SETL Newsletter No. 208

Micha Sharir

April 7, 1978

A Few Cautionary Notes on the Convergence of Iterative Data-Flow Analysis Algorithms

In his paper "Iterative algorithms for global flow analysis," R. E. Tarjan describes a hierarchy of data-flow analysis frameworks, with an iterative algorithm to solve them. Tarjan defines a k-*bounded* data-flow framework as a framework (L,F) (where L is a semilattice of data information, and F is a space of information-propagating maps) in which for each $f \in F$, $x \in L$

$$f^k(x) \geq x \wedge f(x) \wedge \ldots \wedge f^{k-1}(x) \wedge f(1)$$

where 1 is the largest element of L. He also defines a ks-*bounded* framework as a framework in which for each $f \in F$, $x \in L$,

$$f^k(x) \geq x \wedge f(x) \wedge \ldots \wedge f^{k-1}(x)$$

and, in addition, the associated data-flow problem involves data-propagation from the entry block only.

For example, available expressions analysis is 1-bounded (but not 1s-bounded). Indeed, in this case L consists of sets of available expressions and the meet is set-intersection. Each $f \in F$ can be represented as a pair (GEN,KILL) where GEN is the set of expressions which are unconditionally generated (without being later killed) during the propagation described by f, and KILL is the set of expressions which might get killed (and not recomputed later) during that propagation, and for each $\alpha \in L$

$$f(\alpha) = GEN \cup (\alpha \cap KILL^c)$$

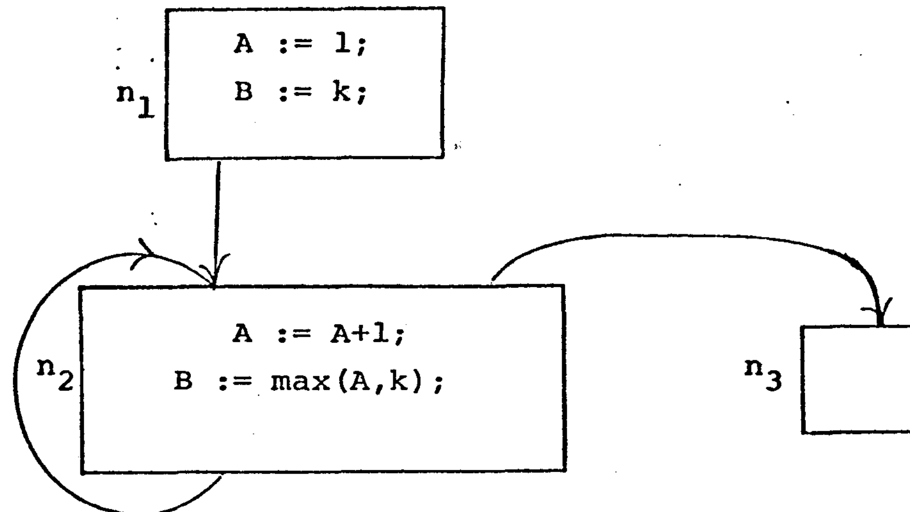
so that

$$f(\alpha) \geq \alpha \cap (GEN \cup KILL^c) = \alpha \wedge f(1)$$

Unfortunately, if we examine the major optimizations to be performed in our optimizer, we arrive at the following results:

Proposition 1. Constant propagation is not ks-bounded for any $k \geq 1$.

Proof: Let $k \geq 1$ be given, and consider the following code:



The standard semilattice L in constant propagation is the set of all partially defined single-valued maps from the set of program variables to the set of values, with set-intersection as a meet. Let $f \in F$ represent the effect of propagation along the loop starting and ending at the start of n_2. Then, $\forall \alpha \in L$,

$$f(\alpha)(A) = \alpha(A) + 1$$
$$f(\alpha)(B) = \max(\alpha(A)+1,\ k)$$
$$f(\alpha)(V) = \alpha(V) \qquad \text{for all other variables } V.$$

Now, let $\alpha = \{[A,1],\ [B,k]\}$ which is the result of propagating from the start of n_1 to the start of n_2. Then it is easy to see that

$$\begin{aligned} f(\alpha) &= \{[A,2],\ [B,k]\} \\ f^2(\alpha) &= \{[A,3],\ [B,k]\} \\ &\vdots \\ f^{k-1}(\alpha) &= \{[A,k],\ [B,k]\} \\ f^k(\alpha) &= \{[A,k+1],\ [B,k+1]\} \end{aligned}$$

so that

$$f^k(\alpha) \not\geq \bigwedge_{j=0}^{k-1} f^j(\alpha) = \{[B,k]\}\ .$$

Q.E.D.

