Aalto-yliopisto
Perustieteiden
korkeakoulu

Transformations and SSA

T-106.5450 Advanced Course on Compilers
Spring 2015 (III-V): Lecture 4

Vesa Hirvisalo
ESG/CSE/Aalto

2015-02-04

Today

- Transformations and analysis
  - what are optimizations done by compilers
  - what kind of analysis we do need?
- Value numbering
  - how we can find values that are equivalent
  - what is so difficult in static analysis of values?
- Static Single Assignment (SSA)
  - SSA and ϕ nodes
  - iterated dominance frontiers
- SSA construction
  - ϕ node explosion and dummy ϕ nodes
  - optimistic and pessimistic construction

Transformations and analyses

Languages

Compilers do several kinds of program transformations. For example,

- inter-language translations (from L_1 to L_2)
- intra-language translations (e.g., change *ops*)
- code re-structuring (e.g., code motion)
- code editions (additions, deletions)
- compile-time executions (e.g., constant folding)

Analyses are typically *purpose-specific*

- this a result of approximation
- usually, we *must* design an analysis based on its usage

There exists, however, many example analyses that can be used as basis.

Analysis

Transformation alter the program, at least its syntactical form.

Note that whole programs are rarely compiled, optimized, etc. Typical targets are subroutines or groups of subroutines.

- typically, transformation alter the *compilation unit* semantics in a conservative way
- compiler often over-estimate (are prepared for sitations that cannot happen)
- are not aware of the whole program or underlying machine (risk of under-estimation)
  - especially true for parallel programs

Note that while conserving the sematics of a compilation unit, the semantics of its parts often do change.

Optimizations

Optimizations are transformations that improve code.

- program analysis yield information
- there are several possible transformations
  - select best: *search*
  - large search space: *approximate*
  - conflicts: *prioritize*

The problem is typically converted into an abstract form, where the optimization is done (or sub-optimization).

- several techniques
- searching for max (min) value
- special techniques like Dynamic Programming (DP), Integer Linear Programming (ILP), ...

Values

- computation
  - basically computing values
- compiling
  - basically: making a native program that computes values
  - this is about getting instructions and data "hard wired"
  - to make transformations we need to know what is computed in the first place
- optimization
  - with minimum resources
  - instruction dependencies and the amount of data are essential
- the required analysis
  - how values are linked
  - the data flow

Typical analysis

It is important to understand and identify values.

- Bound (approximation)
  - must analysis (Greatest Lower Bound, GLB)
  - may analysis (Lest Upper Bound, LUB)
- Direction of analysis
  - analysis goes forward, data points backward
    - defined variables (names)
    - reaching definitions (names and definition positions)
    - ud chains (pointers from uses to definitions)
  - analysis goes forward, data points backward
    - live variables (names)
    - next uses (names and usage positions)
    - du chains (pointers from definitions to uses)

Value numbering

Equivalent Values

Two expressions are semantically equivalent, *iff*

- they compute the same value
- would be very useful, but is not decidable (in theory)
- and also, *in practice* rarely possible to analyze

Two expressions are syntactically equivalent, *iff*

- the operator and the operands are the same

Generalization: equivalence using algebraic identities

- two expressions are congruent, *iff*
  - they are syntactically equivalent or algebraic identical

However, computing useful information such will not be simple

Directed Acyclic Graph (DAG)

Basic blocks can be represented as DAGs. DAG is good representation for both analysis and transformations.

- basically one DAG per BB (basic block)
- leaf nodes are variables or constants
- inner nodes are operators
- the edges represent
  - the order dependencies
  - the operands
- as a whole, partial order
  - extra (non-operand) edges can be used to limit the possible orders

Local value numbering

One method for analysis of bindings is value numbering.

- actually congruences used instead equivalences

To contruct a DAG, go through a block, and

- create a new node for each value, except
- add new label to an old node, if value already computed
  - note that x = y does not create a new node
- an edge from the operator to all operands
- add also side effect edges
- expressions can be killed
  - expr is killed if any of it operands is re-defined

Note: See textbook Alg. 6.3. The textbook handles global value numbering and its relationship to SSA superficially.

Global value numbering

Local analysis is not enough

- we should know value congruences between basic blocks

Basically, we could

- solve locally (i.e., DAGs for basic blocks)
- iterate globally (and initial values for BBs)

However, there are problems

- alias/Points-to problem: Addresses not exactly computable
  - in general: not decidable
- meets in control flow
- alternative definitions $\Rightarrow$ this can lead to an explosion of ud pointers

Static Single Assignment (SSA)

The problem with bindings

Basic CFG seems nice

- intuitive, easy to construct

However, transformation are not so easy

- additions, deletions, moving code

Often the problem is in *bindings* (e.g., what is the value referred to by a variable).

- having a *single* binding (assignment) makes transforming easier (and also analyses)
- construction such a representation is not trivial.

In the following, we use ϕ notation for the single static assignments.

SSA basics

Using Static Single Assignment form (SSA) a common solution to maintain bindings during transformations. It can be formulated in several ways.

