Outline

- Introduction to SSA, Construction, Destruction
- Analyses and Optimizations
  - Classic analyses and optimizations on SSA representations
  - Context-sensitive and Inter-procedural analyses and optimizations
  - Heap analyses and optimizations

Analyses and Optimizations

- Analyses to safely perform optimizations
- Cost model: runtime of a program
  - statically only conservative approximations
    - Loop iterations
    - Conditional code
  - Even for linear code not known in advance:
    - Instruction scheduling
    - Cache access is data dependent
    - Instruction pipelining: execution time is not the sum of individual operations costs
- Alternative cost model: memory size, power consumptions
- Caution: cost of a program ≠ sum of costs of its elements

Optimization: Implementation

- Legal transformations in SSA-Graphs:
  - Simplifying transformations reduce the costs of a program
  - Preparative transformations allow the application of simplifying transformations
- Using
  - Algebraic Identities (e.g. Associative / Distributive law for certain operations)
  - Moving of operations
  - Reduction of dependencies
- Optimization is a sequence of goal directed, legal simplifying and legal preparative transformations
- Legibility proven
  - Locally by checking preconditions
  - Due to static data-flow analyses

Algebraic Identity: Elimination of Operations and its Inverse

- x ⊙ y
- No side effects in $\tau, \tau^{-1}$
Graph Rewrite Schema

SSA-subgraph before Transformation

SSA-subgraph after Transformation

No side effects in $\tau$, $\tau^{-1}$

Pr condition established by local check preparative analyses

Elimination of Memory Operation and its Inverse

Elimination of Duplicated Memory Operations

Elimination of non-essential dependencies

Op1, Op2 memory operations (store, call)
u2, d1, d2 designate may Use/Define
Algebraic Identity: Invariant Compares

Associative Law

Distributive Law

Operator Simplification
Constant Folding

\[ \text{Const} \ x \quad \text{Const} \ y \quad \Rightarrow \quad \text{Const} \ \tau \ x \ y \]

Evaluation using source algebra or target algebra (if allowed by source language)

Constant folding over \( \phi \)-functions

\[ \begin{align*} \text{Const} & \quad \text{Const} \\ \ x & \quad \ y \\ \phi & \quad \tau \ x \ z \\ \tau \ y \ z & \quad \phi \end{align*} \]

General: Moving arithmetic operations over \( \phi \)-functions

\[ \begin{align*} x & \quad y \quad z \\ \phi & \quad \tau \\ \tau & \quad \tau \end{align*} \]

No side effects in \( \tau \) (no call, store)

Optimizations

- Strength reduction:
  - Bauer & Samelson 1959
  - Replace expensive by cheap operations
    - Loops contain multiplications with iteration variable,
    - These operations could be replaced by add operations (Induction analysis)
  - One of the oldest optimizations: already in Fortran I-compiler (1954/55) and Algol 58/60- compiler

- Partial redundancy elimination (PRE):
  - Morel & Renvoise 1978
  - Eliminate partially redundant computations
    - SSA eliminates all static but not dynamic redundancies
    - Problem on SSA: which is the best block to perform the computation
    - Move loop invariant computations out of loops, into conditionals
  - subsumes a number of simpler optimization
Example: Strength reduction

```c
for (i=0; i<n; i++){
    for (j=0; j<n; j++){
        b[i,j] = a[i,j];
    }
}
```

```
// Original Address Computation:
> a[i,j] = > a[0,0] + i*n*d + j*d
```

Induction Analysis: Idea

- Find induction variable $i$ for a loop:
  - $i$ is induction variable if in loop only assignments of form $i := i + c$ with loop constants $c$
    or, recursively, $i := c*i + c'$ with $i'$ induction variable
  - $c$ loop constant: $c$ does not change value in loop, i.e.
    - $c$ is static constant,
    - $c$ computed in enclosing loop
- Transformation goal: values should grow linearly with iteration.
- Transformation:
  - Let $i_0$ initialization of $i$ and induction variables, $i := i + c$ and $i' := c*i + c''$
  - New variable $ia$ initialized $ia = c' * i_0 + c''$
  - At loop end insert $ia = ia + c' * c$
  - Replace consistently operands $i'$ by $ia$
  - Remove all assignments to $i, i'$ and $i, i'$ themselves if $i$ is not used elsewhere

Induction Analysis: Implementation

- Assume initially: all variables are induction variables
- Finding induction variable $i$ for a loop follows definition
- Iteratively until fix point: $i$ is not induction variable if not:
  - $i := i + c$ with loop constants $c$ (direct induction variable)
  - $i := c*i + c'$ with $i'$ induction variable and loop constants $c, c'$ (indirect induction variable)
  - $i := \phi(i_1 \ldots i_n)$ with $i_1$ being direct induction variable
- On SSA simplification possible
  - any loop variable corresponds to a cyclic subgraph
  - Find Strongly Connected Component (SCC) and check those for induction variable condition

```
sum(array[int] a)
{
    s = 0;
    for(i=0; i<100; i++){
        s = a[i] + s;
    }
    return s;
}
```
Induction Variables

Induction Variables (Schematic)

Direct Induction Variable Cycle

Move * over φ-function

Conditional jump

Integrate s
Move Addition

Associative Law

Change Compare
Partial Redundancy Elimination: Idea

- SSA is representation
  - without (provable) static redundancies
  - with all dependencies explicit
- Question which block should contain the computation guaranteeing
  - that the result is used on all path to the end
  - that the computation is not repeatedly performed in loops
- First idea: compute each operation earliest (as soon as all arguments are available)
- Observation:
  - Fast introduction of many live values: high register pressure
  - Many execution path compute but do not use a certain value
- Solution is Partial Redundancy Elimination (PRE):
  - Delay computation until it is used on all paths
  - In practice: move them out of loops into conditional code

Observations on SSA

Operations that must be executed in original block:
1. \(\phi\)-nodes,
2. Computations with exceptions
3. Jumps
4. “pinned” operations (postponed)

1. Observation:
   All other nodes could be computed in other blocks as well iff data dependencies are obeyed.

2. Observation:
   No statically redundant computation at all, i.e., one important goal of optimization immediately follows from the representation. Dynamic redundancy remains a problem.
Example: Initial Situation

Immature $\phi'$

Mature $\phi' \rightarrow \phi$

Placement of computations
Placing $t_1$ earliest

$t_1$ not needed on many paths.

