Global Value Numbering

Sebastian Hack
hack@cs.uni-saarland.de

17. Dezember 2013
Value Numbering

- Replace second computation of $a + 1$ with a copy from $x$
Value Numbering

- Goal: Eliminate redundant computations
- Find out if two variables have the same value at given program point
  - In general undecidable
- Potentially replace computation of latter variable with contents of the former
- Resort to Herbrand equivalence:
  - Do not consider the interpretation of operators
  - Two expressions are equal if they are structurally equal

- This lecture: A costly program analysis which finds all Herbrand equivalences in a program and a “light-weight” version that is often used in practice.
Herbrand Interpretation

- The Herbrand interpretation $\mathcal{I}$ of an $n$-ary operator $\omega$ is given as

  $$\mathcal{I}(\omega) : T^n \rightarrow T \quad \mathcal{I}(\omega)(t_1, \ldots, t_n) := \omega(t_1, \ldots, t_n)$$

  Especially, constants are mapped to themselves.

- With a state $\sigma$ that maps variables to terms

  $$\sigma : V \rightarrow T$$

- we can define the Herbrand semantics $\langle t \rangle \sigma$ of a term $t$

  $$\langle t \rangle \sigma := \begin{cases} 
  \sigma(v) & \text{if } t = v \text{ is a variable} \\
  \mathcal{I}(\omega)(\langle x_1 \rangle \sigma, \ldots, \langle x_n \rangle \sigma) & \text{if } t = \omega(x_1, \ldots, x_n)
  \end{cases}$$
Programs with Herbrand Semantics

- We now interpret the program with respect to the Herbrand semantics.

- For an assignment
  \[ x \leftarrow t \]
  the semantics is defined by:
  \[
  [x \leftarrow t] \sigma := \sigma [\langle t \rangle \sigma / x]
  \]

- The state after executing a path \( p : \ell_1, \ldots, \ell_n \) starting with state \( \sigma_0 \) is then:
  \[
  [p] \sigma_0 := ([\ell_n] \circ \cdots \circ [\ell_1]) \sigma_0
  \]

- Two expressions \( t_1 \) and \( t_2 \) are **Herbrand equivalent** at a program point \( \ell \) iff
  \[
  \forall p : r, \ldots, \ell. \langle t_1 \rangle [p] \sigma_0 = \langle t_2 \rangle [p] \sigma_0
  \]
Kildall’s Analysis

- Track Herbrand equivalences with a **forward** data flow analysis

- A lattice element is an equivalence class of the terms **and** variables of the program

- The equivalence relation is a **congruence relation** w.r.t. to the operators in our expression language.
  For each operator $\omega$, each eq. relation $R$, and $e, e_1, \cdots \in V \cup T$:
  \[
  e \ R (e_1 \omega e_2) \implies e_1 \ R \ e'_1 \implies e_2 \ R \ e'_2 \implies e \ R (e'_1 \omega e'_2)
  \]

- Two equivalence classes are joined by intersecting them
  \[
  R \sqcup S := R \cap S := \{(a, b) \mid a \ R \ b \land a \ S \ b\}
  \]

- $\bot = \{(x, y) \mid x, y \in V \cup T\}$
  - optimistically assume all variables/terms are equivalent

- Initialize with $\top = \{(x, x) \mid x \in V \cup T\}$
  - at the beginning, nothing is equivalent
Kildall’s Analysis

Example

\[
\begin{align*}
a &:= 2 \\
x &:= a + 1
\end{align*}
\]

\[
\begin{align*}
\{[a, 2], [x, a + 1, 2 + 1]\}
\end{align*}
\]

\[
\begin{align*}
a &:= 3 \\
x &:= a + 1
\end{align*}
\]

\[
\begin{align*}
\{[a, 3], [x, a + 1, 3 + 1]\}
\end{align*}
\]

\[
\begin{align*}
y &:= a + 1
\end{align*}
\]

\[
\begin{align*}
\{[x, a + 1]\}
\end{align*}
\]

\[
\begin{align*}
\{[x, y, a + 1]\}
\end{align*}
\]
Kildall’s Analysis
Transfer Functions

... of an assignment

\( \ell : x \leftarrow t \)

