Lecture 25

EECS 483
COMPILER CONSTRUCTION

Announcements

- HW6: Analysis and Optimization
  - Due on Thursday, May 2
- Final Exam
  - 4-6pm April 29
  - Two Rooms:
    - Last name A-K: DOW1005
    - Last name L-Z: DOW1010
  - One-page, letter-sized, double-sided "cheat sheet" of notes permitted
  - Partial credit is awarded, don't leave anything blank!
- Office Hours
  - I (Max) am holding my usual Mon/Thu office hours (Zoom only) this week and next, except for April 29 (use the queue)

Exam Review

- Today: review of the topics thus far
  - Everything covered in lecture and homeworks is fair game, even if we don't specifically review it today.
  - One page of the final will be on the first half material
  - I will only use slides from previous lectures to ensure there's no new material covered today.
- Outline
  - Lambda Calculus/Scope
  - Type systems, subtyping
  - Dataflow analysis
  - Control-flow analysis/SSA
  - Register Allocation

Lectures 15-16

SCOPE, FIRST-CLASS FUNCTIONS

Free Variables and Scoping

```
let add = fun x → fun y → x + y
let inc = add 1
```

- The result of **add 1** is itself a function
  - After calling **add**, we can't throw away its argument (or its local variables) because those are needed in the function returned by add.

- We say that the variable x is *free* in fun y → x + y
  - Free variables are defined in an outer scope

- We say that the variable y is *bound* by "fun y" and its *scope* is the body "x + y" in the expression fun y → x + y

- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

Free Variable Calculation

- An OCaml function to calculate the set of free variables in a lambda expression:

```
let rec free_vars (e:exp) : VarSet.t =
  begin match e with
   | Var x        -> VarSet.singleton x
   | Fun(x, body) -> VarSet.remove x (free_vars body)
   | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
  end
```

- A lambda expression e is *closed* if free_vars e returns VarSet.empty
- In mathematical notation:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(\text{fun } x \rightarrow exp) &= fv(exp) \setminus \{x\} \quad \textit{('x' is a bound in exp)} \\ fv(exp_1\ exp_2) &= fv(exp_1) \cup fv(exp_2) \end{aligned}$$

Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might capture the free variables:

Note: x is *free* in (fun z → x)

(fun x → (x y)){(fun z → x)/y}
= fun x → (x (fun z -> x))

free x is *"captured"!!*

- Usually *not* the desired behavior
  - This property is sometimes called "dynamic scoping"
    The meaning of "x" is determined by where it is bound dynamically, not where it is bound statically.
  - Some languages (*e.g.*, emacs lisp) are implemented with this as a "feature"
  - But: it leads to hard-to-debug scoping issues

Alpha Equivalence

- Note that the names of bound variables don't matter to the semantics
  - *i.e.*, it doesn't matter which variable names you use, if you use them consistently:

    (fun x → y x) is the "same" as (fun z → y z)

    the choice of "x" or "z" is arbitrary, so long as we consistently rename them

Two terms that differ only by consistent renaming of *bound* variables are called *alpha equivalent*

- The names of *free* variables do matter:

  (fun x → y x) is *not* the "same" as (fun x → z x)

  Intuitively: y an z can refer to different things from some outer scope

Students who cheat by "renaming variables" are trying to exploit alpha equivalence…

Closure Conversion Summary

- A ***closure*** is a pair of an environment and a code pointer
  - the environment is a map data structure binding variables to values
  - environment could just be a list of the values (with known indices)
- Building a closure value:
  - code pointer is a function that takes an extra argument for the environment: $A \rightarrow B$ becomes $(Env * A \rightarrow B)$
  - body of the closure "projects out" then variables from the environment
  - creates the environment map by bundling the free variables
- Applying a closure:
  - project out the environment, invoke the function (pointer) with the environment and its "real" argument
- Hoisting:
  - Once closure converted, all functions can be lifted to the top level