<u>Remark</u>. In the above example, if we perform constant propagation by an iterative propagation algorithm of Kildall's type, we arrive, as an <u>under-estimation</u>, at the conclusion that B is not

constant, but come to this conclusion in a relatively small number of iterations as follows:

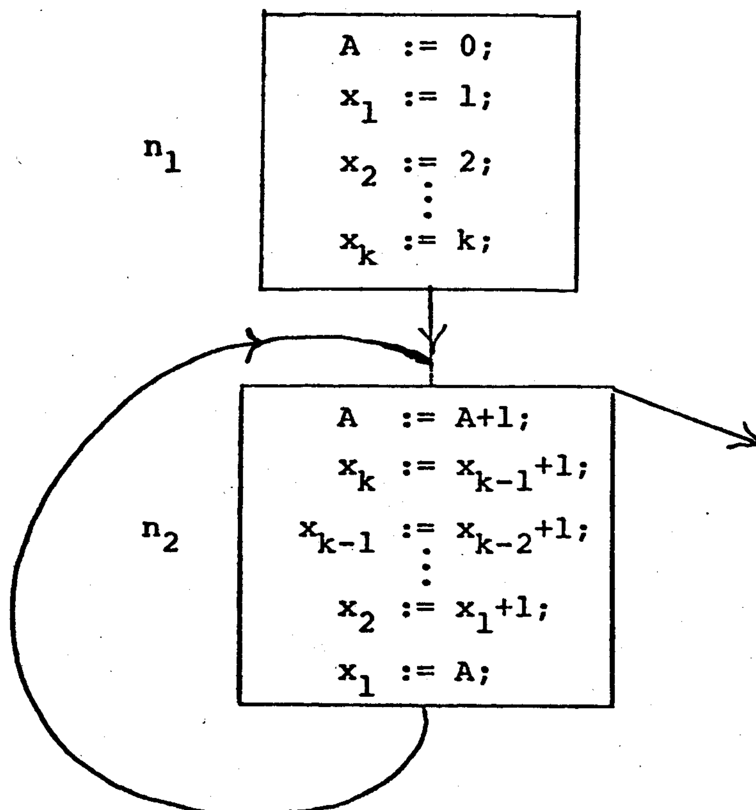
Let $\alpha(n)$ denote the lattice information, gathered so far at the start of node n. Initially $\alpha(n_1) = \emptyset$; . Then, propagating

from	to	yields
n_1	n_2	$\alpha(n_2) = \{[A,1],[B,k]\}$
n_2	n_2	$\alpha(n_2) = \{[B,k]\}$
n_2	n_2	$\alpha(n_2) = \emptyset$

(in the third propagation, the value of A is unknown upon the beginning of the loop, so that the value of B also becomes unknown). Thus, the algorithm stabilizes after two iterations. However, we have

Proposition 2. The number of iterations required by an iterative algorithm of Kildall's type to perform constant propagation, cannot be bounded by a function of the graph-theoretic parameters of the flow graph.

Proof: Let $k \geq 1$ be given. The following code, whose flow graph is independent of k, requires k+1 iterations to achieve stabilization.



Here, after j iterations through the loop, we have:

$$j = 0 \quad \alpha(n_2) = \{ [A,0], [x_1, 1], \ldots, [x_k, k] \};$$
$$j = 1 \quad \alpha(n_2) = \{ [x_1, 1], [x_2, 2], \ldots, [x_k, k] \};$$
$$j = 2 \quad \alpha(n_2) = \{ [x_2, 2], \ldots, [x_k, k] \};$$
$$\vdots$$
$$j = k \quad \alpha(n_2) = \{ [x_k, k] \};$$
$$j = k+1 \quad \alpha(n_2) = \emptyset$$

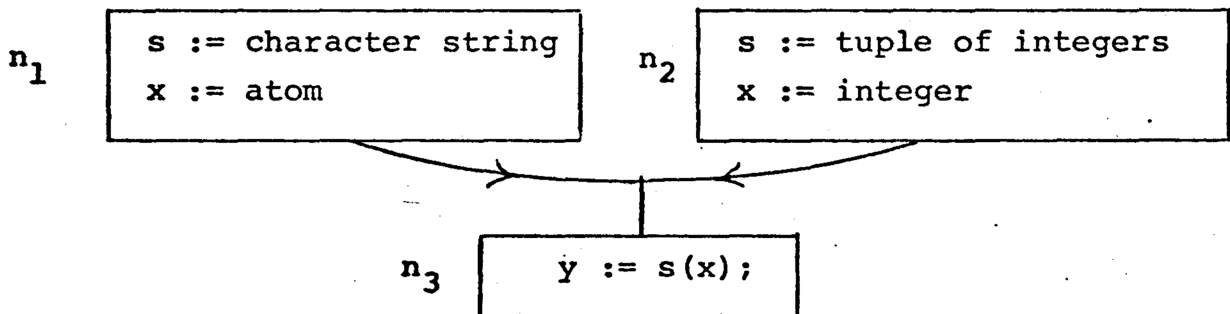
Q.E.D.

Remark. Constant propagation is not a typical case for Tarjan's analysis, since he assumes that the data-flow framework is distributive, which is not the case for constant propagation. We have used constant propagation as an introductory example, showing what can happen in the class of attribute flow analyses. It will be shown below that similar counterexamples also exist for other flow analyses in this class.

