Mechanized semantics, third lecture

Advanced compilation: optimizations, static analyses, and their verification

Xavier Leroy

2019-12-19

Collège de France, chair of software sciences

Compiler optimizations

Automatically transform the code written the programmer in equivalent code that

- Runs faster
  - Eliminate useless or redundant computations
  - Use cheaper operations
  - Increase parallelism (instruction-level, threads).
- Is more compact
- Uses less energy
- Resists attacks better

Dozens of optimizations are known, each targeting a specific kind of inefficiency.

Some classic optimizations

Constant propagation:

```
a = 1;                   a = 1;
b = 2;          -->      b = 2;
c = a + b;               c = 3;
d = x - a;               d = x + (-1);
```

Dead code elimination:

```
a = 1;                   skip;
b = 2;          -->      b = 2;
c = 3;                   c = 3;
```

(if `a` is unused later)

Some classic optimizations

Common subexpression elimination:

```
c = a;                    c = a;
d = a + b;      -->       d = a + b;
e = c + b;                e = d;
```

Copy propagation:

```
e = d;                    skip;
f = d + 1;      -->       f = d + 1;
g = e * 2;                g = d * 2;
```

Some loop optimizations

Lifting loop-invariant computations:

```
for i = 1 to N do                    c = a + b;
  c = a + b;            -->          for i = 1 to N do
  x = x + c * A[i];                    x = x + c * A[i];
done                                 done
```

Induction variable elimination:

```
                                     a = p;
 for i = 1 to N do                   for i = 1 to N do
   a = p + i * 4;       -->            skip;
   ...                                 ...
                                       a = a + 4;
 done                                done
```

A cache optimization

To improve spatial locality of memory accesses.

Loop tiling (also called loop blocking):

```
for i = 0 to N-1 do          for i = 0 to N-1 step K do
  for j = 0 to N-1 do          for j = 0 to N-1 step K do
    a[i][j] = b[j][i]            for i2 = i to i+K-1 do
  done                             for j2 = j to j+K-1 do
done                    -->          a[i2][j2] = b[j2][i2]
                                   done
                                 done
                               done
                             done
```

Optimizations and static analyses

Some optimizations are unconditionally valid, such as

$$\begin{aligned} \texttt{x} * \texttt{2} &\rightarrow \texttt{x} + \texttt{x} \\ \texttt{x} * \texttt{4} &\rightarrow \texttt{x} << \texttt{2} \end{aligned}$$

Other optimizations apply only when certain conditions hold:

$$\begin{array}{rcll} \texttt{x} \,/\, \texttt{4} & \rightarrow & \texttt{x} >> \texttt{2} & \text{only if } \texttt{x} \geq 0 \\ \texttt{x} + \texttt{1} & \rightarrow & 1 & \text{only if } \texttt{x} = 0 \\ \texttt{if}\ \texttt{x} < \texttt{y}\ \texttt{then}\ c_1\ \texttt{else}\ c_2 & \rightarrow & c_1 & \text{only if } \texttt{x} < \texttt{y} \\ \texttt{x} := \texttt{y} + \texttt{1} & \rightarrow & \texttt{skip} & \text{only if } \texttt{x} \\ & & & \text{is unused later} \end{array}$$

We need a static analysis of the code before we can apply these optimizations.

Static analysis

Determine in advance ("statically") properties that hold for all possible executions of a program.

Often, these are properties of the values of variables at a given program point, such as

$$\mathrm{x} = n \qquad \mathrm{x} \in [n, m] \qquad \mathrm{x} = expr \qquad a.\mathrm{x} + b.\mathrm{y} \leq n$$

(x, y: program variables;
n, m, a, b constants determined by the static analysis.)

We're not talking about executing the program a few times!

- Program inputs are unknown.
- The analysis must terminate always.
- The analysis must take reasonable time and space.

Uses for static analysis

1960–2000: to improve performance.