Lectures 17-19

TYPE SYSTEMS, TYPE CHECKING

Static Program Analysis

- **Static** means the program is analyzed at **compile-time**
- Used for two main purposes in the compiler:
  - Last stage of the frontend: “Type checking” or “Semantic Analysis”
    - Not every program that passes parsing is valid
      - `int main() { return x; }`
      - `int main() { return “hello world”; }`
    - If the type checker fails, the program is rejected, like a parse error
    - After the program passes the frontend, we consider it well-formed and will compile it.
  - During optimization: “static analysis”
    - We can do more optimizations if we know more about the program
    - Are these equivalent programs?
      - `int main() { int y = f(); return 0; }`
      - `int main() { return 0; }`
    - We can optimize the first to the second if we establish that f is side-effect free.
    - Since they take place after the frontend, the analysis never rejects the program
- Next few weeks: type checking, after that optimization and analyis

Inference Rules

- We can read a judgment $G \vdash e$ as
  "the expression e is well scoped and has free variables in G"
- For any environment G, expression e, and statements s_1, s_2.

$$G \vdash \text{if } (e)\ s_1 \text{ else } s_2$$

  holds if $G \vdash e$ and $G \vdash s_1$ and $G \vdash s_2$ all hold.
- More succinctly: we summarize these constraints as an ***inference rule***:

Premises

Conclusion

$$\frac{G \vdash e \qquad G \vdash s_1 \qquad G \vdash s_2}{G \vdash \text{if } (e)\ s_1 \text{ else } s_2}$$

- Such a rule can be used for *any* substitution of the syntactic metavariables G, e, s_1 and s_2.

Scope-Checking Lambda Calculus

- Consider how to identify "well-scoped" lambda calculus terms
  - Given: G, a *set* of variable identifiers, e, a term of the lambda calculus
  - *Judgment*: $G \vdash e$ "the free variables of e are included in G"

$$\frac{x \in G}{G \vdash x}$$

"the variable x is free, but in scope"

$$\frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}$$

"G contains the free variables of e_1 and e_2"

$$\frac{G \cup \{x\} \vdash e}{G \vdash \text{fun } x \rightarrow e}$$

"x is available in the function body e"

Example Derivation Tree

$$\text{APP}\ \dfrac{\text{FUN}\ \dfrac{\text{FUN}\ \dfrac{\text{APP}\ \dfrac{\text{VAR}\ \dfrac{x \in \{x,y\}}{\{x,\ y\} \vdash x} \qquad \text{VAR}\ \dfrac{y \in \{x,y\}}{\{x,\ y\} \vdash y}}{\{x,\ y\} \vdash x\ y}}{\{x\} \vdash \text{fun}\ y\ \text{->}\ x\ y}}{\{\} \vdash (\text{fun}\ x\ \text{->}\ \text{fun}\ y\ \text{->}\ x\ y)} \qquad \text{FUN}\ \dfrac{\text{VAR}\ \dfrac{z \in \{z\}}{\{z\} \vdash z}}{\vdash (\text{fun}\ z\ \text{->}\ z)}}{\{\} \vdash (\text{fun}\ x\ \text{->}\ \text{fun}\ y\ \text{->}\ x\ y)\ (\text{fun}\ z\ \text{->}\ z)}$$

- Note: the OCaml function **scope_check** verifies the existence of this tree. The structure of the recursive calls when running **scope_check** is the same shape as this tree!
- Note that $x \in E$ is implemented by the function **VarSet.mem**

Simply-typed Lambda Calculus

- For the language in "tc.ml" we have five inference rules:

$\boxed{\text{INT}}$

$$\frac{}{E \vdash i : \text{int}}$$

$\boxed{\text{VAR}}$

$$\frac{x : T \in E}{E \vdash x : T}$$

$\boxed{\text{ADD}}$

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

$\boxed{\text{FUN}}$

$$\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \ : T \text{ -> } S}$$

$\boxed{\text{APP}}$

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1 \; e_2 : S}$$

- Note how these rules correspond to the code.

Checking Derivations

- A *derivation* or *proof tree* has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms* (i.e. rules with no premises)
  - Example: the INT rule is an axiom
- Goal of the type checker: verify that such a tree exists.
- Example1: Find a tree for the following program using the inference rules in oat.pdf:

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

Example2: There is no tree for this ill-scoped program:

```
var x2 = x1 + x1;
return(x2);
```

Type Safety

"Well typed programs do not go wrong."

