Lecture notes – Kildall's algorithm

September 12, 2022

Slide 2 (of 62). Say an abstract interpretation framework (D, f_D, γ_D) is a *consistent abstraction* of another framework (C, f_C, γ_C). By definition, for any c_0, d_0 such that $\gamma_{DC}(d_0) \geq c_0$, $\overline{\gamma}(\text{JOP}_{\overline{D}}) \geq \text{JOP}_{\overline{C}}$. Therefore, if we can compute some over-approximation of the $\text{JOP}_{\overline{D}}$ (as shown by the "*" in the figure), then the γ image of this over-approximation would be an over-approximation of $\text{JOP}_{\overline{C}}$. The aim of Kildall's algorithm is to compute such an over-approximation of $\text{JOP}_{\overline{D}}$.

Slide 3. This slide shows Kildall's algorithm.

Slides 4-15. These slides illustrate Kildall's algorithm on an example program, using the odd-even lattice. The most-precise transfer functions for this lattice are used. Even though the transfer function for "+" is *not* distributive, in this example the algorithm happens to compute the precise JOP.

Slides 16-26. These slides illustrate Kildall's algorithm on another example program, this time using the CP lattice. The most-precise transfer functions for this lattice are used. In this example the algorithm happens to over-approximate the precise JOP.

Slides 21, 22 illustrate how the *early joining* at the point before Node 3 (which has a non-distributive transfer function) makes it impossible to infer the fact '(y ,1)' after Node 3. The abstract JOP, on the other hand, performs *late joining*, using the results of all paths that come to each point. However, in general, the abstract JOP is not computable.

Slide 28. This slide is about proving the termination of Kildall's algorithm when the lattice D has no infinite chains. Let $\overline{d_i}$ be the vector such that the Nth element of this vector is the dataflow value at program point N after i steps (i.e., iterations) of Kildall's algorithm. Consider step $i + 1$ of the algorithm. The value v that's chosen at this step for propagation gets unmarked. Now, there are two possibilities. Case 1: $\overline{d}_{i+1} > \overline{d}_i$. Case 2: $\overline{d}_{i+1} = \overline{d}_i$. Case 2 can happen only when no successor value of v changes. That is, none of the successor values of v that was previously unmarked gets marked. That is, the total number of marks *decreases*.

The plot in the slide illustrates the above scenarios. In the x-axis are the steps of the algorithm. The y-axis denotes elements of $\overline{D}$. At each value i of the x-axis the point shown in the figure denotes $\overline{d_i}$. Thus, if the points corresponding to steps i and $i+1$ are at the same height then we are in Case 2, else we are in Case 1.

Since each $\overline{d}_{i+1}$ is $\geq \overline{d}_i$, the points in the figure (not counting duplicated points in "horizontal" portions) constitute a chain of the lattice chain in $\overline{D}$. It can be shown that if the lattice D has only finite chains, then $\overline{D}$ also contains only finite chains. Therefore, the points in the figure cannot climb an infinite number of times. Furthermore, each contiguous horizontal sequence has at most M points, where M is the number of program points. Therefore, the algorithm is guaranteed to terminate on all programs whenever D has no infinite chains.

Note, on some programs the algorithm could terminate even in the presence of infinite chains. In particular, the algorithm always terminates on loop-free programs. Also, a technique called *widening* can be used to ensure termination (with over-approximation of results) in the presence of infinite chains. Widening is not in the syllabus of our course.

Slide 29-30. Any program P induces a set of *dataflow equations*. These equations are mathematical equations, involving mathematical variables. The *domains* of the variables are the given lattice D; i.e., any solution to these equations maps each variable to an element of D. Each equation has a variable on the LHS, and an expression involving zero or more variables or specific lattice elements on the rhs.

The variables in the equations correspond to the program points. Each variable x_N, corresponding to program point N, appears in the lhs of atmost one equation, as described below.

- If point N is the initial program point I of the program, the equation is simply '$x_I = d_0$', where d_0 is the given initial dataflow value.
- If point N is the successor of a point M, and the node n that is in between M and N is not a join node, then the equation is '$x_N = f_n(x_M)$'.
- Finally, if N comes after a join node, and L and M are the points that precede the join node, the equation is '$x_N = x_L \sqcup x_M$'.

Note that the equations are derived from the program P, the given initial value d_0, as well as the given transfer functions.

The idea behind forming these equations is as follows: Rather than compute the JOP directly, Kildall's algorithm attempts to find a solution to these equations. We will later on relate this solution to the JOP.