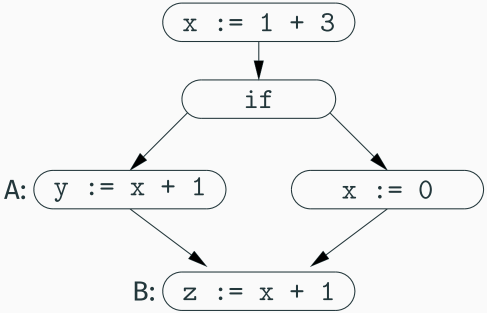
- Determine when an optimization applies.
- Guide code generation heuristics.

Since 2000: to improve safety.

- Guarantee the absence of certain run-time errors
  (for instance, out-of-bound array accesses).
- More modestly: warn about plausible programming errors.

The control flow / data flow approach

At the level of the control-flow graph, connect variable definitions with their uses.



Use point A, only one possible definition of `x`: `x = 4`.
Use point B, two incompatible definitions are possible: `x = 4` and `x = 0`.

Dataflow analysis

To each node n of the control-flow graph, associate a set of facts (such as "variable = constant"), connected by dataflow equations:

$$in(n) = \bigcap \{out(p) \mid p \text{ predecessor de } n\}$$
$$out(n) = gen(n) \cup (in(n) \setminus kill(n))$$

$in(n)$: facts true "before" executing n
$out(n)$: facts true "after" executing n
$kill(n)$: facts invalidated by the execution of n
(for example $x := a$ invalidates "$x = N$" for all N)
$gen(n)$: facts established by the execution of n
(for example $x := 1 + 2$ establishes "$x = 3$")

Dataflow analysis

$$\begin{aligned} in(n) &= \bigcap \{ out(p) \mid p \text{ predecessor de } n \} \\ out(n) &= gen(n) \cup (in(n) \setminus kill(n)) \end{aligned}$$

Solve these equations by fixed-point iteration.
($\approx$ Recompute $out(n)$ whenever one of the $out(p)$ changes.)

Extends from sets of facts to lattices (of finite height).

G. Kildall, *A unified approach to global program optimization*, POPL 1973.

J. B. Kam et J. D. Ullman, *Monotone data flow analysis frameworks*, Acta Informatica 1977.

Example: dead-code elimination via liveness analysis

Eliminate assignments $x := a$, replacing them by `skip`, if x is never used later in the program execution.

Example

```
x := 1;  y := y + 1;  x := 2
```

The assignment `x := 1` can be removed since `x` is not used before being redefined by `x := 2`.

This optimization builds on a static analysis called liveness analysis. It’s a backward dataflow analysis.

Liveness analysis

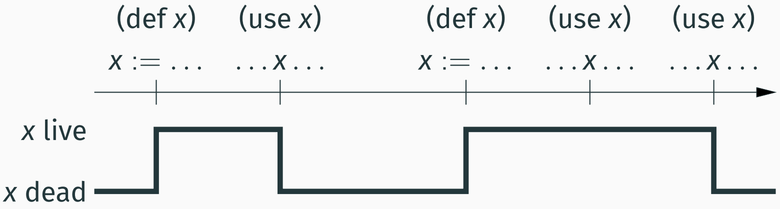
Liveness analysis

A variable is dead at a program point if its value is never used later in the program execution:

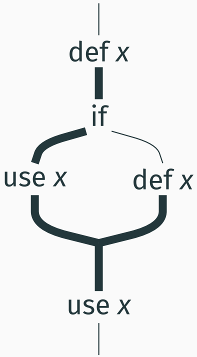
- the variable is not mentioned until the end of its scope;
- or it is redefined before every use.

A variable is live if it is not dead.

Easy to determine for straight-line code:



Liveness of a variable



For branches, we over-approximate liveness by assuming that `if` conditions can be true or false, and that `while` loops are executed 0 or several times.