Placing $t_1$ latest

$t_1$ (re-)computed in each iteration.

Placing $t_1$ “optimally”
According of Knoop, Steffen

Still $t_1$ (re-)computed in each iteration.

Insert Blocks

Virtually, insert empty blocks, to capture operations executed only on a specific path.
Placing for $t_1$ out of loops into conditional code

However: Exist path $t_1$ not used on.

PRE: Discussion

- Placing earliest
  - Advantage: short code, could be fast code because of instruction cache; no unnecessary computations in loops
  - Disadvantage: many paths do not need result, high register pressure
- Placing lazily
  - Advantage: computation needed on all paths
  - Disadvantage: unnecessary computations in loops
- Placing out of loops into conditional code
  - Advantage: no unnecessary computations in loops
  - Disadvantage: some unnecessary computations in general as some paths do not need result

PRE: Implementation

- Find partially (dynamically) redundant computations
  - $B$ contains operation $\tau$ computing $t$, $B'$ contains $\tau'$ consuming $t$
  - If $B'$ post-dominates $B$ ($B' \leq B$) no dynamic redundancy
  - Assume all operations as dynamically redundant
  - For each operation $\tau$ computing $t$ and the set of operations $\tau_1, \ldots, \tau_n$ consuming $t$: if $B(\tau'_k) \leq B(\tau)$ for some $k$ in $1 \ldots n$ then $\tau$ is not placed partially redundant – all others are (!)
- Eliminate partially redundant computations
  - Compute earliest position (all arguments available, all uses dominated) for each partially redundant operation $\tau$
  - Move copies of $\tau$ towards the consuming operations $\tau_1, \ldots, \tau_n$ along the dominator tree until no dynamic redundancies but stop at loop heads
  - Not deterministic but that does not matter (!)

PRE: “Pinned” Operations

- Placement sometimes only possible if it is the last transformation on SSA
  - Same computation computed several times
  - Further optimizations recognize this wanted static redundancy
- Solution:
  - Let $t_1 : a_1 + b_1$ and $t_2 : a_1 + b_1$ semantic equivalent computations at different positions (blocks)
  - Replace $+$ by a „pinned“ $\oplus_{\text{Block}}$
  - Thereby $\oplus$ operation additionally depends on the current block as new arguments
  - computations $a_1 \oplus b_1$ and $a_1 \oplus b_1$ not congruent any more
Further Optimizations

- Constant evaluation (simple transformation rule)
- Constant propagation (iterative application of that rule)
- Copy propagation (on SSA construction)
- Dead code elimination (on SSA construction)
- Common subexpression elimination (on SSA construction)
- Specialization of basic blocks, procedures, i.e. cloning
- Procedure inlining
- Control flow simplifications
- Loop transformations (Splitting/merging/unrolling)
- Bound check eliminations
- Cache optimizations (array access, object layout)
- Parallelization
- …

Observations

- Order of optimizations matters in theory:
  - Application of one optimization might destroy precondition of another
  - Optimization can ruin the effects of the previous one
- Optimal order unclear (in scientific papers usual statements like: “Assume my optimization is the last …”)
- Simultaneous optimization too complex
- Usually first optimization gives 15% sum of remaining 5%, independent of the chosen optimizations
- Might differ in certain application domains, e.g. in numerical applications operator simplification gives factor >2, cache optimization factor 2-5

Outline

- Introduction to SSA, Construction, Destruction
- Analyses and Optimizations
  - Classic analyses and optimizations on SSA representations
  - Context-sensitive and Inter-procedural analyses and optimizations
  - Heap analyses and optimizations

Static Analyses

- Flow-insensitive vs. flow-sensitive
  - Abstraction from the direction of flow or not
  - We will stick to flow-sensitive analyses, flow-insensitive are proposed in analyses for comprehension, reengineering large systems etc.
- Contexts-insensitive vs. context-sensitive
  - Abstraction from the execution history (call context) or not
  - We will derive context-sensitive from contexts-insensitive
- Intra-procedural vs. inter-procedural
  - Consider each procedure separately or analyze the whole program
  - We will introduce both
Context-insensitive Data flow analysis

\[ x + y = \{2,3,4\} \]
\[ a + b = \{4,5,6\} \]
\[ 2(x+y) \geq a+b? \]

Context-sensitive Data flow analysis

\[
\begin{array}{cccccccc}
 & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
 x & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 \\
y & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\
a & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 \\
b & 3 & 3 & 4 & 4 & 3 & 3 & 4 & 4 \\
\end{array}
\]

Representation of context in decision tables
Too large! Too much redundancy!

Concept of \( \chi \) terms
Advantages of $\chi$-Terms

- Compact representation of context sensitive information
- Delayed widening (abstract interpretation) of terms until no more memory: no unnecessary loss of information
- Each SSA node type $n$ has a concrete semantics $\nu(n)$.
- Delayed widening (abstract interpretation) of terms until no more memory.
- On execution of $\nu(n)$, map inputs $i$ to outputs $o$ and $o = \nu(n)(i)$.
- No unnecessary loss of information.
- Inputs $i$ and outputs $o$ are records of typed values (P1).
- $\chi$-Terms are implementation of decision diagrams.
- Abstract semantics $T_n$ is called transfer function of node type $n$.
- On analysis of $\nu(n)$, map abstract analysis inputs $\nu(i)$ to abstract analysis outputs $\nu(o)$ and $\nu(o) = T_n(\nu(i))$.
- Abstract semantics $T_n$ is a lattice (actually only a complete partial order).