Slides 32-33. In this example we use the collecting semantics lattice as the domain for the variables, and the concrete (i.e., *nstate′* transfer functions). Let d_0 be any set of (initial) concrete states. It turns out there are *two* solutions to the induced equations:

1. $x_A = d_0$, $x_B = \{2\}, x_C = \{i | i \geq 2\}$, $x_D = \{i | i \geq 3\}$.
2. $x_A = d_0$, $x_B = \{2\}, x_C = \mathbb{Z}, x_D = \mathbb{Z}$

There are no other solutions to the induced equations. Note that the first solution above corresponds to the collecting semantics of the program, and is dominated by the second solution. In fact, the collecting semantics will *always* be the least solution to the induced equations if we use the collecting-semantics lattice and transfer functions. (We will prove this later.)

Slides 34-35. Consider any set of equations such that (a) the lhs of each equation is a variable, and the rhs is an expression involving zero or more variables, as well as operators and functions, and (b) each variable appears in the lhs of a unique equation. These equations could be the induced equations from a program, or somehow otherwise given. Let D be the domain of the variables.

The equations induce a function $\overline{f}$ from D^M to D^M, where M is the number of variables. (We also denote D^M as $\overline{D}$.) If the equations are induced by a program, this function $\overline{f}$ is called the induced *vectorized* transfer function of the program.

The definition of this function is given in the slide. The input to the function is a vector $(d_I, d_B, \ldots, d_M)$. The function returns a vector; the ith element of the returned vector is nothing but the rhs of the dataflow equation whose lhs is x_i, with each variable x_j occurring in this expression replaced by d_j.

It is easy to see that an element of the domain $\overline{D}$ is a solution to the equations iff it is a fix-point of $\overline{f}$.

If D is a complete lattice then so is $\overline{D}$. Also, if all the operators and functions that appear in the rhs's of the dataflow equations are monotone, then so is the function $\overline{f}$. Therefore, the Knaster-Tarski theorem applies, and tells us that $\overline{f}$ (a) has at least one fix-point, and (b) has a unique least fix-point. (This is the relevance of the Knaster-Tarski theorem to program analysis!)

The *Kleene iteration* is the procedure of generating the ascending chain $\bot_{\overline{D}}, \overline{f}(\bot_{\overline{D}}), \overline{f}^2(\bot_{\overline{D}}), \ldots$. It is easy to prove that if this chain stabilizes at some value then that value is the lfp of $\overline{f}$. It is also easy to prove the this chain will stabilize if $\overline{D}$ has no infinite chains.

We prove later on that Kildall's algorithm, whenever it terminates, identifies the lfp of the set of equations given to it (or equivalently, the lfp of the function $\overline{f}$ induced by these equations). It is an interesting fact that Kildall's algorithm need not be applied to program analysis alone. It can be used to find the lfp of any set of equations that have the structure described above, or equivalently, the lfp of any function $\overline{f}$ of the form described above provided $\overline{f}$ is monotone.

Slide 36. This figure summarizes the results that we are going to prove subsequently.

Slide 37 – Definitions. We are already aware of the definitions of monotonic and distributive functions. It is easy to show every distributive function is also monotone.

An *infinitely distributive* function f is one such that for any subset S of the lattice D, $f(\sqcup S) = \sqcup\{f(s) | s \in S\}$, where S is of finite or infinite size. Note that if D is a finite lattice then every distributive function on it is also infinitely distributive. Otherwise, every infinitely distributive function is distributive, but the converse does not hold necessarily.

Slides 38-41. Let (D, f_D, γ_D) be an abstract interpretation framework, and let $d_0 \in D$ be an initial value. Let $\overline{f} : \overline{D} \rightarrow \overline{D}$ be the induced vectorized transfer function of a program P wrt initial value d_0. Let $\overline{c} \in \overline{D}$ be any fix-point of $\overline{f}$. Let $\overline{d} \in \overline{D}$ be the abstract JOP at all points of program P using d_0 as the initial value. We prove now that $\overline{d} \leq \overline{c}$. We denote $\overline{c}[M]$ as c_M, $\overline{c}[N]$ as c_N, etc.

First, we prove using induction on the lengths of program paths, the following claim: For any path p in the program that ends at a point N, $f_p(d_0)$ is dominated by c_N.

The base case is the path of length 0. For this path, $f_p(d_0)$ is d_0. $\overline{c}[I]$, where I is the initial point in the program, is also equal to d_0, because of the equation $x_I = d_0$.