Dataflow equations for liveness

Let L be the set of variables live “after” command c.
We define `live` $c\ L$, the set of variables live “before” c.

$$\texttt{live}\ \mathrm{SKIP}\ L = L$$

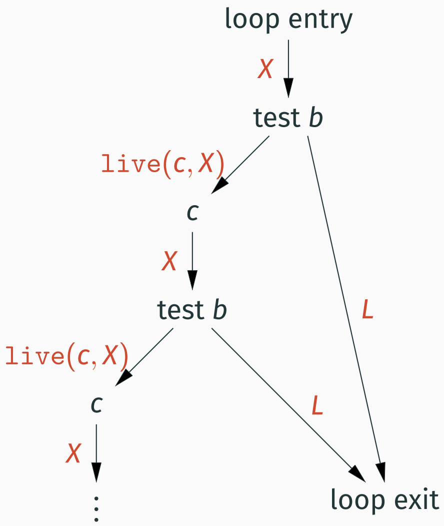
$$\texttt{live}\ (x := a)\ L = \begin{cases} (L \setminus \{x\}) \cup FV(a) & \text{si } x \in L; \\ L & \text{si } x \notin L. \end{cases}$$

$$\texttt{live}\ (c_1; c_2)\ L = \texttt{live}\ c_1\ (\texttt{live}\ c_2\ L)$$

$$\texttt{live}\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ L = FV(b) \cup \texttt{live}\ c_1\ L \cup \texttt{live}\ c_2\ L$$

$$\texttt{live}\ (\texttt{while}\ b\ \texttt{do}\ c)\ L = X \ \text{ such that } \ X = L \cup FV(b) \cup \texttt{live}\ c\ X$$

Liveness analysis for a loop



X must satisfy:

- $FV(b) \subseteq X$
  (to evaluate b safely)
- $L \subseteq X$
  (if b is false)
- $\texttt{live}(c, X) \subseteq X$
  (if b is true and c is run)

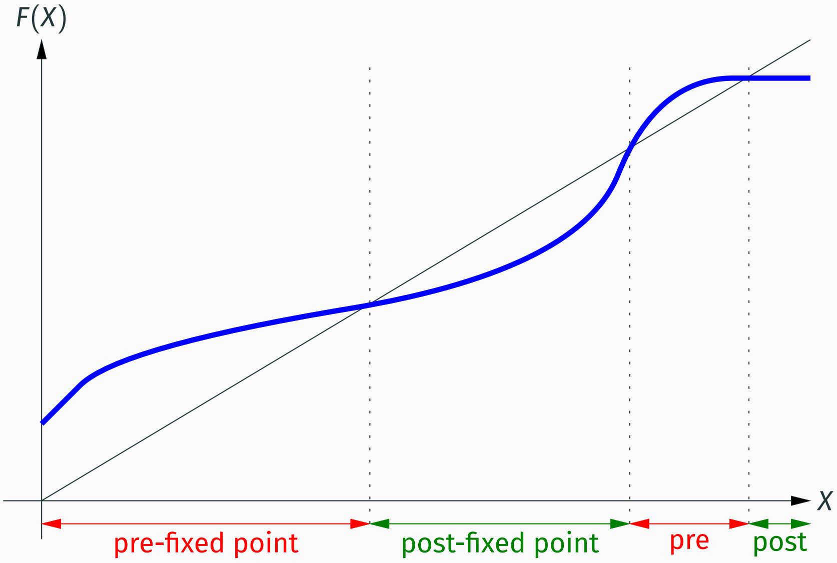
Fixed points

Consider $F = \lambda X.\, L \cup FV(b) \cup \texttt{live}(c, X)$.

To analyze the loop $\texttt{while}\ b\ \texttt{do}\ c$, we would like to compute a smallest fixed point de F, that is, a minimal X such that $F(X) = X$.

This is what makes the analysis maximally precise. For semantic correctness, it suffices to compute a post-fixed point of F, that is, any X such that $F(X) \subseteq X$.

Fixed points for a monotonically increasing function



The mathematician's approach

Theorem (Knaster-Tarski)

Let $A, \leq$ be a partially ordered type, and $F : A \to A$.

