, etc.)
REFERENCE COUNTING
Reference Counting

• Idea: Keep track of the number of references to a given object.
  – When creating a new reference to the object, increase the reference count
  – On a call to \texttt{free}, decrement the reference count
  – If the reference count is 0, the object can be deallocated immediately

• Deallocating an object will decrement reference counts of objects it points to
  – Deallocations can “cascade,” causing lots of objects to be deallocated

• Benefit: immediate reclamation of the space (no need to wait for garbage collector)

• Challenges:
  – Tracking reference counts efficiently
  – Cyclic data structures
Example Reference Counts

- Objects track reference counts.
Example Reference Counts

- On `free(x)`
Example Reference Counts

- On `free(x)`
Example Reference Counts

- **On** \texttt{free(x)}

\begin{center}
\begin{tikzpicture}
\node[draw, blue, fill=blue!20, minimum width=2cm, minimum height=2cm, label=right:EAX] (EAX) at (2,4) {2};
\node[draw, blue, fill=blue!20, minimum width=2cm, minimum height=2cm, label=right:EBX] (EBX) at (6,4) {3};
\node[draw, blue, fill=blue!20, minimum width=2cm, minimum height=2cm, label=right:1] (1) at (8,4) {1};
\node[draw, blue, fill=blue!20, minimum width=2cm, minimum height=2cm, label=right:1] (1) at (10,4) {1};
\node[draw, blue, fill=blue!20, minimum width=2cm, minimum height=2cm, label=right:2] (2) at (2,1) {2};
\node[draw, blue, fill=blue!20, minimum width=2cm, minimum height=2cm, label=right:x] (x) at (2,0) {x};
\node[draw, blue, fill=blue!20, minimum width=2cm, minimum height=2cm, label=right:1] (1) at (4,0) {1};
\draw[->, blue] (EAX) -- (EBX);
\draw[->, blue] (EBX) -- (1);
\draw[->, blue] (1) -- (1);
\draw[->, blue] (1) -- (2);
\draw[->, blue] (2) -- (x);
\end{tikzpicture}
\end{center}
Example Reference Counts

Note that the cycle won’t be freed.
Dealing with Cycles

• Option 1: Require programmers to explicitly null-out references to break cycles.

• Option 2: Periodically run GC to collect cycles

• Option 3: Require programmers to distinguish “weak pointers” from “strong pointers”
  – *weak pointers*: if all references to an object are “weak” then the object can be freed even with non-zero reference count.
  – “Back edges” in the object graph should be designated as weak
  – (Aside: weak pointers useful in GC settings too.)

• In practice: Reference counts
  – Apples Cocoa framework used ref counts, recent versions use GC
  – iOS supports “automatic reference counting”
COMPILER VERIFICATION
Compiler Verification

• 1967: Correctness of a Compiler for Arithmetic Expressions [McCarthy, Painter]

• 1972: Proving Compiler Correctness in a Mechanized Logic [Milner, Weyhrauch]

• … many interesting developments

See: Compiler Verification, A Bibliography [Dave, 2003]

• 2006-present: CompCert [Leroy, et al.]
  – (Nearly!) fully verified compiler from C to Power PC, ARM, etc.

• Others:
  – Vellvm: Verified LLVM [Zdancewic, et a.]
Motivation: Safety-critical Software

• How do you know that the program you are running is correct?

• Aircraft flight control software
• Automobile engine controllers
• Pacemakers
• Autonomous vehicles
• Embedded systems

• Formal verification is expensive and time consuming, but sometimes warranted…
Motivation: SoftBound/CETS

[Nagarakatte, et al. *PLDI ’09, ISMM ’10]*

- Buffer overflow vulnerabilities.
- Detect spatial/temporal memory safety violations in legacy C code.
- Implemented as an LLVM pass.
- What about correctness?

http://www.cis.upenn.edu/acg/softbound/
Vellvm Framework

- LLVM IR
- Type System and SSA
- Operational Semantics
- Syntax
- Memory Model
- Proof Techniques & Metatheory
- Coq
- OCaml Bindings
- Parser
- Printer
- LLVM
- Other Optimizations
- Target

C Source Code → LLVM IR → Transform → LLVM IR → Other Optimizations → Target

Extract
Vellvm Framework

- Type System and SSA
- Operational Semantics
- Syntax
- Memory Model

Proof Techniques & Metatheory

Coq

C Source Code -> LLVM IR

Verified Transform

OCaml Bindings

Parser, Printer

LLVM IR -> LLVM IR

Other Optimizations -> Target
Motivation: Compiler Bugs

[Yang et al. PLDI 2011]

Random test-case generation

Source Programs

 GCC

 LLVM

 64open

\{8 other C compilers\}

Csmith – compiler testing infrastructure

79 bugs: 25 critical

202 bugs

325 bugs in total
CompCert

- Initiated by Xavier Leroy of INRIA in 2006.
- Idea: Build a compiler using an interactive theorem prover.
  - Prove formally that each compilation translation pass is correct.
• Initiated by Xavier Leroy of INRIA in 2006.
• Idea: Build a compiler using an interactive theorem prover.
  – Prove formally that each compilation translation pass is correct.
  – Implemented in Coq
The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors.

This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

Finding and understanding bugs in C compilers
Yang et al. PLDI 2011
FORMALLY SPECIFYING SEMANTICS
Execution Models

• Interpretation:
  – program represented by abstract syntax
  – tree traversed by interpreter

• Compilation to native code:
  – program translated to machine instructions
  – executed by hardware

• Compilation to virtual machine code:
  – program translated to “virtual machine” instructions
  – interpreted (efficiently)
  – further translated to machine code
  – just-in-time compiled to machine code
Simple Imperative Language

id := X | Y | Z | ...

Variables

aexp := n | id | aexp + aexp |
      aexp - aexp | aexp * aexp

Arithmetic Expressions

bexp := true | false | aexp = aexp |
      !bexp | bexp && bexp

Boolean Expressions

cmd :=
  | SKIP
  | id ::= aexp
  | cmd ;; cmd
  | IFB bexp THEN cmd ELSE cmd FI
  | WHILE bexp DO cmd END

Assignment
Sequence
Conditional
Loop

Do nothing

See Vminus/Imp.v for the Coq formalism
Formal Semantics

• Basic idea: implement interpreters or simulators
  – Just as in the earliest 341 projects

• “small step”: \( \text{cmd} / \text{st} \longrightarrow \text{cmd}' / \text{st}' \)
  – say how a single step of computation affects the state
    \[ x ::= 3 \quad / \quad \{x=0\} \longrightarrow \text{skip} / \{x=3\} \]
  – Implementation as an interpreter:
    \[ \text{step} : (\text{cmd} \times \text{state}) \rightarrow (\text{cmd} \times \text{state}) \]

• “large step”: \( \text{cmd} / \text{st} \Downarrow \text{st}' \)
  – say how a command runs to completion to produce a final state
  – Implementation as an interpreter:
    \[ \text{eval} : (\text{cmd} \times \text{state}) \rightarrow \text{state} \]
Correct Execution?

• What does it mean for such a program to be executed correctly?

• Even at the interpreter level we could show equivalence between the small-step and the large-step operational semantics:

\[
\text{cmd} / \text{st} \xrightarrow{*} \text{SKIP} / \text{st}' \\
\text{iff} \\
\text{cmd} / \text{st} \downarrow \text{st}'
\]