Global Value Numbering

Sebastian Hack
hack@cs.uni-saarland.de

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:
  \[ [[x \leftarrow t]] \sigma := \sigma \left[ \langle t \rangle \sigma / x \right] \]

- 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 a structured partition of the terms and variables of the program
  - Two terms in the same partition are deemed equivalent
  - A partition $\pi$ is structured iff
    \[(e, e_1 \omega e_2) \in \pi \land (e_1, e_1') \in \pi \land (e_2, e_2') \in \pi \implies (e, e_1' \omega e_2')\]

- Two partitions are joined by intersecting them

- $\bot$ is the partition that contains all terms and variables
  - Optimistically assume all variables/terms are equivalent

- The initial value for the start node is the partition that consists of singleton equivalence classes
  - At the beginning, nothing is equivalent
Kildall’s Analysis

Example

```
a := 2
x := a + 1

∅

∅

∅

∅

{[a, 2], [x, a + 1, 2 + 1]}

{[x, a + 1]}

{[x, y, a + 1]}

{[a, 3], [x, a + 1, 3 + 1]}

y := a + 1

{x, a + 1}
```
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 \( t \)

\[ F_\ell(\pi) := \{(t_1, t_2) \mid (t_1[t/x], t_2[t/x]) \in \pi\} \]
Kildall’s Analysis

Example

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

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

Example

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

\{ [x, 0], [y, x + 1, 0 + 1] \}

\{ [y, x + 1] \}

\rightarrow

\{ [y, x + 1] \}

\rightarrow

\{ [y, x + 1] \}

\rightarrow

\{ [x, y] \}

\rightarrow

\{ [x, y] \}

\rightarrow

\{ [x, y] \}
Kildall’s Analysis
Comments

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

- However, it suffers from exponential explosion (pathological):
  - In the worst case \( \pi_1 \cap \pi_2 \) can have \(|\pi_1| \cdot |\pi_2|\) equiv. classes
  - In a naïve implementation also the size of one equiv. class can explode due to the structuring constraint. For example:

\[
\pi = \{[a, b], [c, d], [e, f], [x, a + c, a + d, b + c, b + d], [y, x + e, x + f, (a + c) + e, \ldots, (b + d) + f]\}
\]

- Thus: not used in practice
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

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

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

Example

\[ x_0 := 0 \]
\[ y_0 := 0 \]

\[ x_1 := \phi_2(x_2, x_0) \]
\[ y_1 := \phi_2(y_2, y_0) \]

\[ x_2 := x_1 + 1 \]
\[ y_2 := y_1 + 1 \]
The AWZ Algorithm

Example

\[ \phi_2 \]

\[ x_1 \]

\[ x_2 \]

\[ x_0 \]

\[ 1 \]

\[ + \]

\[ 0 \]

\[ y_1 \]

\[ y_2 \]

\[ y_0 \]

\[ 1 \]
The AWZ Algorithm

Example
Kildall compared to AWZ

\[
\begin{align*}
    a_0 & := 2 \\
    x_0 & := a_0 + 1 \\

    a_2 & := \phi_4(a_0, a_1) \\
    x_2 & := \phi_4(x_0, x_1) \\
    y_0 & := a_2 + 1
\end{align*}
\]
Kildall compared to AWZ

\[ y = \phi_4 a_2 x_0 + \phi_4 x_2 \]

\[ = \phi_4 (a_0 + 3 a_1) \]

\[ + x_1 \]

Diagram: 

- Node 2: \( \phi_4 \) with input \( a_0 \) and output \( a_2 \)
- Node 3: \( \phi_4 \) with input \( x_0 \) and output \( x_2 \)
- Node 1: Binary addition with inputs \( x_0 \) and \( x_1 \)
- Node 4: Binary addition with inputs \( a_2 \) and \( x_2 \)

\[ y = \phi_4 (a_0 + 3 a_1) + x_1 \]
Kildall compared to AWZ

\[ y = \phi_4 x_0 x_2 + a_2 \phi_4 x_2 + x_0 a_1 x_1 + a_0 2 \]