The sequence $\bot,\ F(\bot),\ F(F(\bot)),\ \ldots,\ F^n(\bot), \ldots$

converges to the smallest fixed point of F, provided that

- *F is increasing: $x \leq y \Rightarrow F(x) \leq F(y)$.*
- *$\bot$ is the smallest element of A.*
- *There are no infinite strictly-increasing sequences $x_0 < x_1 < \cdots < x_n < \cdots$*

Problems with the Knaster-Tarski approach

The condition about infinite strictly-increasing sequences is

1. difficult to mechanize and to use
   (well-founded orders + Noetherian induction);
2. often false! In the case of liveness analysis, the $\subset$ order has infinite strictly-increasing sequences:
   $\emptyset \subset \{x_1\} \subset \{x_1, x_2\} \subset \cdots$

Time for an alternate approach…

The engineer's approach

$$F = \lambda X.\, L \cup FV(b) \cup \mathtt{live}(c, X)$$

A bounded iteration:

- Compute $F(\emptyset), F(F(\emptyset)), \ldots, F^N(\emptyset)$ up to a fixed N.
- Stop as soon as a post-fixed point is found ($F^{i+1}(\emptyset) \subseteq F^i(\emptyset)$).
- Otherwise, return an over-approximation that is guaranteed to be a post-fixed point
  (in our example, $L \cup FV(\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done})$).

A compromise between analysis time and analysis precision.

Dead code elimination

Dead code elimination

The program transformation removes assignments to dead variables:

$x := a$ becomes `SKIP` if x not live “after” the assignment

Implemented as a function

$$\texttt{dce} : \texttt{com} \rightarrow \texttt{IdentSet.t} \rightarrow \texttt{com}$$

taking as second argument the set of variables live “after” the command, and updates this set during recursive calls.

(Implementation & examples in the Coq module `Optim`.)

Semantic interpretation of liveness

How can we characterize semantically that a variable x is live at a program point?

Hmmm…

Semantic interpretation of liveness

How can we characterize semantically that a variable x is live at a program point?

Hmmm…

How can we characterize semantically that a variable x is dead at a program point?

By the fact that the precise value of x at this point makes no difference to the program execution!

Liveness as an “hyper-property” of two executions

Consider two executions of the same command c in different initial stores:

$$c/s_1 \Downarrow s'_1 \qquad\qquad c/s_2 \Downarrow s'_2$$

Assume that the two initial stores agree on the variables $\texttt{live}\ c\ L$ that are live “before” c:

$$\forall x \in \texttt{live}\ c\ L,\ \ s_1(x) = s_2(x)$$

Then, both executions terminate on final stores that agree on the variables L live “after” c:

$$\forall x \in L,\ \ s'_1(x) = s'_2(x)$$

The `agree` relation and its properties

```
Definition agree (L: IdentSet.t) (s1 s2: store) : Prop :=
  forall x, IdentSet.In x L -> s1 x = s2 x.
```

The relation is decreasing with respect to the set `L`:

```
Lemma agree_mon:
  forall L L’ s1 s2,
  agree L’ s1 s2 -> IdentSet.Subset L L’ -> agree L s1 s2.
```

An expression evaluates to the same value in two stores that agree on the free variables of the expression:

```
Lemma aeval_agree:
  forall a s1 s2, agree (fv_aexp a) s1 s2 -> aeval a s1 = aeval a s2.
Lemma beval_agree:
  forall b s1 s2, agree (fv_bexp b) s1 s2 -> beval b s1 = beval b s2.
```

The `agree` relation and its properties

The relation is preserved by parallel assignment to a variable:

```
Lemma agree_update_live:
  forall s1 s2 L x v,
  agree (IdentSet.remove x L) s1 s2 ->
  agree L (update x v s1) (update x v s2).
```

The relation is preserved by unilateral assignment to a dead variable:

```
Lemma agree_update_dead:
  forall s1 s2 L x v,
  agree L s1 s2 -> ~IdentSet.In x L ->
  agree L (update x v s1) s2.
```

Semantic preservation

We prove that the execution of `dce c L` simulates the execution of c while preserving the `agree` relation between the stores.

$$\begin{tikzcd}
s \arrow[r, dash, "\texttt{agree}\ (\texttt{live}\ c\ L)"] \arrow[d, "\texttt{cexec}\ c"'] & s_1 \arrow[d, "\texttt{cexec}\ (\texttt{dce}\ c\ L)"] \\
s' \arrow[r, dash, "\texttt{agree}\ L"] & s_1'
\end{tikzcd}$$

```
Theorem dce_correct_terminating:
  forall s c s', cexec s c s' ->
  forall L s1,
  agree (live c L) s s1 ->
  exists s1', cexec s1 (dce c L) s1' /\ agree L s' s1'.
```