- Compute a new partition checking (in the old partition) who is equivalent if we replace \( x \) by \( t \)

\[
[x \leftarrow t]^{\#} R := \{(t_1, t_2) \mid t_1[t/x] R t_2[t/x]\}
\]
Kildall’s Analysis

Example

\[
x := 0 \\
y := x + 1
\]

\[
x := x + 1 \\
y := y + 1
\]
Kildall’s Analysis

Example

\[
\begin{align*}
x &:= 0 \\
y &:= x + 1
\end{align*}
\]

\[
\begin{align*}
\{[x, 0], [y, x + 1, 0 + 1]\}
\end{align*}
\]

\[
\begin{align*}
\{[y, x + 1]\}
\end{align*}
\]

\[
\begin{align*}
x &:= x + 1 \\
y &:= y + 1
\end{align*}
\]

\[
\begin{align*}
\{[y, x + 1]\}
\end{align*}
\]

\[
\begin{align*}
\{[x, y]\}
\end{align*}
\]

\[
\begin{align*}
\{[x, y]\}
\end{align*}
\]
Kildall’s Analysis

Comments

- One can show that Kildall’s Analysis is **sound and complete**

- Naïve implementations suffer from exponential explosion (pathological):
  - Because the equivalence relation must be a congruence size of eq. classes can explode:
    
    \[
    R = \{[a, b], [c, d], [e, f], [x, a + c, a + d, b + c, b + d], \\
    [y, x + e, x + f, (a + c) + e, \ldots, (b + d) + f]\}
    \]

- In practice: Do not make congruence explicit in representation

- Instead: Before analysis, scan program for all appearing expressions (and subexpressions!) and only include those in the representation of the equivalence classes
The Alpern, Wegman, Zadeck (AWZ) Algorithm

- Incomplete

- Flow-insensitive
  - does not compute the equivalences for every program point but sound equivalences for the whole program

- Uses SSA
  - Control-flow joins are represented by $\phi$s
  - Treat $\phi$s like every other operator (cause for incompleteness)
  - SSA compensates flow-insensitivity

- Interpret the SSA data dependence graph as a finite automaton and minimize it
  - Refine partitions of “equivalent states”
  - Using Hopcroft’s algorithm, this can be done in $O(e \cdot \log e)$
The AWZ Algorithm

- In contrast to finite automata, do not create two partitions but a class for every operator symbol
  - Note that the $\phi$’s block is part of the operator
  - Two $\phi$’s from different blocks have to be in different classes

- Optimistically place all nodes with the same operator symbol in the same class
  - Finds the least fixpoint
  - You can also start with singleton classes and merge but this will (in general) not give the least fixpoint

- Successively split class when two nodes in the class are detected not equivalent
The AWZ Algorithm

Example

$$x := 0$$
$$y := 0$$

$$x := x + 1$$
$$y := y + 1$$
The AWZ Algorithm

Example

\begin{align*}
x_0 & := 0 \\
y_0 & := 0 \\
x_1 & := \phi_2(x_2, x_0) \\
y_1 & := \phi_2(y_2, y_0) \\
x_2 & := x_1 + 1 \\
y_2 & := y_1 + 1
\end{align*}
The AWZ Algorithm

Example

\[
\phi_2 x_1 + x_0 y_1 + x_0 y_0
\]

\[
\phi_2 x_2 + 0 x_0 + 0 y_0
\]
The AWZ Algorithm

Example

\[ \phi(x_1, y_1) + x_2, y_2 = 0 \]

\[ x_0, y_0 = 1 \]
Kildall compared to AWZ

1

2

\[
a_0 := 2 \\
x_0 := a_0 + 1
\]

3

\[
a_1 := 3 \\
x_1 := a_1 + 1
\]

4

\[
a_2 := \phi_4(a_0, a_1) \\
x_2 := \phi_4(x_0, x_1) \\
y_0 := a_2 + 1
\]
Kildall compared to AWZ
Kildall compared to AWZ

\[ \phi_4 + a_2 \rightarrow y_0 \]

\[ \phi_4 + x_2 \rightarrow \]

\[ a_0 \rightarrow 2 \]

\[ a_1 \rightarrow 3 \]

\[ x_0 \rightarrow 1 \]

\[ x_1 \rightarrow \]