DFA on SSA (cont.)

- Provided that
  - Transfer functions are monotone: $x \leq y \Rightarrow T_n(x) \leq T_n(y)$ with $x,y \in \bigodot_A$ and $\leq$ defined element-wise with $\subseteq$ of the respective abstract type lattices.
  - Abstract type lattices are finite (sufficient).
- Following iteration terminates in a unique fixed point
  - Initialize the input of each node the SSA graph with the smallest (bottom) element of the lattice corresponding its abstract type.
  - Initialize the start nodes of the SSA graph with proper abstract values of the corresponding its abstract types.
  - Attach each node with its corresponding transfer function.
  - Uniform randomly compute abstract output values.
  - (Fewer updates use SCC and interval analysis to determine an traversal strategy, backward problems analyzed analogously.)

Data-Flow Analyses (DFA) on SSA

- Each SSA node type $n$ has a concrete semantics $\nu(n)$.
  - On execution of $\nu(n)$, map inputs $i$ to outputs $o$ and $o = \nu(n)(i)$.
  - Inputs $i$ and outputs $o$ are records of typed values (P1).
  - $\nu(n) : \text{type}(\nu(i)) \rightarrow \text{type}(\nu(o))$.
- Each static data-flow analysis abstracts from concrete semantics and values.
  - Abstract semantics $T_n$ is called transfer function of node type $n$.
  - On analysis of $\nu(n)$, map abstract analysis inputs $\nu(i)$ to abstract analysis outputs $\nu(o)$ and $\nu(o) = T_n(\nu(i))$.
  - $T_n : \text{type}(\nu(i)) \rightarrow \text{type}(\nu(o))$.
- For each abstract type $A = \text{type}(\nu(\bullet))$ there is a partial order relation $\subseteq$ and a meet operation $\cup$.
  - $\subseteq : A \times A$
  - $\cup : A \times A \rightarrow A$ with $\cup = T_n$
  - $(A, \subseteq)$ is a lattice (actually only a complete partial order).

Generalization to $\chi$-terms

- Given such a context-insensitive analysis (lattices for abstract values, set of transfer functions, initialization of start node) we can systematically construct a context-sensitive analysis.
- $\chi$-term algebras $\Xi$ over abstract value $a \in A$ introduced.
  - $a \in A \Rightarrow a \in \Xi$.
  - $t_1, t_2 \in \Xi \Rightarrow \chi(t_1, t_2) \in \Xi$.
  - Induces sensitive lattices for abstract values $(\Xi, \subseteq)$ and for $a_1, a_2 \in A$ and $t_1, t_2, t_3, t_4 \in \Xi$:
    - $a_1 \subseteq a_2 \Rightarrow a_1 \subseteq a_2$,
    - $\chi(a_1, a_2) \subseteq a_1 \cup a_2$,
    - $t_1 \subseteq t_3, t_2 \subseteq t_4 \Rightarrow \chi(t_1, t_2) \subseteq \chi(t_3, t_4)$.
- New transfer functions induced.
New transfer functions

- **ϕ-node’s transfer functions:**
  - Insensitive: $T_\phi = \bigcup a(\phi_1) \cup \ldots \cup a(\phi_k)$
  - Sensitive: $S_\phi = \bigcup a'(\phi_1 \cup \phi_2 \cup \ldots \cup \phi_k)$ with $b$ block number of φ-node and for $a_1, a_2 \in A$ and $t_1, t_2, t_3, t_4 \in X$:
    $a_1 \cup a_2 = \chi_b(a_1, a_2)$
    $\chi(x, y) = \chi_b(t_1 \cup t_2, t_3 \cup t_4)$ iff $x = b$ (cases $y = b$ analog)
    $\chi(x, y) \cup \chi(x, y) = \chi_b(t_1 \cup t_2, t_3 \cup t_4)$ otherwise

- **Ordinary operation’s (τ node’s) transfer functions:**
  - Insensitive (w.l.o.g. binary operation): $T_\tau :: A_a \times A_b \rightarrow A_c$
  - Sensitive: $S_\tau :: X_a \times X_b \rightarrow X_c$, and for $a_1, a_2 \in A_a, A_b$ and $t_1, t_2 \in X_a, X_b$:
    $S_\tau(a_1, a_2) = T_\tau(a_1, a_2)$
    $S_\tau(\chi_a(t_1, t_2), \chi_b(t_3, t_4)) = \chi_b(S_\tau(\chi_a(t_1, t_2), \chi_a, 1), S_\tau(\chi_b(t_3, t_4), \chi_a, 2))$
    with $k$ larger of $x, y$ and cof is the co-factorization

Sensitive Transfer Schema (case a)

Sensitive Transfer Schema (case b)

Co-factorization

- $\text{cof}(\chi_a(t_1, t_2), \chi_a, i)$ selects the $i$-th branch of a $\chi$-term if $\chi_a = \chi_k$
  and returns the whole $\chi$-term, otherwise

- $\text{cof}(\chi_a(t_1, t_2), \chi_a, i)$ = $\chi_a(t_1, t_2)$ iff $k > x$
- $\text{cof}(\chi_a(t_1, t_2), \chi_a, 1)$ = $t_1$ iff $k = x$
- $\text{cof}(\chi_a(t_1, t_2), \chi_a, 2)$ = $t_2$ iff $k = x$
Example revisited: insensitive