Semantic preservation

This result extends to diverging programs by proving a simulation diagram using the IMP reduction semantics. (Exercise.)

$$\begin{tikzcd}
c/s \arrow[r, dash, "\texttt{agree}\ (\texttt{live}\ c\ L)\ s\ s_1"] \arrow[d] & \texttt{dce}\ c\ L/s_1 \arrow[d, "1\ \text{ou}\ (0\ \text{et}\ |c'| < |c|)"] \\
c'/s' \arrow[r, dash, "\texttt{agree}\ (\texttt{live}\ c'\ L)\ s'\ s'_1"] & \texttt{dce}\ c'\ L/s'_2
\end{tikzcd}$$

Advanced topic:
Register allocation

Register allocation

Placing the variables used by the program (in unbounded numbers) in

- either processor registers
  (very fast access; small number of available registers)
- or memory locations (generally in the call stack)
  (available in great numbers; slower access).

Try to maximize the use of processor registers.

A crucial step for producing efficient machine code.

Two approaches to register allocation

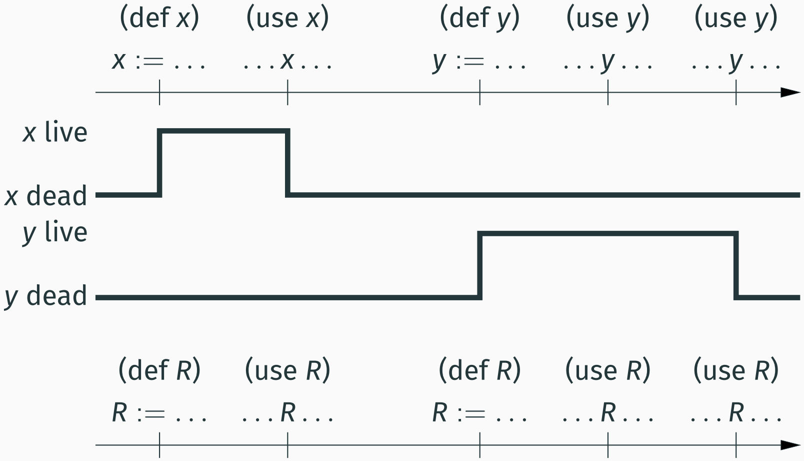
Naive approach: (one-to-one placement)

- Place the N most used variables in the N available registers.
- Place the remaining variables in memory.

Optimized approach (many-to-one placement)

- Place several variables in the same register, provided that these variables are never live at the same time.

Exemple de partage de registre



A register allocation for IMP

Simplified presentation: a transformation IMP $\rightarrow$ IMP that tries to minimize the number of different variables used.

The (non-injective) renaming of variables can be seen as a placement of variables in registers or stack locations.

The program transformation

Assume given a variable placement $f : \texttt{ident} \to \texttt{ident}$.

The program transformation consists in:

- Rename variables: x becomes $f\ x$.
- Eliminate dead code:

  $x := a \quad \longrightarrow \quad \texttt{SKIP}$ if x is dead "after"
- Eliminate redundant assignments:

  $x := y \quad \longrightarrow \quad \texttt{SKIP}$ if $f\ x = f\ y$

Correctness conditions on variable placements

Not all placements f preserve semantics!

Example

Assume $f\ \mathtt{x} = f\ \mathtt{y} = f\ \mathtt{z} = \mathtt{R}$

```
x := 1;                      R := 1;
y := 2;          ---->       R := 2;
z := x + y;                  R := R + R;
```

The transformed code puts 4 in `R` instead of 3...

What are sufficient conditions over f to preserve semantics?

We can discover them by reworking our proof of dead code elimination.