– Robin Milner, 1978

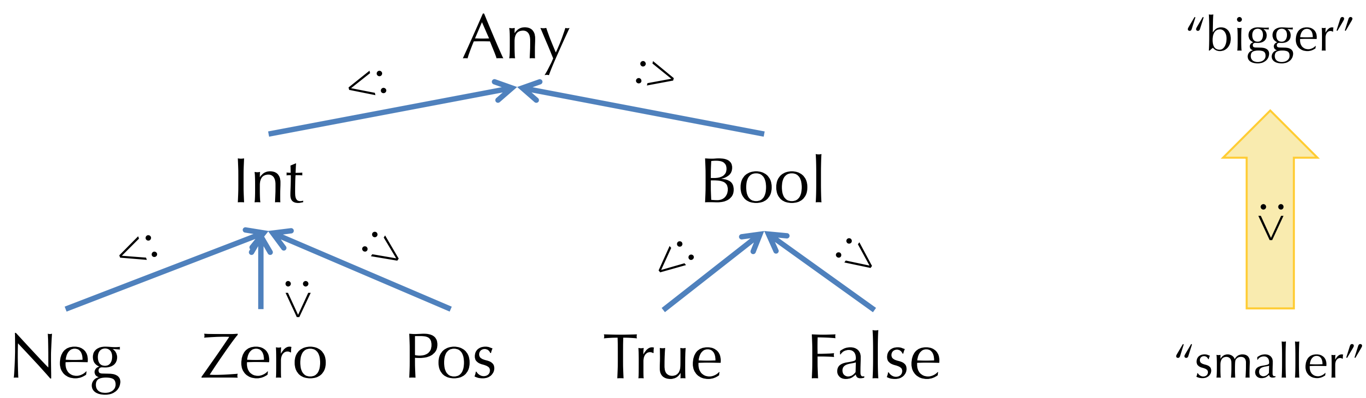
Theorem: (simply typed lambda calculus with integers)

If $\vdash e : t$ then there exists a value v such that $e \Downarrow v$.

- Note: this is a *very* strong property.
  - Well-typed programs cannot "go wrong" by trying to execute undefined code (such as 3 + (fun x -> 2))
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)

Subtyping and Upper Bounds

- If we think of types as sets of values, we have a natural inclusion relation: Pos ⊆ Int
- This subset relation gives rise to a *subtype* relation: Pos <: Int
- Such inclusions give rise to a *subtyping hierarchy*:



- Given any two types T_1 and T_2, we can calculate their *least upper bound* (LUB) according to the hierarchy.
  - Definition: LUB(T_1, T_2) is the smallest T such that T_1 <: T and T_2 <: T
  - Example: LUB(True, False) = Bool, LUB(Int, Bool) = Any
- Note: might want to add types for “NonZero”, “NonNegative”, and “NonPositive” so that set union on values corresponds to taking LUBs on types.

Soundness of Subtyping Relations

- We don't have to treat *every* subset of the integers as a type.
  - *e.g.*, we left out the type NonNeg

- A subtyping relation $T_1 <: T_2$ is *sound* if it approximates the underlying semantic subset relation.
- Formally: write $[\![T]\!]$ for the subset of (closed) values of type T
  - *i.e.*, $[\![T]\!] = \{v \mid \vdash v : T\}$
  - *e.g.*, $[\![\text{Zero}]\!] = \{0\}$, $[\![\text{Pos}]\!] = \{1, 2, 3, \ldots\}$

- If $T_1 <: T_2$ implies $[\![T_1]\!] \subseteq [\![T_2]\!]$, then $T_1 <: T_2$ is sound.
  - *e.g.*, Pos <: Int is sound, since $\{1,2,3,\ldots\} \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
  - *e.g.*, Int <: Pos is *not* sound, since it is *not* the case that $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$

