Script generated by TTT

Title: Seidl: Programmoptimierung (04.12.2013)

Date: Wed Dec 04 08:32:32 CET 2013

Duration: 87:34 min

Pages: 24

Theorem

Let $x_i \sqsupseteq f_i\,(x_1, \ldots, x_n)$, $i = 1, \ldots, n$ denote a constraint system over the complete lattice $\mathbb{D}$ of hight $h > 0$.

(1) The algorithm terminates after at most $h \cdot N$ evaluations of right-hand sides where

$$N = \sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \qquad // \text{ size of the system } :\text{-)}$$

(2) The algorithm returns a solution.
If all f_i are monotonic, it returns the least one.

Proof:

Ad (1):

Every unknown x_i may change its value at most h times :-)

Each time, the list $I[x_i]$ is added to W .

Thus, the total number of evaluations is:

$$\begin{aligned} &\leq\ n + \textstyle\sum_{i=1}^{n} h \cdot \#\,(I[x_i]) \\ &=\ n + h \cdot \textstyle\sum_{i=1}^{n} \#\,(I[x_i]) \\ &\cancel{=}\ n + h \cdot \textstyle\sum_{i=1}^{n} \#\,(Dep\ f_i) \\ &\leq\ h \cdot \textstyle\sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \\ &=\ h \cdot N \end{aligned}$$

N

Ad (2):

We only consider the assertion for monotonic f_i .

Let D_0 denote the least solution. We show:

- $D_0[x_i] \sqsupseteq D[x_i]$ (all the time)
- $D[x_i] \not\sqsupseteq f_i\ \mathsf{eval} \implies x_i \in W$ (at exit of the loop body)
- On termination, the algo returns a solution :-))

Discussion:

- In the example, fewer evaluations of right-hand sides are required than for RR-iteration :-)
- The algo also works for non-monotonic f_i :-)
- For monotonic f_i, the algo can be simplified:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{;}$$

- In presence of widening, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t = D[x_i] \mathbin{\sqcup\mkern-12mu-} t;}$$

- In presence of Narrowing, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t = D[x_i] \mathbin{\sqcap\mkern-12mu-} t;}$$

Ad (2):

We only consider the assertion for monotonic f_i .

Let D_0 denote the least solution. We show:

- $D_0[x_i] \sqsupseteq D[x_i]$ (all the time)
- $D[x_i] \not\sqsupseteq f_i$ eval $\Longrightarrow x_i \in W$ (at exit of the loop body)
- On termination, the algo returns a solution :-))

Example:

$$\begin{array}{lcl} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

$D[x_1]$	$D[x_2]$	$D[x_3]$	W
$\emptyset$	$\emptyset$	$\emptyset$	$\boxed{x_1}, x_2, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_2}, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_3}$
$\{a\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_1}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_3}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_2}$
$\{a, c\}$	$\{a\}$	$\{a, c\}$	$[\,]$

Warning:

- The algorithm relies on explicit dependencies among the unknowns.

  So far in our applications, these were obvious. This need not always be the case :-(
- We need some strategy for extract which determines the next unknown to be evaluated.
- It would be ingenious if we always evaluated first and then accessed the result ... :-)

$\Longrightarrow$ recursive evaluation ...

Idea:

- → If during evaluation of f_i , an unknown x_j is accessed, x_j is first solved recursively. Then x_i is added to $I[x_j]$:-)

$$
\begin{aligned}
\text{eval } x_i \; x_j \quad = \quad & \text{solve } x_j; \\
& I[x_j] = I[x_j] \cup \{x_i\}; \\
& D[x_j];
\end{aligned}
$$

- → In order to prevent recursion to descend infinitely, a set *Stable* of unknown is maintained for which solve just looks up their values :-)

  Initially, $\mathit{Stable} = \emptyset$ ...

The Function solve :

$$
\begin{aligned}
\text{solve } x_i \quad = \quad & \text{if } (x_i \notin \mathit{Stable}) \; \{ \\
& \quad \mathit{Stable} = \mathit{Stable} \cup \{x_i\}; \\
& \quad t = f_i \, (\text{eval } x_i); \\
& \quad t = D[x_i] \sqcup t; \\
& \quad \text{if } (t \neq D[x_i]) \; \{ \\
& \qquad W = I[x_i]; \quad I[x_i] = \emptyset; \\
& \qquad D[x_i] = t; \\
& \qquad \mathit{Stable} = \mathit{Stable} \backslash W; \\
& \qquad \text{app solve } W; \\
& \quad \} \\
& \}
\end{aligned}
$$

