Lecture 22

COMPILER DESIGN
• HW6: Analysis & Optimizations
  – Alias analysis, constant propagation, dead code elimination, register allocation
  – **Due**: Tuesday, December 17\textsuperscript{th} at 23:59
  – *May submit by Thursday, December 19\textsuperscript{th} at 23:59 with penalty*

• Final Exam
  – Scheduled for Friday, January 31\textsuperscript{st}, 9-11 AM
Some Further Details

• How to model function invocations?

• Context-sensitive vs. Context-insensitive (via an example)

• How to model context sensitivity?
Converting to SSA: Overview

• Start with the ordinary control flow graph that uses allocas
  – Identify “promotable” allocas
• Compute dominator tree information
• Calculate def/use information for each such allocated variable
• Insert $\phi$ functions for each variable at necessary “join points”

• Replace loads/stores to alloc’ed variables with freshly-generated $\%uids$

• Eliminate the now unneeded load/store/alloca instructions
Where to Place $\phi$ functions?

• Need to calculate the “Dominance Frontier”

• Node A **strictly dominates** node B if A dominates B and $A \neq B$
  – Note: A does not strictly dominate B if A does not dominate B or $A = B$

• The **dominance frontier** of a node B is the set of all CFG nodes y such that B dominates a predecessor of y but does not strictly dominate y
  – Intuitively: starting at B, there is a path to y, but there is another route to y that does not go through B

• Write $DF[n]$ for the dominance frontier of node n
Dominance Frontiers

• Example of a dominance frontier calculation results
• \( \text{DF}[1] = \{1\}, \quad \text{DF}[2] = \{1,2\}, \quad \text{DF}[3] = \{2\}, \quad \text{DF}[4] = \{1\}, \quad \text{DF}[5] = \{8,0\}, \quad \text{DF}[6] = \{8\}, \quad \text{DF}[7] = \{7,0\}, \quad \text{DF}[8] = \{0\}, \quad \text{DF}[9] = \{7,0\}, \quad \text{DF}[0] = \{\} \)
Algorithm For Computing $DF[n]$

- Assume that $doms[n]$ stores the dominator tree (so that $doms[n]$ is the *immediate dominator* of $n$ in the tree)

- Adds each $B$ to the DF sets to which it belongs

for all nodes $B$

  if $#(\text{pred}[B]) \geq 2$ \hspace{1cm} // (just an optimization)

  for each $p \in \text{pred}[B]$

    runner := $p$ \hspace{1cm} // start at the predecessor of $B$

    while (runner $\neq doms[B]$) \hspace{1cm} // walk up the tree adding $B$

      $DF[runner] := DF[runner] \cup \{B\}$

      runner := $doms[runner]$

    }

}
Insert $\phi$ at Join Points

- Lift the $DF[n]$ to a set of nodes $N$ in the obvious way:
  \[ DF[N] = \bigcup_{n \in N} DF[n] \]

- Suppose that at variable $x$ is defined at a set of nodes $N$.

\[
\begin{align*}
DF_0[N] &= DF[N] \\
DF_{i+1}[N] &= DF[DF_i[N] \cup N]
\end{align*}
\]

Let $J[N]$ be the least fixed point of the sequence:
\[ DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots \]
That is, $J[N] = DF_k[N]$ for some $k$ such that $DF_k[N] = DF_{k+1}[N]$
- $J[N]$ is called the “join points” for the set $N$

- We insert $\phi$ functions for the variable $x$ at each node in $J[N]$.
  - $x = \phi(x, x, \ldots, x)$; (one “$x$” argument for each predecessor of the node)
  - In practice, $J[N]$ is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes, just as in the dataflow solver.

- Intuition:
  - If $N$ is the set of places where $x$ is modified, then $DF[N]$ is the places where phi nodes need to be added, but those also “count” as modifications of $x$, so we need to insert the phi nodes to capture those modifications too…
Example Join-point Calculation

- Suppose the variable $x$ is modified at nodes 3 and 6
  - Where would we need to add phi nodes?