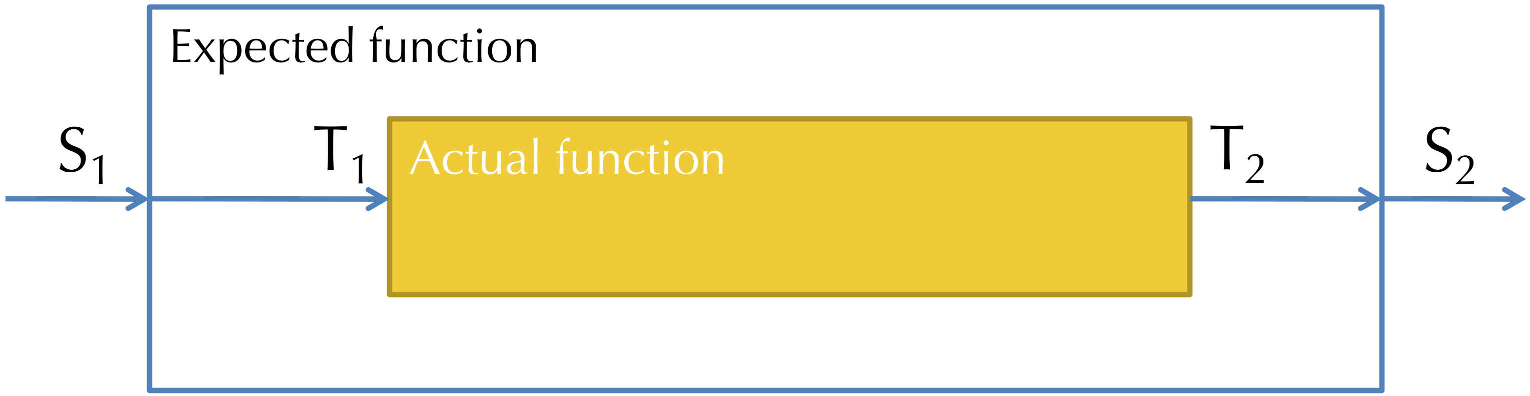
Extending Subtyping to Other Types

- What about subtyping for tuples?
  - Intuition: whenever a program expects something of type $S_1 * S_2$, it is sound to give it a $T_1 * T_2$.
  - Example: (Pos * Neg) <: (Int * Int)

$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

Subtyping for Function Types

- One way to see it:



- Need to convert an S_1 to a T_1 and T_2 to S_2, so the argument type is ***contravariant*** and the output type is ***covariant***.

$$\frac{S_1 <: T_1 \qquad T_2 <: S_2}{(T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2)}$$

Lectures 20-22

OPTIMIZATION, DATAFLOW ANALYSIS

Safety

- Whether an optimization is *safe* depends on the programming language semantics.
  - Languages that provide weaker guarantees to the programmer permit more optimizations but have more ambiguity in their behavior.
  - *e.g.*, In C, loading from initialized memory is undefined, so the compiler can do anything if a program reads uninitalized data.
  - *e.g.*, In Java tail-call optimization (which turns recursive function calls into loops) is not valid because of "stack inspection".
- Example: *loop-invariant code motion*
  - Idea: hoist invariant code out of a loop

```
while (b) {
  z = y/x;
  …                     // y, x not updated
}
```

```
z = y/x;
while (b) {
  …                     // y, x not updated
}
```

- Is this more efficient?
- Is this safe?

Common Subexpression Elimination

- *fold redundant computations together*
  - in some sense, it's the opposite of inlining
- Example:

```
a[i] = a[i] + 1
```

compiles to:

```
[a + i*4] = [a + i*4] + 1
```

Common subexpression elimination removes the redundant add and multiply:

```
t = a + i*4; [t] = [t] + 1
```

- For safety, you must be sure that the shared expression always has the same value in both places!

Unsafe Common Subexpression Elimination

- Example: consider this OAT function:

```
unit f(int[] a, int[] b, int[] c) {
    var j = …; var i = …; var k = …;
    b[j] = a[i] + 1;
    c[k] = a[i];
    return;
}
```

- The optimization that shares the expression a[i] is unsafe… why?

```
unit f(int[] a, int[] b, int[] c) {
    var j = …; var i = …; var k = …;
    t = a[i];
    b[j] = t + 1;
    c[k] = t;
    return;
}
```

Dataflow Analysis

- *Idea*: compute liveness information for all variables simultaneously.
  - Keep track of sets of information about each node
- Approach: define *equations* that must be satisfied by any liveness determination.
  - Equations based on "obvious" constraints.
- Solve the equations by iteratively converging on a solution.
  - Start with a "rough" approximation to the answer
  - Refine the answer at each iteration
  - Keep going until no more refinement is possible: a *fixpoint* has been reached
- This is an instance of a general framework for computing program properties: dataflow analysis