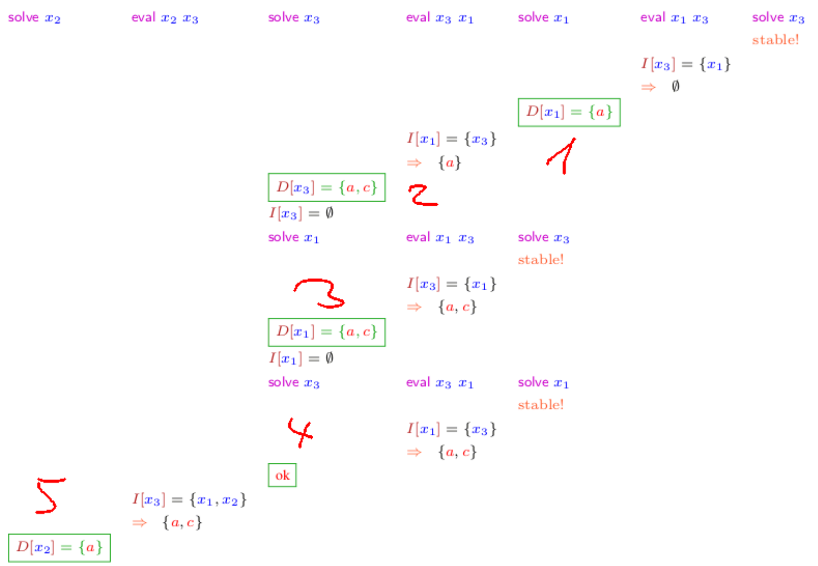
Example:

Consider our standard example:

$$
\begin{aligned}
x_1 & \supseteq \{a\} \cup x_3 \\
x_2 & \supseteq x_3 \cap \{a, b\} \\
x_3 & \supseteq x_1 \cup \{c\}
\end{aligned}
$$

A trace of the fixpoint algorithm then looks as follows:

solve x_2	eval x_2 x_3	solve x_3	eval x_3 x_1	solve x_1	eval x_1 x_3	solve x_3
						stable!
					$I[x_3] = \{x_1\}$	
					$\Rightarrow \emptyset$	
				$\boxed{D[x_1] = \{a\}}$		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \{a\}$			
		$\boxed{D[x_3] = \{a, c\}}$				
		$I[x_3] = \emptyset$				
		solve x_1	eval x_1 x_3	solve x_3		
				stable!		
			$I[x_3] = \{x_1\}$			
			$\Rightarrow \{a, c\}$			
		$\boxed{D[x_1] = \{a, c\}}$				
		$I[x_1] = \emptyset$				
		solve x_3	eval x_3 x_1	solve x_1		
				stable!		
			$I[x_1] = \{x_3\}$			
			$\Rightarrow \{a, c\}$			
		$\boxed{\text{ok}}$				
	$I[x_3] = \{x_1, x_2\}$					
	$\Rightarrow \{a, c\}$					
$\boxed{D[x_2] = \{a\}}$						



→ Evaluation starts with an interesting unknown x_i (e.g., the value at *stop*)

→ Then automatically all unknowns are evaluated which influence x_i :-)

→ The number of evaluations is often smaller than during worklist iteration ;-)

→ The algorithm is more complex but does not rely on pre-computation of variable dependencies :-))

→ It also works if variable dependencies during iteration change !!!

$\Longrightarrow$ interprocedural analysis

Warning II:

- The recursive algorithm may not evaluate right-hand sides atomicly.
- Evaluations of right-hand sides may be continued which have been started with out-dated data. $\Longrightarrow$ in some cases, it may fail to determine the least solution !?!

Idea:

- Identify outdated computations ...
- Abort !!

Idea (cont.):

→ Record when evaluation of a variable has started by means of a set *Called*.

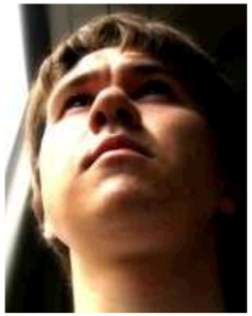
→ Whenever during evaluation of a rhs f_i, we detect that no longer $x_i \in \mathit{Called}$, we abort ...

$$
\begin{array}{lll}
\mathsf{eval}\ x_i\ x_j & = & \mathsf{solve}\ x_j; \\
 & & \mathsf{if}\ (x_i \notin \mathit{Called})\ \mathsf{raise}\ \mathbf{Abort}; \\
 & & I[x_j] = I[x_j] \cup \{x_i\}; \\
 & & D[x_j];
\end{array}
$$

→ Initially, $\mathit{Called} = \emptyset$ ...