- SSA node ⊕
- Semantic
  - \([\text{⊕}]: \text{Int} \times \text{Int} \to \text{Int}\)
  - \([\text{⊕}](a,b) = a+b\)
- Abstract Int values \{⊥, 1, 2, ..., maxint, ⊤\}
- Context-insensitive transfer function:
  - \(T_\#(\text{⊕}, x) = T_\#(x, \text{⊥}) = \text{⊥}\)
  - \(T_\#(\text{⊕}, \text{⊥}) = T_\#(\text{⊥}, x) = \text{τ}\)
  - \(T_\#(a,b) = [\text{⊕}](a,b) = a+b\) for \(a,b \in \text{Int}\)
- Context-insensitive meet function
  - \(T_\#(\text{⊥}, x) = T_\#(x, \text{⊥}) = x\)
  - \(T_\#(\text{⊥}, \text{⊥}) = T_\#(\text{⊥}, x) = \text{τ}\)
  - \(T_\#(x, x) = x\)
  - \(T_\#(x, y) = \text{τ}\)

Context-sensitive \(a \oplus b\)

\[
S_\otimes( x_3(x_1(1,2), x_2(x_1(1,2), 2), x_3(3,4)) ) = \ldots
\]

\[
= x_3( S_\otimes( \text{cof}( x_1(1,2), x_2(x_1(1,2), 2), x_3(3,4) ) , x_3(1) ) , x_3( x_2(x_1(1,2)(3), x_3(4)) ) , x_3( x_2(1(1,2)(3), x_3(4)) , x_3(2) ) ) )
\]

\[
= x_3( S_\otimes( x_3(1), x_3(2), x_3(3) ) , x_3( x_3(1), x_3(2), x_3(4)) ) , x_3( S_\otimes( x_2(x_1(1,2)(2), x_3(1)) , x_3( x_2(x_1(1,2)(2), x_3(4)) , x_3(2) ) , x_3( x_2(x_3(1), x_3(4)) ) , x_3( x_2(1(1,2)(3), x_3(4)) ) )
\]

\[
= x_3( x_3( x_3(1), x_3(2), 3) , x_3( S_\otimes( x_3(1), x_3(2), 4)) , x_3( x_3(1), x_3(2), 3))
\]

\[
S_\otimes( x_3(1), x_3(2), x_3(3), x_3(4))
\]

Context-sensitive \(a \oplus b\) (cont.)

\[
S_\otimes( x_3(x_1(1,2), x_2(x_1(1,2), 2)) , x_3(3,4)) = \ldots
\]

\[
= x_3( x_3( x_3(1), x_2(x_1(1,2), 2), x_3(3,4)) , x_3(1) , x_3( x_3(1), x_3(2)) )
\]

\[
= x_3( x_3( x_3(1), x_2(x_1(1,2), 2), x_3(3,4)) , x_3(1) , x_3( x_3(1), x_3(2)))
\]

\[
= x_3( x_3( x_3(1), x_2(x_1(1,2), 2), x_3(3,4)) , x_3(1) , x_3( x_3(1), x_3(2)))
\]

\[
= x_3( x_3( x_3(1), x_2(x_1(1,2), 2), x_3(3,4)) , x_3(1) , x_3( x_3(1), x_3(2)))
\]

\[
= x_3( x_3( x_3(1), x_2(x_1(1,2), 2), x_3(3,4)) , x_3(1) , x_3( x_3(1), x_3(2)))
\]

\[
= x_3( x_3( x_3(1), x_2(x_1(1,2), 2), x_3(3,4)) , x_3(1) , x_3( x_3(1), x_3(2)))
\]

\[
= x_3( x_3( x_3(1), x_2(x_1(1,2), 2), x_3(3,4)) , x_3(1) , x_3( x_3(1), x_3(2)))
\]

\[
= x_3( x_3( x_3(1), x_2(x_1(1,2), 2), x_3(3,4)) , x_3(1) , x_3( x_3(1), x_3(2)))
\]
Assignment III

Given the SSA fragment on the left

- Perform context-insensitive data-flow analysis (using the definitions on the previous slides). What is the the value at the entry of node x?
- Perform context-sensitive data-flow analysis (using the definitions on the previous slides). What is the the value at the entry of node x?
- Why is the former less precise than the latter?
- Construct a scenario where you could take advantage of that precision in an optimization!

Inter-Procedural Analysis

- Tasks:
  - Find the impact of calls to caller and callee:
    - Find recursive procedures
    - Find values that might change (side effects) and those that definitely don’t
    - Anti-dependencies: are there calls that needn’t be executed in strict order
  - Determine the called procedures (in the presence of polymorphism and procedure variables)
  - Which procedure bodies could be inlined
- Reason:
  - Extend the value numbering over procedure boundaries
  - Simplify calls of non-recursive and tail-recursive procedures
  - Replace indirect calls by direct calls
  - Inlining
- Precondition:
  - Call graph
  - Complete program+runtime system (no separated compilation)

Simple call graph

- Nodes are methods edges calls

```c
void p() {
    r(); q(); r()
};
void q() {
    if B then s()
    else t()
};
void r() {
    if B then q()
};
void s() {
    if ¬B then r()
};
void t() {
    ...
}
```

- Main is a procedure
- Cycles show potential recursion
- Multiple calls represented only once
- No tracing call → procedure → return possible
- In oo programs: long chains of calls
- # of ”dead ends” in call graph > 50%

Complex Call Graph

- New nodes in SSA graph
  - callBegin – call of a procedure
  - callEnd – take the results into the context of the caller
- New edges in SSA graph
  - Edge from „callBegin“ to „start“ node of the callee procedure
  - Edge from „return“ to „callEnd“ all possible caller nodes
  - Edge from „callBegin“ to „callEnd“ node (control dependency)
  - Edge from actual parameters of the caller to formal parameter uses in the procedure (φ-function if more than one call possible)
  - Edge from results to „callEnd“ in callers
- Polymorphism is resolved by explicit dispatcher
- Interprocedural dataflow analysis now possible
Procedures