Now, consider a path p of length $i + 1$ that ends at a point N; also let p be equal to a shorter path p' of length i that ends at a point M, followed by a transition through a node n that lies between M and N. Let $d \equiv f_p(d_0)$, $d' \equiv f_{p'}(d_0)$, and $c_M \equiv \overline{c}[M]$.

Consider the case where node n is a non-join node. By the inductive hypothesis:

$d' \leq c_M$ (1)

By definition of d, d':

$d = f_{MN}(d')$ (2)

Program points M, N induce a dataflow equation $x_N = f_{MN}(x_M)$. Since $\overline{c}$ is a solution to the equations, it follows that:

$c_N = f_{MN}(c_M)$ (3)

Now, by equations 1-3 above, and by monotonicity of f_{MN}, it follows that $d \leq c_N$.

Consider the other case, where node n is a join node. In this case, since n has an *id* transfer function, it follows that:

$d' = d$ (1)

Let M, L be the points that precede n. The dataflow equation induced by n is '$x_N = x_M \sqcup x_L$'. Since $\overline{c}$ is a solution to the equations, it follows that $c_N = c_M \sqcup c_L$. That is:

$c_M \leq c_N$ (2)

By the inductive hypothesis:

$d' \leq c_M$ (3)

From equations 1-3 above it follows that $d \leq c_N$.

Hence we have proved our claim.

It follows from the above claim that the dataflow facts that arrive at any point N due to any path ending at N are all dominated by c_N. Therefore, it follows that $\overline{d}[N] \leq c_N$. Therefore, any technique (e.g., Kildall's algorithm, or Kleene iteration) that computes *any* fix-point of $\overline{f}$ can be used to safely over-approximate the JOP $\overline{d}$.

Slides 42-45. We continue to use the terms introduced above. We now prove that if the dataflow functions of all the edges in a program are infinitely distributive, then the abstract JOP $\overline{d}$ is a fix-point of the function $\overline{f}$ induced by the program. It follows from this result and from the previously proved result that the lfp of $\overline{f}$ coincides with the JOP $\overline{d}$.

Let N be any arbitrary program point. We will argue, using two cases below, that $\overline{d}$ satisfies the data flow equation who left hand side is the variable x_N.

The first case is that N is preceded by a non-join node. Let M be the point that precedes this node. Therefore, the data flow equation whose lhs is x_N is "$x_N = f_{MN}(x_M)$". Let S_M be the (potentially infinite) set of abstract states (i.e., elements of D) that arrive at M along paths that end at M by starting with the abstract state d_0. Similarly, let S_N be the (potentially infinite) set of abstract states that arrive at N along paths that end at N by starting with the abstract state d_0. It is clear that $S_N = \{f_{MN}(s) | s \in S_M\}$ (see Slide 43). Therefore, because f_{MN} is infinitely distributive, it follows that $(\sqcup S_N) = f_{MN}(\sqcup S_M)$. Now, by definition of JOP, $(\sqcup S_M) = \overline{d}[M]$ and $(\sqcup S_N) = \overline{d}[N]$. Therefore, we have shown that $\overline{d}[N] = f_{MN}(\overline{d}[M])$. In other words, $\overline{d}$ satisfies the equation mentioned above.

Going on to the case where N is preceded by a join node, let M, P be the points that precede this join-node. In this case, the data flow equation whose lhs is x_N is "$x_N = x_M \sqcup x_P$". It can be argued (see Slide 44) that $\overline{d}[N] = \overline{d}[M] \sqcup \overline{d}[P]$. Therefore, the $\overline{d}$ satisfies the equation mentioned above.

Therefore, we have shown that $\overline{d}$ satisfies all the dataflow equations induced by the program P. Therefore, $\overline{d}$ is a fix-point of $\overline{f}$.

Slide 46.

A note about the collecting semantics: Let (C, f_C) be the collecting-semantics framework. That is, C is the collecting-semantics lattice, and f_C is the set of *nstate′* transfer functions. C, being a power set lattice, is a complete lattice. The *nstate′* functions can all be shown to be distributive. Therefore, the JOP using this framework, which is nothing but the collecting semantics, is the lfp of the dataflow equations built using the *nstate′* transfer functions.

Slides 47-55. In these slides we show that at any point in the algorithm the vector of values $\overline{d}$ computed so far by the algorithm is dominated by the lfp $\overline{l}$ of $\overline{f}$. We do this using induction on the number of steps completed by the algorithm. $\overline{d}^0$ has d_0 in the I^{th} slot and $\perp$ in all other slots (this is the algorithm's initialization). From the equations, it follows that any fix point assigns d_0 to variable x_I. Therefore, the base case is proved.