The new Function solve :

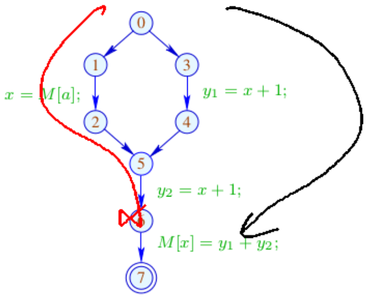
$$
\begin{array}{lll}
\textsf{solve } x_i & = & \textsf{if } (x_i \notin \mathit{Stable}) \ \{ \\
& & \quad \mathit{Stable} = \mathit{Stable} \cup \{x_i\}; \mathit{Called} = \mathit{Called} \cup \{x_i\}; \\
& & \quad \textsf{try } \{ \quad t = f_i \ (\textsf{eval } \ x_i); \ t = D[x_i] \sqcup t; \\
& & \qquad \mathit{Called} = \mathit{Called} \backslash \{x_i\}; \\
& & \qquad \textsf{if } (t \neq D[x_i]) \ \{ \\
& & \qquad\quad W = I[x_i]; \quad I[x_i] = \emptyset; \\
& & \qquad\quad D[x_i] = t; \\
& & \qquad\quad \mathit{Stable} = \mathit{Stable} \backslash W; \\
& & \qquad\quad \textsf{app solve } W; \\
& & \qquad \} \ \} \quad \textsf{with } \mathbf{Abort} \to (); \\
& & \}
\end{array}
$$



Aleks Karbyshev, TU München :-))

1.7 Eliminating Partial Redundancies

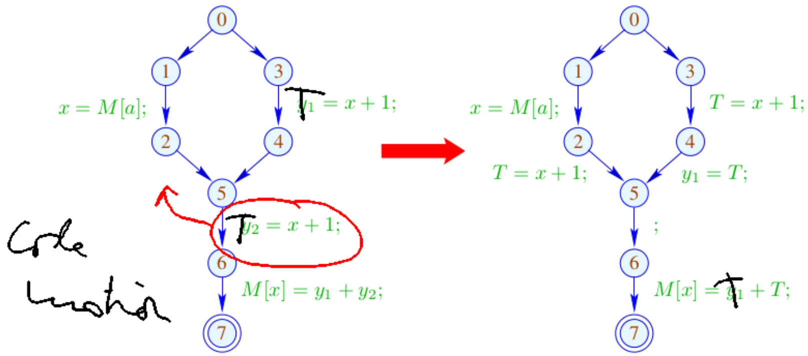
Example:



// $x + 1$ is evaluated on every path ...

// on one path, however, even twice :-(

Goal:



Idea:

(1) Insert assignments $T_e = e;$ such that e is available at all points where the value of e is required.

(2) Thereby spare program points where e either is already available or will definitely be computed in future.

Expressions with the latter property are called very busy.

(3) Replace the original evaluations of e by accesses to the variable T_e.

$\Longrightarrow$ we require a novel analysis :-))

An expression e is called busy along a path π , if the expression e is evaluated before any of the variables $x \in \mathit{Vars}(e)$ is overwritten.

// backward analysis!

e is called very busy at u , if e is busy along every path $\pi : u \to^* \mathit{stop}$.

An expression e is called busy along a path π , if the expression e is evaluated before any of the variables $x \in \mathit{Vars}(e)$ is overwriten.

// backward analysis!

e is called very busy at u , if e is busy along every path $\pi : u \to^* \mathit{stop}$.

Accordingly, we require:

$$\mathcal{B}[u] = \bigcap \{ [\![\pi]\!]^\sharp \, \emptyset \mid \pi : u \to^* \mathit{stop} \}$$

where for $\pi = k_1 \ldots k_m$:

$$[\![\pi]\!]^\sharp = [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

Our complete lattice is given by:

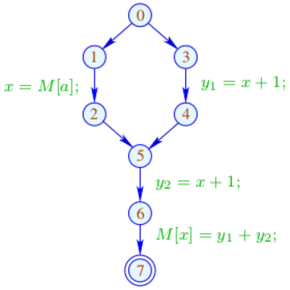
$$\mathbb{B} = 2^{\mathit{Expr} \setminus \mathit{Vars}} \qquad \text{with} \qquad \sqsubseteq \; = \; \supseteq$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, \mathit{lab}, v)$ only depends on lab , i.e., $[\![k]\!]^\sharp = [\![\mathit{lab}]\!]^\sharp$ where:

$$\begin{array}{lll} [\![;]\!]^\sharp \, B & = & B \\ [\![\mathit{Pos}(e)]\!]^\sharp \, B & = & [\![\mathit{Neg}(e)]\!]^\sharp \, B \quad = \quad B \cup \{e\} \\ [\![x = e;]\!]^\sharp \, B & = & (B \backslash \mathit{Expr}_x) \cup \{e\} \\ [\![x = M[e];]\!]^\sharp \, B & = & (B \backslash \mathit{Expr}_x) \cup \{e\} \\ [\![M[e_1] = e_2;]\!]^\sharp \, B & = & B \cup \{e_1, e_2\} \end{array}$$

These effects are all distributive. Thus, the least solution of the constraint system yields precisely the MOP — given that *stop* is reachable from every program point :-)

Example:



7	$\emptyset$
6	$\{y_1 + y_2\}$
5	$\{x + 1\}$
4	$\{x + 1\}$
3	$\{x + 1\}$
2	$\{x + 1\}$
1	$\emptyset$
0	$\emptyset$