- Extension to Memory SSA:
  - New node: begin and end of calls distinguished
  - Edges: connection between caller and callees

Construction of Inter-Procedural SSA

- Inter-procedural Value Analysis
- Almost everything as usual, but:
  - No initialization of value numbers with "undefined" for „start“ node – take callers’ values instead
  - No new value numbers after „callEnd“ – take values from procedure instead
  - \(\phi\)-Functions on different values possible at „start“ node,
  - „callEnd“ is a kind of \(\phi\)-function too since actual call target is not decidable and there will be more than one returns reaching the same „callEnd“
  - Immature \(\phi\)’-functions, if values are unknown (because procedure not visited yet)
  - Fix point iteration until no algebraic identities detected (as before)

Example

- Example Construction
Example Construction

Observation

- Information merges
  - at start of a procedure
  - at return from a procedure (non realizable path, non decidable call targets)
- No distinction of the call contexts so far
- Comparison
  - Intra procedural analysis:
    - (Data flow-)values are undefined at procedure start
    - Call sets all values to undefined in caller (new value numbers)
  - Inter procedural analysis:
    - (Data flow-)values are supremum of values of the caller at procedure start
    - Call of procedure sets value(-number)s in caller to value(-number)s at the end of the callees, non-realizable paths
- Problem: too strong abstraction
- Idea: Distinguish the contexts ($\chi$ terms)

Inter-procedural context-sensitive Value Analysis

- Like context-insensitive inter-procedural value analysis but:
  - use $\chi$-terms to analyze values at certain program points (SSA-nodes)
    - After SSA construction
    - Initially construct the $\chi$-terms from the $\phi$-functions
  - analyze values at start and end of calls with differentiation of the contexts
  - fixpoint iteration until no simplifications of $\chi$-terms possible
  - if necessary (analysis runs out of memory) melt $\chi$-terms

Example (cont.)
Before/after SSA construction

\[ p: \begin{align*} x &= 0; \\ x &= r(x); \\ x &= q(x); \\ x &= r(x) \end{align*} \]

\[ r: \begin{align*} \text{if } (x = 1) \text{ then } x &= s(x); \\ \text{return } x \end{align*} \]

\[ q: \begin{align*} \text{if } (x = 1) \text{ then } x &= s(x) \text{ else } x &= t(x); \\ \text{return } x \end{align*} \]

\[ s: \begin{align*} \text{if } (x = 0) \text{ then } x &= r(x); \\ \text{return } x \end{align*} \]

\[ t: \begin{align*} \text{return } x + 1 \end{align*} \]

Virtually inline \( r \) in first call in \( p \)

\[ p: \begin{align*} x_1 &= 0, \\ B_r(x_1), \\ x_6 &= E_r, \\ B_q(x_6), \\ x_9 &= E_q, \\ B_r(x_9), \\ x_6 &= E_r \\ r: \begin{align*} x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \\ B_s(x_4), \\ x_{11} &= E_s, \\ x_6 &= \chi_{\{k_4=1\}}(x_{11}, x_4) \\ q: \begin{align*} B_s(x_6), \\ x_{11} &= E_s, \\ B_t(x_6), \\ x_{12} &= E_t, \\ x_9 &= \chi_{\{x_6=1\}}(x_{11}, x_{12}) \\ s: \begin{align*} x_{10} &= \chi_{\{r,q\}}(x_4, x_6), \\ B_r(x_{10}), \\ x_6 &= E_r, \\ x_{11} &= \chi_{\{x_{10}=0\}}(x_6, x_{10}) \\ t: \begin{align*} x_{12} &= x_6 + 1 \end{align*} \end{align*} \]

Specialize \( r \) in context of first call in \( p \)

\[ p: \begin{align*} x_1 &= 0, \\ B_r(x_1), \\ x_6 &= E_r, \\ B_q(x_6), \\ x_9 &= E_q, \\ B_r(x_9), \\ x_6 &= E_r \\ r: \begin{align*} x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \\ B_s(x_4), \\ x_{11} &= E_s, \\ x_6 &= \chi_{\{k_4=1\}}(x_{11}, x_4) \\ q: \begin{align*} B_s(x_6), \\ x_{11} &= E_s, \\ B_t(x_6), \\ x_{12} &= E_t, \\ x_9 &= \chi_{\{x_6=1\}}(x_{11}, x_{12}) \\ s: \begin{align*} x_{10} &= \chi_{\{r,q\}}(x_4, x_6), \\ B_r(x_{10}), \\ x_6 &= E_r, \\ x_{11} &= \chi_{\{x_{10}=0\}}(x_6, x_{10}) \\ t: \begin{align*} x_{12} &= x_6 + 1 \end{align*} \end{align*} \]

SSA form and initial \( \chi \)-Terms

\[ p: \begin{align*} x_1 &= 0, \\ B_r(x_1), \\ x_6 &= E_r, \\ B_q(x_6), \\ x_9 &= E_q, \\ B_r(x_9), \\ x_6 &= E_r \\ r: \begin{align*} x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \\ B_s(x_4), \\ x_{11} &= E_s, \\ x_6 &= \chi_{\{k_4=1\}}(x_{11}, x_4) \\ q: \begin{align*} B_s(x_6), \\ x_{11} &= E_s, \\ B_t(x_6), \\ x_{12} &= E_t, \\ x_9 &= \chi_{\{x_6=1\}}(x_{11}, x_{12}) \\ s: \begin{align*} x_{10} &= \chi_{\{r,q\}}(x_4, x_6), \\ B_r(x_{10}), \\ x_6 &= E_r, \\ x_{11} &= \chi_{\{x_{10}=0\}}(x_6, x_{10}) \\ t: \begin{align*} x_{12} &= x_6 + 1 \end{align*} \end{align*} \]