Typical textbook definition

- subscripting variables
- using the ϕ function

Analysis-based view

- explicit ud-chaining

Construction-based view

- using ϕ nodes to represent values

SSA properties

A representation having single static assignments has several properties:

1. Operations of BBs can form DAGs
2. Input and output of ϕ are of the same type
3. The i^{th} operand of a ϕ-operation is available at the end of the i^{th} predecessor.
4. If operation x in BB_x defines an operand of operation y in BB_y then there is a path $BB_y \rightarrow^+ BB_x$. (Note loops!)
5. Let A, B be blocks with a definition of x reaching to a use of x in block C. Let D be the first common block of execution paths $A \rightarrow^+ C$, $B \rightarrow^+ C$, then D contains a ϕ operation for x.

Phi-node positioning

Note especially the last property. It indicates how SSA actually works, and how SSA can be constructed.

The concept of SSA is actually very simple, but the related analysis is not.

The last SSA property also indicates, where to position ϕ functions.

- note that ϕ-functions cause definitions
  - iteration will be needed
- bindings must be understood
  - a value can span over several uses and defs

Dominance frontiers

The basic tool is defined by *dominance frontiers*.

Dominance Frontier $DF(n)$ is informally the set of nodes almost dominated by n:

- the dominance frontier of a node n is the set of nodes m such that n dominates some predecessor of m but not all

More formally

$$DF(n) = \{m \mid n \not> m \ \wedge \ \exists p \in pred(m) : n \geq p\}$$

where the order is the domination order.

Iterated dominance frontiers

Dominance frontier of a set M of nodes

- $DF(M) = \bigcup_{n \in M} DF(n)$

What about control flow?

- control joins and something is not dominated
- the phi-node position?
- gets complicated

Iterated dominance frontiers (DF^+)

- we define it to be the minimum fix-point
  $DF_0 = DF(M)$
  $DF_{i+1} = DF(DF_i)$

note: having the definition does not make computing the fix point trivial!

SSA construction

Simple DF-based approach

A straight-forward way to construct SSA

- split used variables at branches
- propagate changes

This can lead to an explosion. Why?

- Program can contain $O(n)$ variables.
  - $O(n)$ ϕ functions per variable
  - complexity $O(n^2)$!

Many of the ϕ-functions are dummy

- eliminate dummy ϕ-functions
- we will waste a lot of space

In practice, things are no so bad.

Construction algorithm

Informal description.

- Perform local value numbering
- For any used variable that is used but not locally defined compute set of definition points
- Compute *iterated* (i.e., minumum fix point) dominance frontiers of definition points
- Insert ϕ function and rename variables accordingly

There are several ways to construct SSA.

- the result differs
- the complexity differs
- the applicability differs

Equivalence-based construction

Two basic approaches

- pessimistic construction
  - assume all values not-equivalent
- optimistic construction
  - assume all value equivalent

E.g., for the optimistic approach

- Values are different constants
- Values are generated form syntactical different operations
- Values are generated form syntactical equivalent operations with proven different values as operands

The optimistic approach needs plenty of support

- iteratively we can alternate between the two

A semi-formal example

Computing live values

Let *live values* be the values that are going to be used after a program point. Values are labeled using variables with subscriptions. For example, let x and y share the same value (let us call that V_1) before the instructions

```
x = x + y
if x > 0 goto L
```

Then, we have three nodes and two edges in our DAG:

- V_1: <operator: ""; labels: {x_1, y_1}; operands: {}>
- V_2: <operator: "+"; labels: {x_2}; operands: {V_1, V_1}>
- V_3: <operator: "integer-0", labels: {}; operands: {}>

Note: Here we use the empty operator (i.e., "") with no operands as we do not how x and y got their previous value, i.e., V_1 is a leaf node. Further, the constant 0 in the condition has a node of its own.

A piece of code

Consider *may* liveness analysis of the following code (assuming that the values of a, b, and t live at the end of the code):

```
(S1)      i = 0
(S2) L1:  if i >= N goto L2
(S3)      j = 4 * i
(S4)      t = a[j]
(S5)      i = i + 1
(S6)      j = 8 * i
(S7)      b[j] = t
(S8)      goto L1
(S9) L2:  a[k] = 1
(S10)     t = a[i]
```

Note that all names except the labels L1 and L2 are variables holding a value (i.e., scalar values or array pointers to memory).

Understanding the values

It is rather easy to see that the code

- uses constants 0, 1, 4, and 8
- defines values of i (in S1, S5), j (S3, S6) and t (S4, S10)
- uses the undefined N (in S2), a (S4), b (S7), k and l (S9)

We know that $i_1 = 0$ and do not know if $a_1 = b_1$ (are the arrays aliases?), thus the code seems to handle 14 static values.

However, we have two more static values

- the array assignments modifying the array(s) (in S7, S9)

Note that it is unclear if S9 and S10 handle the same array location (the array is the same, but what about the index?).