- $\text{DF}_0[\{3,6\}] = \text{DF}[\{3,6\}] = \text{DF}[3] \cup \text{DF}[6] = \{2,8\}$
- $\text{DF}_1[\{3,6\}]$
  - $= \text{DF}[\text{DF}_0[\{3,6\}] \cup \{3,6\}]$
  - $= \text{DF}[\{2,3,6,8\}]$
  - $= \text{DF}[2] \cup \text{DF}[3] \cup \text{DF}[6] \cup \text{DF}[8]$
  - $= \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\}$
- $\text{DF}_2[\{3,6\}]$
  - $= \ldots$
  - $= \{1,2,8,0\}$

- So $\text{J}[\{3,6\}] = \{1,2,8,0\}$, and we need to add phi nodes at those 4 spots
AUTOMATIC MEMORY MANAGEMENT (GC)
Plan

• Why Automatic Memory Management?

• Garbage Collection

• Three Techniques
  – Mark and Sweep
  – Stop and Copy
  – Reference Counting
Why Automatic Memory Management?

• Storage management is still a hard problem in modern programming

• C and C++ programs have many storage bugs
  – forgetting to free unused memory
  – dereferencing a dangling pointer
  – overwriting parts of a data structure by accident
  – and so on...

• Storage bugs are hard to find
  – a bug can lead to a visible effect far away in time and program text from the source
Type Safety and Memory Management

• Some storage bugs can be prevented in a strongly typed language
  – e.g., we cannot overrun the array limits

• Can types prevent errors in programs with manual allocation and deallocation of memory?
  – Some fancy type systems (linear types) were designed for this purpose
  – ... but they complicate programming significantly

• If we want type safety, we must use automatic memory management
Automatic Memory Management

• This is an old problem
  – Studied since the 1950s for LISP

• There are several well-known techniques for performing completely automatic memory management

• For a (long) while, they were unpopular outside Lisp family of languages
  – Just like type safety used to be unpopular
The Basic Idea

• When an object that takes memory space is created, unused space is automatically allocated
  – New objects are created by malloc or new in C/C++
• After a while there is no more unused space
• Some space is occupied by objects that will never be used again
• This space can be freed to be reused later
The Basic Idea (Cont.)

• How can we tell whether an object will “never be used again”?
  – In general it is impossible to tell
  – We will have to use a heuristic to find many (not all) objects that will
    never be used again

• Observation: a program can use only the objects that it can find
  
  let x : A = new A in { x = y; ... }

  – After $x = y$ there is no way to access the newly allocated object
Garbage

• An object $x$ is **reachable** if and only if
  – A register contains a pointer to $x$, or
  – Another reachable object $y$ contains a pointer to $x$

• One can find all reachable objects by starting from registers and following all the pointers

• An unreachable object can never be referred by the program
  – These objects are called **garbage**
Reachability is an Approximation

• Consider the program

\begin{verbatim}
  x = new A  
y = new B  
x = y  
  if alwaysTrue() then x = new A else x.foo() fi
\end{verbatim}

• After $x = y$ (assuming y becomes dead there)
  – The object A is not reachable anymore
  – The object B is reachable (through x)
  – Thus, B is not garbage and is not collected
  – But, object B is never going to be used
Tracing Reachable Values

- Assume the only register is the accumulator
  - it points to an object, and
  - this object may point to other objects, etc.
- The stack is more complex
  - each stack frame contains pointers
    - e.g., method parameters
  - each stack frame also contains non-pointers
    - e.g., return address
  - if we know the layout of the frame, we can find the pointers in it
A Simple Example

- We start tracing from acc and stack
  - they are called the roots
- Note that B and D are not reachable from acc or the stack
- Thus, we can reuse their storage
Elements of Garbage Collection