Specialize \( r \) in context of first call in \( p \)

\[ p: \begin{align*} x_1 &= 0, \\ B_r(x_1), \\ x_6 &= E_r, \\ B_q(x_6), \\ x_9 &= E_q, \\ B_r(x_9), \\ x_6 &= E_r \\ r: \begin{align*} x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \\ B_s(x_4), \\ x_{11} &= E_s, \\ x_6 &= \chi_{\{k_4=1\}}(x_{11}, x_4) \\ q: \begin{align*} B_s(x_6), \\ x_{11} &= E_s, \\ B_t(x_6), \\ x_{12} &= E_t, \\ x_9 &= \chi_{\{x_6=1\}}(x_{11}, x_{12}) \\ s: \begin{align*} x_{10} &= \chi_{\{r,q\}}(x_4, x_6), \\ B_r(x_{10}), \\ x_6 &= E_r, \\ x_{11} &= \chi_{\{x_{10}=0\}}(x_6, x_{10}) \\ t: \begin{align*} x_{12} &= x_6 + 1 \end{align*} \end{align*} \]

Specialize \( r \) in context of first call in \( p \)

\[ p: \begin{align*} x_1 &= 0, \\ B_r(x_1), \\ x_6 &= E_r, \\ B_q(x_6), \\ x_9 &= E_q, \\ B_r(x_9), \\ x_6 &= E_r \\ r: \begin{align*} x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \\ B_s(x_4), \\ x_{11} &= E_s, \\ x_6 &= \chi_{\{k_4=1\}}(x_{11}, x_4) \\ q: \begin{align*} B_s(x_6), \\ x_{11} &= E_s, \\ B_t(x_6), \\ x_{12} &= E_t, \\ x_9 &= \chi_{\{x_6=1\}}(x_{11}, x_{12}) \\ s: \begin{align*} x_{10} &= \chi_{\{r,q\}}(x_4, x_6), \\ B_r(x_{10}), \\ x_6 &= E_r, \\ x_{11} &= \chi_{\{x_{10}=0\}}(x_6, x_{10}) \\ t: \begin{align*} x_{12} &= E_r + 1 \end{align*} \end{align*} \]

Specialize \( r \) in context of first call in \( p \)

\[ p: \begin{align*} x_1 &= 0, \\ B_r(x_1), \\ x_6 &= E_r, \\ B_q(x_6), \\ x_9 &= E_q, \\ B_r(x_9), \\ x_6 &= E_r \\ r: \begin{align*} x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \\ B_s(x_4), \\ x_{11} &= E_s, \\ x_6 &= \chi_{\{k_4=1\}}(x_{11}, x_4) \\ q: \begin{align*} B_s(x_6), \\ x_{11} &= E_s, \\ B_t(x_6), \\ x_{12} &= E_t, \\ x_9 &= \chi_{\{x_6=1\}}(x_{11}, x_{12}) \\ s: \begin{align*} x_{10} &= \chi_{\{r,q\}}(x_4, x_6), \\ B_r(x_{10}), \\ x_6 &= E_r, \\ x_{11} &= \chi_{\{x_{10}=0\}}(x_6, x_{10}) \\ t: \begin{align*} x_{12} &= E_r + 1 \end{align*} \end{align*} \]
Optimize r in context of first call in p

\[ \begin{align*}
p: \quad x_1 &= 0, \quad B_{r_1}(x_1), \quad E_{r_1} = \chi_{\{x_1\}}(x_1, x_1), \quad B_{q}(E_{r_1}), \quad x_9 = E_{q}, \quad B_{r_2}(x_9), \quad x_6 = E_{r_2} \\
r: \quad x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \quad B_{s}(x_4), \quad x_{11} = E_{s}, \quad x_6 = \chi_{\{x_{11}\}}(x_11, x_4) \\
q: \quad B_{s}(E_{r_1}), \quad x_{11} = E_{s}, \quad B_{r}(E_{r_1}), \quad x_{12} = E_{r}, \quad x_9 = \chi_{\{x_9\}}(x_11, x_12) \\
s: \quad x_{10} = \chi_{\{r,q\}}(x_4, E_{r_1}), \quad B_{r}(x_{10}), \quad x_6 = E_{r}, \quad x_{11} = \chi_{\{x_{10}\}}(x_6, x_{10}) \\
t: \quad x_{12} = E_{r_1} + 1
\end{align*} \]

Propagate Result

\[ \begin{align*}
p: \quad x_1 &= 0, \quad B_{r_1}(x_1), \quad E_{r_1} = 0, \quad B_{q}(0), \quad x_9 = E_{q}, \quad B_{r_2}(x_9), \quad x_6 = E_{r_2} \\
r: \quad x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \quad B_{s}(x_4), \quad x_{11} = E_{s}, \quad x_6 = \chi_{\{x_{11}\}}(x_11, x_4) \\
q: \quad B_{s}(0), \quad x_{11} = E_{s}, \quad B_{r}(0), \quad x_{12} = E_{r}, \quad x_9 = \chi_{\{x_9\}}(x_11, x_12) \\
s: \quad x_{10} = \chi_{\{r,q\}}(x_4, 0), \quad B_{r}(x_{10}), \quad x_6 = E_{r}, \quad x_{11} = \chi_{\{x_{10}\}}(x_6, x_{10}) \\
t: \quad x_{12} = 0 + 1
\end{align*} \]