- several alias problems, we approximate them unaliased
- as we are doing a *may* liveness analysis

The SSA-based on dominance frontiers (DF)

The code consists of four basic blocks

- B1: {S1}, B2: {S2}, B3: {S3–S8}, B1: {S9,S10}

The dominator of these are

- D(B1) = {B1}, D(B2) = {B1,B2}, D(B3) = {B1,B2,B3}, D(B4) = {B1,B2,B4}

and there is a natural loop L(B3,B2) = {B2,B3}.

Other inter-block values are unique (i.e., single assingment), but i (used in S3, S5) has two definitions (in S1, S5)

- we solve this by placing a phi-node for i in B2

B2 is the correct place as $DF^{+}(B1) = DF^{+}(B3) = \{B2\}$. Thus including the phi-node, we have 17 value global nodes.

The value nodes

B1: V_1: <operator: "integer-0"; labels: {i_1}; operands: {}>
B2: V_2: <operator: "phi"; labels: {i_2}; operands: {V_1, V_9}>
B2: V_3: <operator: "var"; labels: {N_1}; operands: {}>
B3: V_4: <operator: "integer-4"; labels: {}; operands: {}>
B3: V_5: <operator: "*"; labels: {j_1}; operands: {V_4, V_2}>
B3: V_6: <operator: "var"; labels: {a_1}; operands: {}>
B3: V_7: <operator: "[]"; labels: {t_1}; operands: {V_6, V_5}>
B3: V_8: <operator: "integer-1"; labels: {}; operands: {}>
B3: V_9: <operator: "+"; labels: {i_3}; operands: {V_4, V_8}>
B3: V_{10}: <operator: "integer-8"; labels: {}; operands: {}>
B3: V_{11}: <operator: "*"; labels: {j_2}; operands: {V_{10}, V_9}>
B3: V_{12}: <operator: "var"; labels: {b_1}; operands: {}>
B3: V_{13}: <operator: "[]="; labels: {}; operands: {V_{12}, V_{11}, V_7}>
B4: V_{14}: <operator: "var"; labels: {l_1}; operands: {}>
B4: V_{15}: <operator: "var"; labels: {k_1}; operands: {}>
B4: V_{16}: <operator: "[]="; labels: {}; operands: {V_6, V_{15}, V_{14}}>
B4: V_{17}: <operator: "[]"; labels: {t_2}; operands: {V_6, V_2}, V_{16}[1]>

[1] Because of the potential alias, we add a non-functional DAG edge

Liveness dataflow equations

Note that in the previous, we did not exactly specify how we use the memory (which is important in real compilers). But after this, we can operate solely with the abstract values.

We need *backward may* analysis:

out[B] = $\cup_{s \in succ(B)}$ in[s]
in[B] = gen[B] $\cup$ (out[B] - kill[B])
$\perp_B = \{\}$

We are interested on usage of *created* values and V_1, V_4, V_8, V_{10} are constants. Thus, the gen sets (using a value created elsewhere) and kill sets (creating a value used elsewhere) for the blocks are:

B1:	gen = {}	kill = {}
B2:	gen = {V_3}	kill = {V_2}
B3:	gen = {V_2, V_6, V_{12}}	kill = {V_7, V_9, V_{11}}
B4:	gen = {V_6, V_{14}, V_{15}}	kill = {V_{17}}

The global solution

Below is the iteration using the order B4, B2, B3, B1:

	I1	I2	I3
out[B4]	V_6, V_{12}, V_{17}	fixed (our assumption: a,b,t)	fixed
in[B4]	$V_6, V_{12}, V_{14}, V_{15}$	no change	no change
out[B2]	$V_6, V_{12}, V_{14}, V_{15}$	$V_2, V_3, V_6, V_{12}, V_{14}, V_{15}$	no change
in[B2]	$V_3, V_6, V_{12}, V_{14}, V_{15}$	no change	no change
out[B3]	$V_3, V_6, V_{12}, V_{14}, V_{15}$	no change	no change
in[B3]	$V_2, V_3, V_6, V_{12}, V_{14}, V_{15}$	no change	no change
out[B1]	$V_3, V_6, V_{12}, V_{14}, V_{15}$	no change	no change
in[B1]	$V_3, V_6, V_{12}, V_{14}, V_{15}$	no change	no change

As there are no more changes, the iteration has terminated (we have a fix-point solution)

- For some other iteration orders, the iteration is longer!
- A depth-first-search exit-order (in the direction of analysis) is usually a good choice

Conclusion

From the result, it is easy to see what values need to be represented at each program point.

- The local solution is implied by the DAGs
- Usually, this analysis is used for register allocation
  - The choice of evaluation order of the DAGs affects the allocation
- From the solution we can see, e.g., that
  - V_7 (value t) can be discarded immediately after its use in B3, as it is not live at out[B3]
  - As previously assumed V_3, V_6, V_{12}, V_{14}, V_{15} (that is: n, a, b, l, and k) must get their values before the code is executed as they are live at in[B1]

Note that we omitted the dataflow equations (and their solution) for the global value numbering and dominance frontier.