Revisiting the `agree` relation

```
Definition agree' (L: IdentSet.t) (s1 s2: store) : Prop :=
  forall x, IdentSet.In x L -> s1 x = s2 (f x).
```

An expression and its renaming evaluate to the same value in related memory stores:

```
Lemma aeval_agree':
  forall a s1 s2,
  agree' (fv_aexp a) s1 s2 -> aeval a s1 = aeval (rename_aexp a) s2.
Lemma beval_agree':
  forall b s1 s2,
  agree' (fv_bexp b) s1 s2 -> beval b s1 = beval (rename_bexp b) s2.
```

Revisiting the `agree` relation

Like before, the relation is preserved by unilateral assignment to a dead variable:

```
Lemma agree'_update_dead:
  forall s1 s2 L x v,
  agree' L s1 s2 -> ~IdentSet.In x L ->
  agree' L (update x v s1) s2.
```

Revisiting the `agree` relation

The relation is preserved by parallel assignments to a variable *x* and its renaming *f x*, provided *f* enjoys a non-interference condition (in red below).

```
Lemma agree'_update_live:
  forall s1 s2 L x v,
  agree' (IdentSet.remove x L) s1 s2 ->
  (forall z, IdentSet.In z L -> z <> x -> f z <> f x) ->
  agree' L (update x v s1) (update (f x) v s2).
```

A special case for variable copy

In the case of a variable copy $x := y$, the value assigned to x is not arbitrary: we know it is the value of y. This makes it possible to weaken the non-interference condition.

```
Lemma agree'_update_move:
  forall s1 s2 L x y,
  agree' (IdentSet.union (IdentSet.remove x L) (IdentSet.singleton y))
         s1 s2 ->
  (forall z, IdentSet.In z L -> z <> x -> z <> y -> f z <> f x) ->
  agree' L (update x (s1 y) s1) (update (f x) (s2 (f y)) s2).
```

This makes it possible to place x and y in the same register, even if x and y are simultaneously live.

The interference graph

The non-interference constraints $f\ x \neq f\ y$ can be materialized as an interference graph:

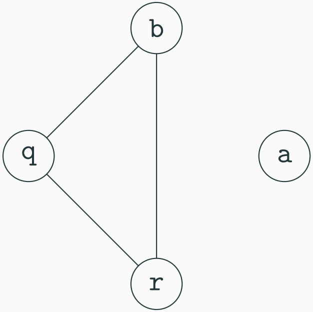
- Vertices = program variables.
- Non-oriented edge between *x* and *y* =
  x and *y* must not be placed in the same location.

Constructing the interference graph (Chaitin's algorithm):

- For each copy $x := y$, add an edge between *x* and every variable *z* live “after”, other than *x* and *y*.
- For each assignment $x := a$, add an edge between *x* and every variable *z* live “after”, other than *x*.

Example of interference graph

```
r := a;
q := 0;
while b <= r do
  r := r - b;
  q := q + 1
done
```



Register allocation by graph coloring

(G. Chaitin et al, *Register allocation via coloring*, 1981.)
(P. Briggs, *Register allocation via graph coloring*, 1992.)
(L. George et A. W. Appel, *Iterated register coalescing*, 1996.)

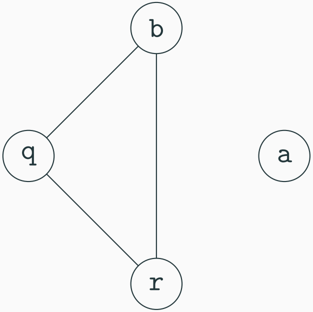
Color the vertices of the interference graph, using colors that are either registers or memory locations,

under the constraint that the two endpoints of an edge have different colors,

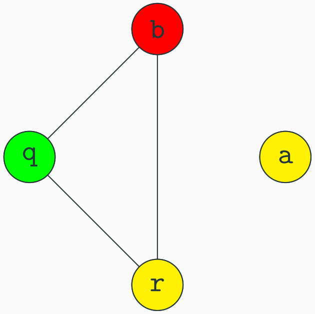
while maximizing the number of vertices colored with a register.

(A NP-complete problem in general, but good linear-time heuristics are known.)

Example of coloring



Example of coloring



```
yellow := yellow;
green := 0;
while red <= yellow do
  yellow := yellow - red;
  green := green + 1
done
```

What needs to be verified in Coq

Direct compiler verification:
formalize and verify a good graph coloring heuristics.

George and Appel's IRC algorithm $\approx$ 6 000 lines of Coq.