• Every garbage collection scheme has the following steps
  1. Allocate space as needed for new objects
  2. When space runs out
     a) Compute what objects might be used again (generally by tracing objects reachable from a set of “root” registers)
     b) Free the space used by objects not found in (a)
• Some strategies perform GC before the space actually runs out
Mark and Sweep [McCarthy 1960]

- When memory runs out, GC executes two phases
  - the mark phase: traces reachable objects
  - the sweep phase: collects garbage objects

- Every object has an extra bit: the **mark** bit
  - reserved for memory management
  - initially the mark bit is 0
  - set to 1 for the reachable objects in the mark phase
Mark and Sweep Example

After mark:

After sweep:
The Mark Phase

let todo = { all roots }
while todo ≠ ∅ do
    pick v ∈ todo
    todo ← todo - { v }
    if mark(v) = 0 then (* v is unmarked yet *)
        mark(v) ← 1
        let v₁,...,vₙ be the pointers contained in v
        todo ← todo ∪ {v₁,...,vₙ}
    fi
od
The Sweep Phase

• The sweep phase scans the heap looking for objects with mark bit 0
  – these objects have not been visited in the mark phase
  – they are garbage
• Any such object is added to the free list
• The objects with a mark bit 1 have their mark bit reset to 0
The Sweep Phase (Cont.)

/* sizeof(p) is the size of block starting at p */
p ← bottom of heap
while p < top of heap do
  if mark(p) = 1 then
    mark(p) ← 0
  else
    add block p...(p+sizeof(p)-1) to freelist
  fi
p ← p + sizeof(p)
end

Details

• While conceptually simple, this algorithm has some tricky details
  – which is typical of GC algorithms

• A serious problem with the mark phase
  – it is invoked when we are out of space
  – yet it needs space to construct the todo list
  – the size of the todo list is unbounded, so we cannot reserve space a priori
Mark and Sweep: Details

- The todo list is used as an auxiliary data structure to perform the reachability analysis.

- There is a trick to allow the auxiliary data to be stored in the objects:
  - **pointer reversal**: when a pointer is followed, reverse it to point to its parent
  - by Deutsch-Schorr-Waite (DSW)

- Similarly, the free list is stored in the free objects themselves.
Mark and Sweep: Evaluation

• Space for a new object is allocated from the new list
  – a block large enough is picked
  – an area of the necessary size is allocated from it
  – the left-over is put back in the free list

• Mark and sweep can fragment the memory

• Advantage: objects are not moved during GC
  – no need to update the pointers to objects
  – works for languages like C and C++
Another Technique: Stop and Copy

- Memory is organized into two areas
  - Old space: used for allocation
  - New space: used as a reserve for GC

- The heap pointer points to the next free word in the old space
  - Allocation just advances the heap pointer
Stop and Copy Garbage Collection

• Starts when the old space is full
• Copies all reachable objects from old space into new space
  – garbage is left behind
  – after the copy phase the new space uses less space than the old one before the collection
• After the copy the roles of the old and new spaces are reversed and the program resumes
Stop and Copy Garbage Collection: Example

Before collection:

\[ \text{root} \quad \xrightarrow{} \quad A \quad B \quad C \quad D \quad E \quad F \quad \text{new space} \]

After collection:

\[ \text{new space} \quad \xrightarrow{} \quad A \quad C \quad F \quad \text{free} \]

heap pointer

root

Compiler Design      Zhendong Su
Implementation of Stop and Copy

• We need to find all the reachable objects, as for mark and sweep
• As we find a reachable object we copy it into the new space
  – And we have to fix ALL pointers pointing to it!
• As we copy an object we store in the old copy a forwarding pointer to the new copy
  – when we later reach an object with a forwarding pointer we know it was already copied
• We still have the issue of how to implement the traversal without using extra space
• The following trick solves the problem:
  – partition the new space in three contiguous regions

```
start
\downarrow
copied and scanned
\downarrow
copied
\downarrow
empty
```

- copied objects whose pointer fields were followed and fixed
- copied objects whose pointer fields were NOT followed
Stop and Copy: Example (1)