A Worklist Algorithm

- Use a FIFO queue of nodes that might need to be updated.

```
for all n, in[n] := Ø, out[n] := Ø
w = new queue with all nodes
repeat until w is empty
    let n = w.pop()                          // pull a node off the queue
      old_in = in[n]                         // remember old in[n]
      out[n] := ∪_{n'∈succ[n]} in[n']
      in[n] := use[n] ∪ (out[n] - def[n])
      if (old_in != in[n]),                  // if in[n] has changed
          for all m in pred[n], w.push(m)    // add to worklist
end
```

Generalizing Dataflow Analyses

- The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well.
  - Reaching definitions analysis
  - Available expressions analysis
  - Alias Analysis
  - Constant Propagation
  - These analyses follow the same 3-step approach as for liveness.

- To see these as an instance of the same kind of algorithm, the next few examples to work over a canonical intermediate instruction representation called *quadruples*
  - Allows easy definition of def[n] and use[n]
  - A slightly "looser" variant of LLVM's IR that doesn't require the "static single assignment" – i.e. it has *mutable* local variables
  - We will use LLVM-IR-like syntax

$\mathcal{L}$ as a Partial Order

- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq \ell_1$

- Properties of a partial order:
  - *Reflexivity:* $\ell \sqsubseteq \ell$
  - *Transitivity:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_3$ implies $\ell_1 \sqsubseteq \ell_2$
  - *Anti-symmetry:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$ implies $\ell_1 = \ell_2$

- Examples:
  - Integers ordered by $\leq$
  - Types ordered by <:
  - Sets ordered by $\subseteq$ or $\supseteq$

Meets and Joins

- The combining operator $\sqcap$ is called the "meet" operation.
- It constructs the *greatest lower bound*:
  - $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_1$ and $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_2$
    "the meet is a lower bound"
  - If $\ell \sqsubseteq \ell_1$ and $\ell \sqsubseteq \ell_2$ then $\ell \sqsubseteq \ell_1 \sqcap \ell_2$
    "there is no greater lower bound"

- Dually, the $\sqcup$ operator is called the "join" operation.
- It constructs the *least upper bound*:
  - $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$
    "the join is an upper bound"
  - If $\ell_1 \sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$ then $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$
    "there is no smaller upper bound"

- A partial order that has all meets and joins is called a *lattice*.
  - If it has just meets, it's called a *meet semi-lattice*.

Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node n):
- $out[n] := F_n(in[n])$

- Equivalently: $out[n] := F_n(\sqcap_{n' \in pred[n]} out[n'])$
  - By definition of in[n]
- We can write this as a simultaneous update of the vector of out[n] values:
  - let $x_n = out[n]$
  - Let $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ it's a vector of points in $\mathcal{L}$
  - $\mathbf{F}(\mathbf{X}) = (F_1(\sqcap_{j \in pred[1]} out[j]), F_2(\sqcap_{j \in pred[2]} out[j]), \ldots, F_n(\sqcap_{j \in pred[n]} out[j]))$

- Any solution to the constraints is a *fixpoint* **X** of **F**
  - i.e. $\mathbf{F}(\mathbf{X}) = \mathbf{X}$

Dataflow Analysis: Summary

- Many dataflow analyses fit into a common framework.
- Key idea: *Iterative solution* of a system of equations over a *lattice* of constraints.
  - Iteration terminates if flow functions are monotonic.

- Dataflow analyses as presented work for an "imperative" intermediate representation.
  - The values of temporary variables are updated ("mutated") during evaluation.
  - Such mutation complicates calculations
  - SSA = "Single Static Assignment" eliminates this problem, by introducing more temporaries – each one assigned to only once.
  - Next up: Converting to SSA, finding loops and dominators in CFGs

Lectures 23-24

CONTROL-FLOW ANALYSIS, REGISTER ALLOCATION

Single Static Assignment (SSA)

- LLVM IR names (via `%uids`) *all* intermediate values computed by the program.
- It makes the order of evaluation explicit.
- Each `%uid` is assigned to only once
  - Contrast with the mutable quadruple form
  - Note that dataflow analyses had these kill[n] sets because of updates to variables…
- Naïve implementation of backend: map `%uids` to stack slots
- Better implementation: map `%uids` to registers (as much as possible)

- Question: How do we convert a source program to make maximal use of `%uids`, rather than alloca-created storage?
  - two problems: control flow & location in memory

- Then: How do we convert SSA code to x86, mapping `%uids` to registers?
  - Register allocation.

