CSC 548: Lecture 4

Overview

Graph coloring

Building interference graphs

Graph coloring algorithms

Implementing graph coloring

Graph coloring

What is a graph? What is a graph coloring? (Graph coloring came from coloring maps: what's the connection?)

What is the *K*-coloring problem? What is the time complexity of *K*-coloring a graph?

For *K*-coloring, what is a small degree node? What is a large degree node? Why are large degee nodes difficult?

What has this got to do with register allocation?

Interference graphs

What is an interference graph?

What is the interference graph for this program?

```
method factTR (n : Integer, a : Integer) : Integer
{
  if (n <= 0) {
    return a;
  } else {
    let x : Integer = n - 1;
    let y : Integer = n * a;
    return this.factTR (x, y);
  }
}
```

Interference graphs with precolored nodes

What is a *precolored node* in an interference graph?

What precolored nodes should be in the graph for this program?

```
method factTR (n : Integer, a : Integer) : Integer
{
  if (n <= 0) {
    return a;
  } else {
    let x : Integer = n - 1;
    let y : Integer = n * a;
    return this.factTR (x, y);
  }
}
```

Interference graphs with hints

What is a *hint* in an interference graph? (Appel calls these MOVEs.)

What hints should be in the graph for this program?

```
method factTR (n : Integer, a : Integer) : Integer
{
  if (n <= 0) {
    return a;
  } else {
    let x : Integer = n - 1;
    let y : Integer = n * a;
    return this.factTR (x, y);
  }
}
```

Building interference graphs

How can we build an interference graph for a program?

We need to calculate the *live* variables at each point in the program.

What are the live vars for this program?

```
method factTR (n : Integer, a : Integer) : Integer
{
  if (n <= 0) {
    return a;
  } else {
    let x : Integer = n - 1;
    let y : Integer = n * a;
    return this.factTR (x, y);
  }
}
```

Building interference graphs

How do we calculate the interference graph for a let block:

```
let var = exp; rest
```

How do we calculate the interference graph for a function call:

```
let var = obj.method (arg1, ... argN); rest
```

Building interference graphs for Hobbes is pretty easy! Why? (*Hint*: no imperative features.)

Greedy graph coloring

One heuristic for graph coloring:

```
Graph color (Graph g) {
  Id[] uncolored = g.getUncoloredNodes ();
  if (uncolored.length > 0) {
    Id n = uncolored[0];
    Access spill = frame.newLocal ();
    g = g.setColor (n, spill);
    g = color (g);
    return g;
  } else {
    return g;
  }
}
```

What's wrong with this code?

Graph coloring by simplification

Another heuristic for graph coloring:

```
Graph color (Graph g) {
  Id[] uncoloredSDs = g.getUncoloredSmallDegreeNodes ();
  if (uncoloredSDs.length () > 0) {
    Id n = uncoloredSDs[0];
    Id[] links = g.getLinks (n);
    g = g.removeNode (n);
    g = color (g);
    g = g.addNode (n);
    g = g.addLinks (n, links);
    find a register for n
  } else {
    return g;
  }
}
```

How does this work? What's wrong with this code?

Graph coloring with coalescing

Simplification doesn't deal with hints:

```
Graph color (Graph g) {
  Id[] uncoloredSDs = g.getUncoloredSmallDegreeNodes ();
  if (uncoloredSDs.length > 0) {
    Id n = uncoloredSDs[0];
    Id[] hints = g.getHints (n);
    if (hints.length > 0) {
      Id m = hints[0];
      if (we should coalesce m and n) { // WHEN?
        merge m and n, then recursively color // coalescing
      } else {
        remove the hint // freezing
      }
    } else {
      simplify as before
  } else {
    return g;
  }
}
```

Graph coloring with coalescing

When should we coalesce n and m?

- Simplistic: don't create any large-degree nodes.
- Briggs: coalesce if the new node has $< K$ large-degree neighbors.
- George: coalesce if every large degree neighbor of n is already a neighbor of m.

Each of these algorithms is safe (if simplification worked before, it still works). Why?

Graph coloring with coalescing and simplification

Our algorithm now looks like:

```
Graph color (Graph g) {
  Id[] uncoloredSDs = g.getUncoloredSmallDegreeNodes ();
  Id[] uncoloredLDs = g.getUncoloredLargeDegreeNodes ();
  if (uncoloredSDs.length > 0) {
    coalesce or simplify
  } else if (uncoloredLDs.length > 0) {
    what should we do now?
  } else {
    return g;
  }
}
```

Graph coloring large degree nodes

Coalescing and simplification remove all the small degree nodes.

We're just left with the annoying large degree nodes!

One solution: optimistic coloring... try simplification, but if this fails, spill, rewrite the source, and start all over again.

Optimistic coloring is good for RISC architectures, but bad on IA32. Why?

Graph coloring large degree nodes

An alternative algorithm: *aggressive coalescing*.

Coalesce nodes if possible, even if they create large degree nodes (we're already dealing with large degree nodes!)

When can we agressively coalesce two nodes?

What do we do with nodes we can't agressively coalesce?

Graph coloring large degree nodes

```
Graph color (Graph g) {
  ... } else if (uncoloredLDs.length > 0) {
    Id n = uncoloredLDs[0];
    Id[] hints = g.getHints (n);
    if (hints.length > 0) {
      Id m = hints[0];
      if (we can coalesce) {
        coalesce n and m
      } else {
        freeze n and m
      }
    } else {
      if (we can color n with an existing color) {
        color n
      } else {
        SPILL!
      }
    }
  } ...
}
```

Implementing graph coloring

This is your job!

Look at the classes in hobbes/regalloc.

You need to complete hobbes/regalloc/GraphColor.java.

Summary

Register allocation can be reduced to graph coloring by computing the interference graph for a program.

Graph coloring is NP-complete, so we have to use heuristics.

Register allocation heuristics include simplification and coalescing for small degree nodes, and agressive coalescing and spilling for large degree nodes.

Next week: OO