Virtually inline r in second call in p

\[ \begin{align*}
p: \quad x_1 &= 0, \quad B_{r_1}(x_1), \quad E_{r_1} = 0, \quad B_{q}(0), \quad x_9 = E_{q}, \quad B_{r_2}(x_9), \quad x_6 = E_{r_2} \\
r: \quad x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \quad B_{s}(x_4), \quad x_{11} = E_{s}, \quad x_6 = \chi_{\{x_{11}\}}(x_11, x_4) \\
q: \quad B_{s}(0), \quad x_{11} = E_{s}, \quad B_{r}(0), \quad x_{12} = E_{r}, \quad x_9 = \chi_{\{x_9\}}(x_11, x_12) \\
s: \quad x_{10} = \chi_{\{r,q\}}(x_4, 0), \quad B_{r}(x_{10}), \quad x_6 = E_{r}, \quad x_{11} = \chi_{\{x_{10}\}}(x_6, x_{10}) \\
t: \quad x_{12} = 1
\end{align*} \]

Specialize r in second call in p

\[ \begin{align*}
p: \quad x_1 &= 0, \quad B_{r_1}(x_1), \quad E_{r_1} = 0, \quad B_{q}(0), \quad x_9 = E_{q}, \quad B_{r_2}(x_9), \quad x_6 = E_{r_2} \\
r: \quad x_4 &= \chi_{\{p_1,p_2,s\}}(x_1, x_9, x_{10}), \quad B_{s}(x_4), \quad x_{11} = E_{s}, \quad x_6 = \chi_{\{x_{11}\}}(x_11, x_4) \\
q: \quad B_{s}(0), \quad x_{11} = E_{s}, \quad B_{r}(0), \quad x_{12} = E_{r}, \quad x_9 = \chi_{\{x_9\}}(x_11, x_12) \\
s: \quad x_{10} = \chi_{\{r,q\}}(x_4, 0), \quad B_{r}(x_{10}), \quad x_6 = E_{r}, \quad x_{11} = \chi_{\{x_{10}\}}(x_6, x_{10}) \\
t: \quad x_{12} = 1
\end{align*} \]
Virtually inline s in call in r

\[ p: x_1=0, B_r(x_1), E_{r_1}=0, B_q(x_0), E_{q_2}=\chi_{(x_0=0)}(x_1, x_0) \]
\[ r: x_4=\chi_{(p_1,p_2,s)}(x_1, x_9, x_{10}), B_s(x_9), E_{r_2} = \chi_{(x_9=1)}(x_11, x_9) \]
\[ q: B_s(0), x_{11}=E_s, B_0(0), x_{12}=E_r, x_0=x_{12} \]
\[ s: x_{10}=\chi_{(x_0=0)}(x_4, 0), B,(x_{10}), x_0=E_r, x_{11}=\chi_{(x_{10}=0)}(x_6, x_{10}) \]
\[ t: x_{12}=1 \]

Specialize s in call in r

\[ p: x_1=0, B_r(x_1), E_{r_1}=0, B_q(x_0), E_{q_2}=\chi_{(x_0=0)}(x_1, x_0) \]
\[ r: x_4=\chi_{(p_1,p_2,s)}(x_1, x_9, x_{10}), B_s(x_9), E_{r_2} = \chi_{(x_9=0)}(x_6, x_{9}) \]
\[ q: B_s(0), x_{11}=E_s, B_0(0), x_{12}=E_r, x_0=x_{12} \]
\[ s: x_{10}=\chi_{(x_0=0)}(x_4, 0), B,(x_{10}), x_0=E_r, x_{11}=\chi_{(x_{10}=0)}(x_6, x_{10}) \]
\[ t: x_{12}=1 \]

Optimize s in call in r

\[ p: x_1=0, B_r(x_1), E_{r_1}=0, B_q(x_0), E_{q_2}=\chi_{(x_0=0)}(x_1, x_0) \]
\[ r: x_4=\chi_{(p_1,p_2,s)}(x_1, x_9, x_{10}), B_s(x_9), E_{r_2} = \chi_{(x_9=0)}(x_6, x_{9}) \]
\[ q: B_s(0), x_{11}=E_s, B_0(0), x_{12}=E_r, x_0=x_{12} \]
\[ s: x_{10}=\chi_{(x_0=0)}(x_4, 0), B,(x_{10}), x_0=E_r, x_{11}=\chi_{(x_{10}=0)}(x_6, x_{10}) \]
\[ t: x_{12}=1 \]

Propagate Result

\[ p: x_1=0, B_r(x_1), E_{r_1}=0, B_q(x_0), E_{q_2}=\chi_{(x_0=0)}(x_1, x_0) \]
\[ r: x_4=\chi_{(p_1,p_2,s)}(x_1, x_9, x_{10}), B_s(x_9), E_{r_2} = \chi_{(x_9=0)}(x_6, x_{9}) \]
\[ q: B_s(0), x_{11}=E_s, B_0(0), x_{12}=E_r, x_0=x_{12} \]
\[ s: x_{10}=\chi_{(x_0=0)}(x_4, 0), B,(x_{10}), x_0=E_r, x_{11}=\chi_{(x_{10}=0)}(x_6, x_{10}) \]
\[ t: x_{12}=1 \]
Optimize s

\[ p: x_1=0, B_{r1}(x_1), E_{r1} =0, B_q(0), x_9= E_q, B_{r2}(x_9), E_{r2} =\chi_{(x_9=1)}(x_{11}, x_9) \]

\[ r: x_4=\chi_{(p_1,p_2,s)}(x_1, x_9, x_{10}), x_6=x_4 \]

\[ q: B_s(0), x_{11}=E_s, B_t(0), x_{12}=E_t, x_9=x_{12} \]

\[ s: x_{10}=0, B_s(x_{10}), x_6=E_r, x_{11}=\chi_{(x_{10}=0)}(x_6, x_{10}) \]

\[ t: x_{12}=1 \]

Clean and final simplifications

\[ p: B_{r1}(0), E_{r1} =0, B_q(0), E_q=1, B_{r2}(1), E_{r2} =\chi_{(1=1)}(x_4, 1) \]

\[ r: x_4=\chi_{(p_1,p_2,s)}(0, 1, 0), \text{ret } x_4 \]

\[ q: B_s(0), x_4=E_s, B_t(0), E_t=1, \text{ret } 1 \]

\[ s: B_s(0), x_4=E_r, \text{ret } x_4 \]

\[ t: \text{ret } 1 \]

Result is 1 😊

Open Problems

- Efficient implementation – large problems
- Memory driven control of fixed point iteration
- Open compilation
  - unknown procedures (components)
  - binary components
- Readability of analysis results (necessary when applied for program comprehension etc.)