Alloca vs. %UID

- Current compilation strategy:

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
%x = alloca i64
%y = alloca i64
store i64* %x, 3
store i64* %y, 0
%x1 = load %i64* %x
%tmp1 = add i64 %x1, 1
store i64* %x, %tmp1
%x2 = load %i64* %x
%tmp2 = add i64 %x2, 2
store i64* %y, %tmp2
```

- Directly map source variables into `%uids`?

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

```
%x1 = add i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %x2, 2
```

- Does this always work?

What about If-then-else?

- How do we translate this into SSA?

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %z2 = %add i64 ???, 3
```

- What do we put for `???`

Phi Functions

- Solution: ϕ functions
  - Fictitious operator, used only for analysis
    - implemented by Mov at x86 level
  - Chooses among different versions of a variable based on the path by which control enters the phi node.

    %uid = phi <ty> v_1, <label$_1$>, … , v_n, <label$_n$>

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %x4 = phi i64 %x2, %then, %x3, %else
  %z2 = %add i64 %x4, 3
```

Register Allocation

Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is live at the same time as m
3. Try to color the graph
   - Each color corresponds to a register
4. In case step 3. fails, “spill” a register to the stack and repeat the whole process.
5. Rewrite the program to use registers

Live Variable Analysis

- A variable v is *live* at a program point if v is defined before the program point and used after it.