(Blazy, Robillard, Appel, ESOP 2010)

Validation a posteriori:
compute a placement using non-verified code;
validate that the placement satisfies the non-interference constraints, using a formally-verified validator;
abort compilation if the validator fails.

Validating a placement

It's easy to write a Boolean-valued Coq function

```
    correct_allocation:
              (ident -> ident) -> com -> IdentSet.t -> bool
```

that returns `true` if and only if the expected non-interference properties are satisfied.

(Or, in other words, if and only if the `ident -> ident` function is a correct coloring of the interference graph.)

Semantic preservation

The semantic preservation proofs for dead code elimination extend easily, with the extra hypothesis that `correct_allocation` returns `true`:

```
Theorem regalloc_correct_terminating:
  forall s c s’, cexec s c s’ ->
  forall L s1,
  agree’ (live c L) s s1 ->
  correct_allocation c L = true ->
  exists s1’, cexec s1 (regalloc c L) s1’ ∧ agree’ L s’ s1’.
```

Back to fixed points

Back to the mathematician's approach

Theorem (Knaster-Tarski)

Let $A, \leq$ be a partially ordered type, and $F : A \rightarrow A$.

The sequence $\bot,\ F(\bot),\ F(F(\bot)),\ \ldots,\ F^n(\bot), \ldots$

converges to the smallest fixed point of F, provided that

- *F is increasing: $x \leq y \Rightarrow F(x) \leq F(y)$.*
- *$\bot$ is the smallest element of A.*
- *There are no infinite strictly-increasing sequences $x_0 < x_1 < \cdots < x_n < \cdots$*

This theorem provides us with an effective algorithm to compute fixed points!
(See the Coq file `Fixpoints`).

Well-founded orders

A constructive reformulation of the condition “there are no infinite strictly-increasing sequences”:

All strictly-increasing sequences are finite.

In other words: the $>$ order is well founded.

In other words: $\forall x : A, \texttt{Acc}\ x$
where `Acc` is the accessibility predicate

```
Inductive Acc: A -> Prop :=
  | Acc_intro : (forall y:A, y > x -> Acc y) -> Acc x.
```

Noetherian induction and recursion

Proof by Noetherian induction:
to show $P(x)$, we can assume $P(y)$ true for all $y > x$.

($\approx$ structural induction on the derivation of `Acc` x.)

Programming by Noetherian recursion:
$F(x)$ can call $F(y)$ for one or several $y > x$.

Application: an algorithm to compute the smallest fixed point!
(See module `Fixpoints`.)

Application to liveness analysis

(See module `Optim`, section 3.4.)

1. To ensure $\subset$ is a well-founded order, we must restrict it to subsets of a finite universe `U`.

   ```
   Definition finset := { x: IdentSet.t | IdentSet.Subset x U }
   ```

2. We then get a fixed-point operation:

   ```
   finset_fixpoint:
     forall (F: finset -> finset), finset_monotone F -> finset
   ```

3. To be able to use it during liveness analysis, we need to simultaneously define the analysis function and prove that it is monotonically increasing.

   ```
   Program Fixpoint live'
         (c: com) (CONT: IdentSet.Subset (fv_com c) U)
       : { f: finset -> finset | monotone f } := ...
   ```

Summary

Summary

Static analyses:

- In this lecture: the dataflow approach.
- A variant: sparse analyses on the SSA intermediate form.
- In lecture #5: the abstract interpretation approach.

A crucial need: computing post-fixed points efficiently.

- In this lecture: bounded local iteration
  (one per loop).
- Classically, in compilers: global iteration on a control-flow graph.
- In lecture #5: widening to accelerate local iteration.

References

References

Optimizations and dataflow analyses:

- A. W. Appel, *Modern Compiler Implementation in Java / ML / C*, Cambridge University Press, 1998. Chapters 10, 11, 17, 18.
- For much more details: S. S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufman, 1997.

How to formally verify them:

- X. Leroy, *A formally verified compiler back-end*, J. Autom. Reasoning, 43(4), 2009.

A variant approach using the SSA form

- A. W. Appel, *op. cit.*, chapter 19.
- G. Barthe, D. Demange, D. Pichardie. *Formal verification of an SSA-based middle-end for CompCert*, ACM TOPLAS 36(1), 2014.