We now assume that $\overline{d}^i \sqsubseteq \overline{l}$, and prove that $\overline{d}^{i+1} \sqsubseteq \overline{l}$. Let N be the program point whose value gets updated at Step $i + 1$. The argument proceeds in two cases.

The first case is that the node preceding N is a non-join node. Let M be the program point that precedes this node. Let d^i_M denote $\overline{d}^i[M]$, d^i_N denote $\overline{d}^i[N]$, l_M denote $\overline{l}[M]$, l_N denote $\overline{l}[N]$, etc.. By the inductive hypothesis:

$d^i_M \leq l_M$ (1)

$d^i_N \leq l_N$ (2)

Points M, N induce the dataflow equation '$x_N = f_{MN}(x_M)$'. Therefore, since $\overline{l}$ is a solution to the equations, we get:

$l_N = f_{MN}(l_M)$ (3)

From equations 1 and 3, and by monotonicity of f_{MN}, we get:

$f_{MN}(d^i_M) \leq l_N$ (4)

Clearly the value d^{i+1}_N at point N after step $i + 1$ is $d^i_N \sqcup f_{MN}(d^i_M)$. From equations 2 and 4 it follows that

$d^{i+1}_N \leq l_N$ (5)

All other components of vector $\overline{d}$ (other than the Nth component) remain the same after $i+1$ steps as they were at the end of i steps. Therefore, we have shown that $\overline{d}^{i+1}$ is dominated by $\overline{l}$.

The next case is that N is a join node. This case is explained in Slide 52-55.

Slides 56-57.

We now show that $\overline{d}$ is a post fix-point of $\overline{f}$.

Let N be any program point. We need to show that $(d_N = \overline{d}[N]) \sqsubseteq \overline{f}(\overline{d})[N]$. The argument proceeds in two cases.

The first case is that the node that precedes N is a non-join node. Let M be the point that precedes this node. Say the dataflow value at M got unmarked for the last time in step i of the algorithm (it must have gotten unmarked in some step because the algorithm has terminated and initially all points were marked). Clearly, since M never got marked again,

$(d_M \equiv \overline{d}[M]) = d^i_M$ (1)

Now, at the end of step i the value at point N becomes $d^i_N \equiv (f_{MN}(d^i_M) \sqcup d^{i-1}_N)$. Therefore, it follows that

$d^i_N \geq f_{MN}(d^i_M)$ (2)

The value at any program point can change in a step only if a predecessor of this point gets unmarked at this step. Since M is the only predecessor of N, and since we have assumed that M never gets unmarked after step i, the value at N does not change after Step i. Therefore,

$d_N = d^i_N$ (3)

Due to the equation $x_N = f_{MN}(x_M)$, it follows that

$(\overline{f}(\overline{d}))[N] = f_{MN}(d_M)$ (4)

From (1), (2), (3), and (4), it follows that

$d_N \geq (\overline{f}(\overline{d}))[N]$ (5)

The other case is that the node that precedes point N is a join node. Let M and P be the program points that precede this join node. Wolog, assume that M gets unmarked for the last time at a step i and P gets unmarked for the last time at a later step

j ($j > i$). After Step i the value at N must dominate the final value at M, i.e., must dominate d_M. In any subsequent steps after step i the value at N can only move upwards in the lattice (this is true of every step in the algorithm, due to the joining at each step). Finally, after step j, the value at N must dominate the final value at P, namely, d_P. Therefore, the final value at N, i.e., d_N, dominates d_M as well as d_P. In other words,

$$d_N \geq d_M \sqcup d_P \tag{6}$$

Due to the equation $x_N = x_M \sqcup x_P$, it is clear that $(\overline{f}(\overline{d}))[N]$ is equal to $d_M \sqcup d_P$. This, in conjunction with (6) above, gives us:

$$d_N \geq (\overline{f}(\overline{d}))[N] \tag{7}$$

From (5) and (7), it follows that $\overline{d} \geq \overline{f}(\overline{d})$. Therefore, we have shown that $\overline{d}$ is a post fix-point of $\overline{f}$.

Slide 58.

As per Knaster-Tarski theorem, the least fix-point is dominated by all post fix-points. Therefore, it follows that $\overline{d} \geq \overline{l}$.

This, in conjunction with our previous result that $\overline{d} \leq \overline{l}$ gives us our final result that $\overline{d} = \overline{l}$.

Slides 60-62. These are a summary of the material discussed so far.