- Before garbage collection
Stop and Copy: Example (3)

- Step 1: Copy the objects pointed by roots and set forwarding pointers (dotted arrow)
Stop and Copy: Example (3)

- Step 2: Follow the pointer in the next unscanned object (A)
  - copy the pointed objects (just C in this case)
  - fix the pointer in A
  - set forwarding pointer
Stop and Copy: Example (4)

- Follow the pointer in the next unscanned object (C)
  - copy the pointed objects (F in this case)
Stop and Copy: Example (5)

- Follow the pointer in the next unscanned object (F)
  - the pointed object (A) was already copied. Set the pointer same as the forwarding pointer
• Since scan caught up with alloc we are done
• Swap the role of the spaces and resume the program
The Stop and Copy Algorithm

while scan <> alloc do
  let O be the object at scan pointer
  for each pointer p contained in O do
    find O’ that p points to
    if O’ is without a forwarding pointer
      copy O’ to new space (update alloc pointer)
      set 1st word of old O’ to point to the new copy
      change p to point to the new copy of O’
    else
      set p in O equal to the forwarding pointer
    fi
  end for
  increment scan pointer to the next object
od
Stop and Copy: Details

• As with mark and sweep, we must be able to tell how large is an object when we scan it
  – And we must also know where are the pointers inside the object

• We must also copy any objects pointed to by the stack and update pointers in the stack
  – This can be an expensive operation
Stop and Copy: Evaluation

- Stop and copy is generally believed to be the fastest GC technique
- Allocation is very cheap
  - Just increment the heap pointer
- Collection is relatively cheap
  - Especially if there is a lot of garbage
  - Only touch reachable objects
- But some languages don’t allow copying (C, C++)
Why Doesn’t C Allow Copying?

- Garbage collection relies on being able to find all reachable objects
  - And it needs to find all pointers in an object
- In C or C++ it is impossible to identify the contents of objects in memory
  - E.g., how can you tell that a sequence of two memory words is a list cell (with data and next fields) or a binary tree node (with a left and right fields)?
  - Thus we cannot tell where all the pointers are
Conservative Garbage Collection

• But it is Ok to be conservative:
  – If a memory word looks like a pointer it is considered a pointer
    • it must be aligned
    • it must point to a valid address in the data segment
  – All such pointers are followed and we overestimate the reachable objects
• But we still cannot move objects because we cannot update pointers to them
  – What if what we thought to be a pointer is actually an account number?
Reference Counting [Collins 1960]

• Rather than wait for memory to be exhausted, try to collect an object when there are no more pointers to it
• Store in each object the number of pointers to that object
  – This is the reference count
• Each assignment operation has to manipulate the reference count
Implementation of Reference Counting

- `new` returns an object with a reference count of 1
- If `x` points to an object then let `rc(x)` refer to the object’s reference count
- Every assignment `x ← y` must be changed:
  
  \[
  \begin{align*}
  rc(y) & ← rc(y) + 1 \\
  rc(x) & ← rc(x) - 1 \\
  \text{if}(rc(x) == 0) & \text{ then mark } x \text{ as free} \\
  x & ← y
  \end{align*}
  \]
Reference Counting: Evaluation

• Advantages:
  – Easy to implement
  – Collects garbage incrementally without large pauses in the execution

• Disadvantages:
  – Manipulating reference counts at each assignment is very slow
  – Cannot collect circular structures
Garbage Collection: Evaluation

• Automatic memory management avoids some serious storage bugs
• But it takes away control from the programmer
  – e.g., layout of data in memory
  – e.g., when is memory deallocated
• Most GC implementations stop the execution during collection
  – not acceptable in real-time applications
Garbage Collection: Evaluation

• Garbage collection is going to be around for a while
• There are advanced garbage collection algorithms
  – Concurrent: allow the program to run while collection is happening
  – Generational: do not scan long-lived objects at every collection
    • JVM uses this kind
  – Parallel: several collectors working in parallel