- Liveness is defined in terms of where variables are *defined* and where variables are *used*

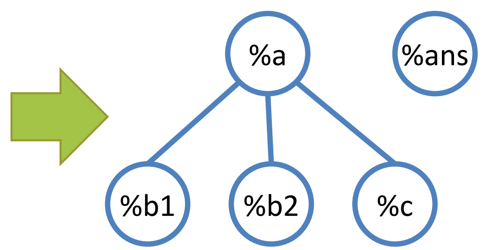
- Liveness analysis: Compute the live variables between each statement.
  - May be *conservative* (i.e. it may claim a variable is live when it isn't) so because that's a safe approximation
  - To be useful, it should be more *precise* than simple scoping rules.

- Liveness analysis is one example of *dataflow analysis*
  - Other examples: Available Expressions, Reaching Definitions, Constant-Propagation Analysis, …

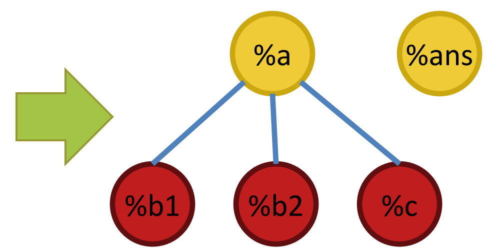
Interference Graphs

- Nodes of the graph are %uids
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



Interference Graph



2-Coloring of the graph
red = r8
yellow = r9

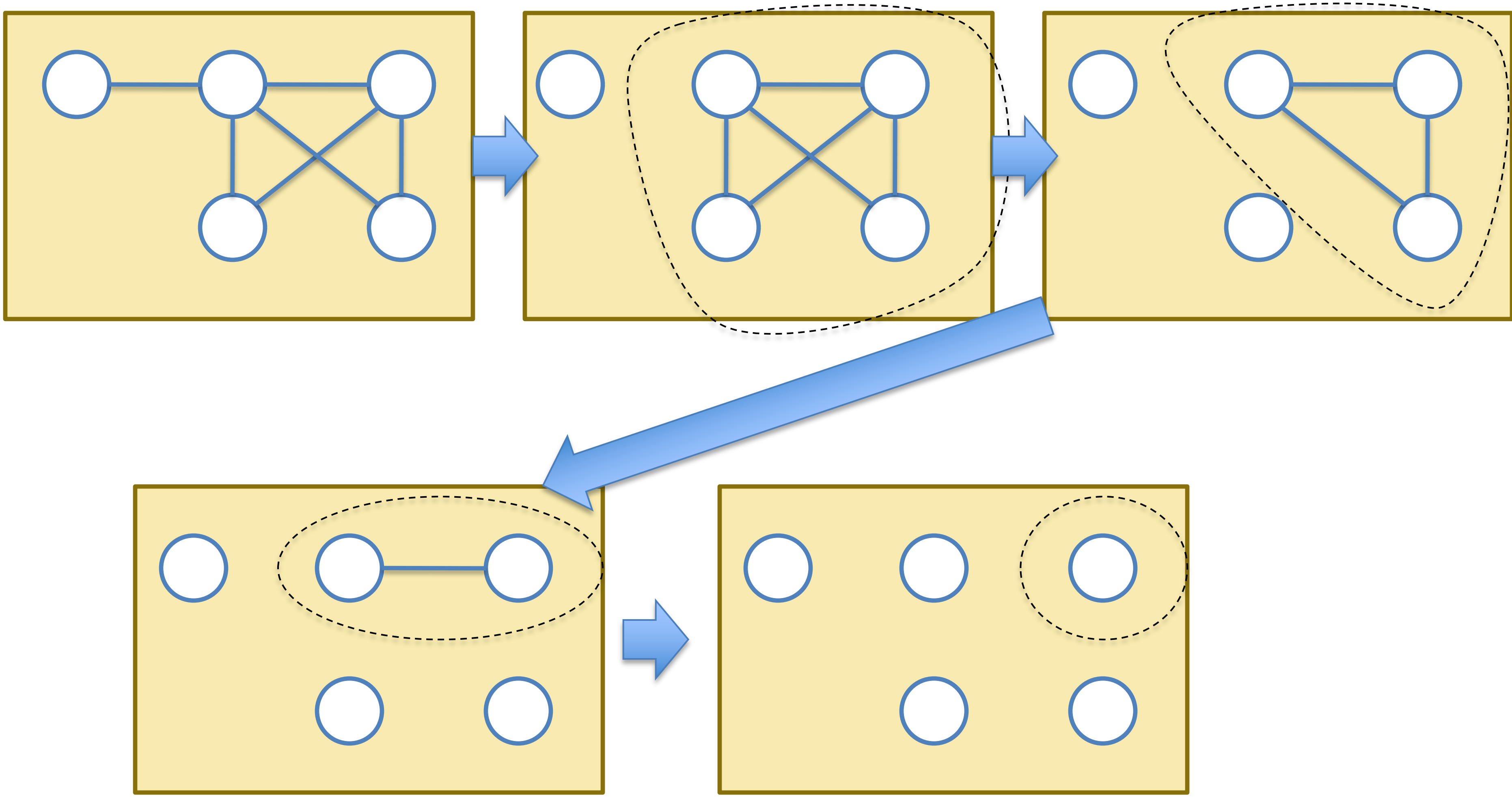
Coloring a Graph: Kempe's Algorithm

Kempe [1879] provides this algorithm for K-coloring a graph.

It's a recursive algorithm that works in three steps:

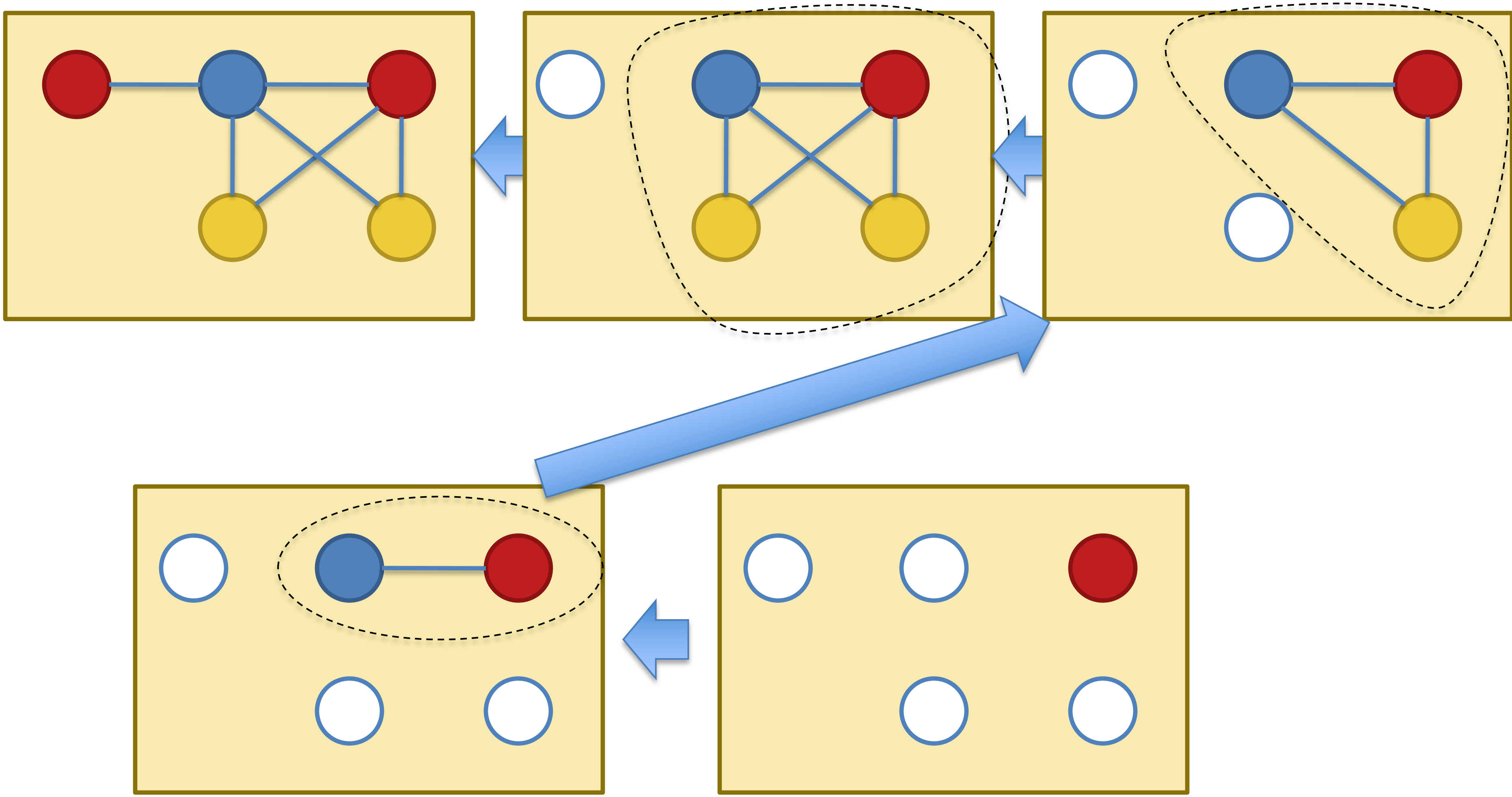
1. Find a node with degree < K and cut it out of the graph.
   - Remove the nodes and edges.
   - This is called *simplifying* the graph
2. Recursively K-color the remaining subgraph
3. When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was < K). Pick such a color.

Example: 3-color this Graph



Recursing Down the Simplified Graphs

Example: 3-color this Graph



Assigning Colors on the way back up.

Complete Register Allocation Algorithm