Outline

- Introduction to SSA, Construction, Destruction
- Analyses and Optimizations
  - Classic analyses and optimizations on SSA representations
  - Context-sensitive and Inter-procedural analyses and optimizations
  - Heap analyses and optimizations
Optimizations on Memory

- Elimination of memory accesses.
- Elimination of object creations.
- Elimination non essential dependencies.
- Those are normalizing transformations for further analyses

Nothing new under the sun:
- Define abstract values for variables, addresses, memory
- Define context-insensitive transfer functions for memory relevant SSA nodes (Load, Store, Call)
- Generalization to context-sensitive analyses (discussed already)
- Optimizations as graph transformations (discussed already)

Memory Values

- Initially: memory is a single value object
- Desired: refinement of that memory object
- Distinguish:
  - heap and stack
  - local arrays with different name
  - disjoint index sets in an array (odd/even etc.)
  - different types of heap objects
  - objects with same type but statically different creation program point
  - objects with same creation program point but with statically different path to that creation program point (execution context, context-sensitive)
- Differentiation schema is called Name Schema (NS)

Example for Context Sensitive NS

class List{
  Object value;
  List tail;
  List(int length){
    tail = new List(length-1);
  }
  ...
  ...  // Heap object
  ...  // with same object type but
  ...  // different creation point and
  ...  // hence distinguished
  List l = new List(10);
  ...
}

Abstract Variables/Addresses/Memory Values

- Abstract addresses $A$ are sets of object creation points: objects with same static creation program point are stored at the same abstract address
- Abstract variables $Var$ are triples $(a, n, i)$ defined by
  - Abstract object address $a$
  - Offset $a$ attribute name $n$
  - Index expression $i$ if entry is an array
- Abstract memory $M$ is a mapping abstract variables to abstract values (might also be abstract addresses)
Updates of Memory

- Given an abstract address of a store operation uniquely
- In general, this abstract address points to more than one real memory cell
- A store operation overwrites only one of these cells, all others contain the same value
- Hence a store to an abstract memory address adds a new possible (abstract) value - weak update
- Only if guaranteed that abstract address matches a concrete address, a new possible (abstract) value overwrites the old value - strong update

Auxiliary function:
\[ u(M, (a, n, T), v) = \begin{cases} v & \text{if strong update possible} \\ M(a, n, T) \cup v & \text{otherwise, weak update} \end{cases} \]

Transfer functions (insensitive)

- \( T_{\text{store}}(M, \mathit{Var}, v) = M[(\mathit{var}_1 \mapsto u(M, \mathit{var}_1, v)) \ldots (\mathit{var}_k \mapsto u(M, \mathit{var}_k, v))] \)
  \[ V = \{ \mathit{var}_1 \ldots \mathit{var}_k \} \quad \mathit{var}_i = (a_i, n_i, T) \]

- \( T_{\text{load}}(M, \mathit{Var}) = (M, M(\mathit{var}_1) \cup \ldots \cup M(\mathit{var}_k)) \)
  \[ V = \{ \mathit{var}_1 \ldots \mathit{var}_k \} \quad \mathit{var}_i = (a_i, n_i, T) \]

- \( T_{\text{alloc}}(\mathit{type})(M) = (M[a_{\text{new}}, n_1, T] \mapsto \bot) \ldots [(a_{\text{new}}, n_k, T) \mapsto \bot], a_{\text{new}}) \)
  \{n_1 \ldots n_k\} attributes of \textit{type}

Example: Main Loop Inner Product Algorithm

```
init(int size)
VecIter iter()
Vector<T>
T times(Vector)

VectorArray<T>
VecIter iter()
VectorIter<T>
boolean hasNext() T next()
```

```
T times(vector v){
i1 = iter();
i2 = v.iter();
for (s = 0; i1.hasNext(); )
    s = s+i1.next()*i2.next();
}
```

Example: SSA

```
init(int size)
VecIter iter()
Vector<T>
T times(Vector)

VectorArray<T>
VecIter iter()
VectorIter<T>
boolean hasNext() T next()
```

```
T times(vector v){
i1 = iter();
i2 = v.iter();
for (s = 0; i1.hasNext(); )
    s = s+i1.next()*i2.next();
}````
No provably different memory addresses

[Diagram showing a flowchart with nodes labeled "Load", "Inc", and "Store".]

Not applicable!

Actually two Iterators?

[Diagram showing a flowchart with nodes labeled "Load", "Inc", and "Store".]

Elimination of non essential dependencies

Initialization: disjoint memory guaranteed

[Diagram showing a flowchart with nodes labeled "Allocate", "Load", "Inc", and "Store".]

Memory objects replaced by values

[Diagram showing a flowchart with nodes labeled "Allocate", "Load", "Inc", and "Store".]

Elimination of reading memory accesses
Value numbering proofs equivalence

Example revisited

Optimization only possible due to joint application of single techniques:
- Global analysis
- Elimination of polymorphism
- Elimination of non essential dependencies
- Elimination of memory operations
- Traditional optimizations