1. Build interference graph (precolor nodes as necessary).
   - Add move related edges
2. Reduce the graph (building a stack of nodes to color).
   1. Simplify the graph as much as possible without removing nodes that are move related (i.e. have a move-related neighbor). Remaining nodes are high degree or move-related.
   2. Coalesce move-related nodes using Brigg's or George's strategy.
   3. Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced.
   4. If no nodes can be coalesced *freeze* (remove) a move-related edge and keep trying to simplify/coalesce.
3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2.
4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack).
   1. If a node must be spilled, insert spill code as on slide 14 and rerun the whole register allocation algorithm starting at step 1.

WRAP UP

What we Learned

- Compilers material
  - Compiler architecture
  - Abstract syntax trees
  - Assembly code programming
  - SSA/LLVM
  - Lexing/Regex
  - Parsing/CFGs
  - Type systems
  - Program Analysis
  - Optimization
- Programming lessons
  - Basic functional programming – structural recursion/pattern matching
  - Debugging large pipelines
  - Testing programs with many edge cases
  - Programming with sophisticated specifications

Thank You

- Thanks to Eric and Tingting, Cyrus and Steven
- Tried out new curriculum this semester
  - Benefits and drawbacks
  - Next winter: synthesize this version with previous
- Please fill out teaching evaluations!