It should first be noted that the standard frameworks of many attribute flow analyses are not distributive. As an important example, we have

Lemma 3. The standard framework of type finding analysis is not distributive.

Proof: Consider the following code:



Let

$$\alpha_1 = \{ [s,char], [x,atom] \}$$

and

$$\alpha_2 = \{ [s, tuple of ints], [x, int] \}$$

denote the information propagated to n_3 from n_1 and n_2 respectively, and let f denote the effect of propagation through n_3. Then

$$f(\alpha_1)(y) = error$$

$$f(\alpha_2)(y) = int$$

and

$$f(\alpha_1 \vee \alpha_2)(y) = int \vee char \neq int = f(\alpha_1)(y) \vee f(\alpha_2)(y)$$

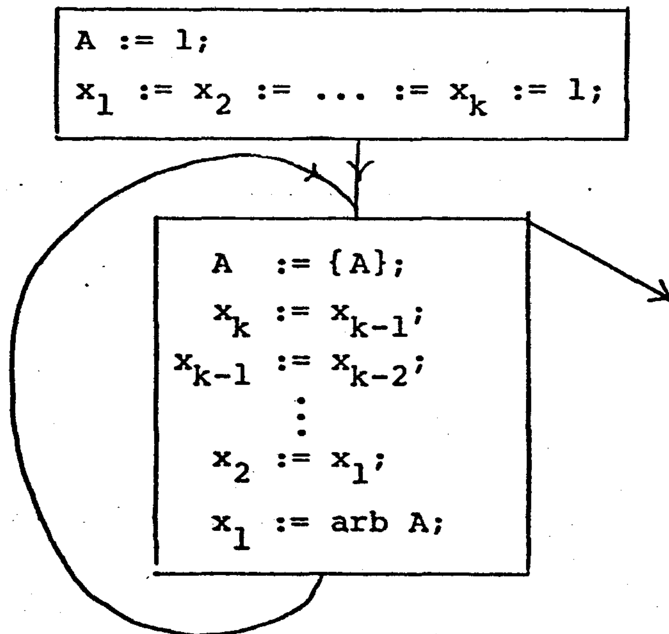
so that the framework is not distributive.

Q.E.D.

Proposition 3. (a) Type finding analysis is not ks-bounded for any $k \geq 1$.

(b) The number of iterations required by an iterative algorithm of Kildall's type to perform type analysis cannot be bounded by a function of the graph-theoretic parameters of the flow graph.

Proof: Let $k \geq 1$ be given, and consider the following code:



As in constant propagation, the semilattice L in the type-finder framework consists of maps from program variables to types, but the meet of two such maps ϕ_1, ϕ_2 is defined as

$$(\phi_1 \wedge \phi_2)(V) = \phi_1(V) \ \underline{\text{dis}} \ \phi_2(V)$$

for all variables V.

Let f denote the effect of propagation through the loop, and let $\alpha = \{[A,\text{int}], [x_1,\text{int}], \ldots, [x_k,\text{int}]\} \in L$ be the initial information at the start of the loop. Then we have

$$f(\alpha) = \{[A,\{\text{int}\}], [x_1,\text{int}], \ldots, [x_n,\text{int}]\}$$
$$f^2(\alpha) = \{[A,\{\{\text{int}\}\}], [x_1,\{\text{int}\}], [x_2, \text{int}], \ldots, [x_k,\text{int}]\}$$
$$\vdots$$
$$f^N(\alpha) = \{[A,\overbrace{\{\ldots\{}^{N}\text{int}\}\ldots\}], [x_1,\overbrace{\{\ldots\{}^{N-1}\text{int}\}\ldots\}], \ldots, [x_N,\text{int}[\ \ldots\ [x_k,\text{int}]\},$$

where N = maximal nesting level of types, and we assume k > N.

$$f^{N+1}(\alpha) = \{[A,\overbrace{\{\ldots\{}^{N}\text{gen}\}\ldots\}], [x_1,\overbrace{\{\ldots\{}^{N-1}\text{gen}\}\ldots\}], [x_2,\overbrace{\{\ldots\{}^{N-1}\text{int}\}\ldots\}] \ldots, [x_N,\{\text{int}\}], [x_{N+1},\text{int}], \ldots, [x_k,\text{int}]\}$$
$$\vdots$$
$$f^{N+k}(\alpha) = \{[A,\overbrace{\{\ldots\{}^{N}\text{gen}\}\ldots\}], [x_1,\overbrace{\{\ldots\{}^{N-1}\text{gen}\}\ldots\}], \ldots, [x_k,\{\ldots\{\text{gen}\underbrace{\}\ldots\}}_{N-1}]\},$$

so that the framework is not (N+k)s-bounded, for any $k \geq N$. The same example shows that it takes N+k iterations through the loop before all types stabilize.

Q.E.D.

Corollary 4. Proposition 3 is also true for value-flow analysis.

Proof: Use the same example as in Proposition 3, and note that it takes k iterations through the loop to find that A at line 1 of the loop is in CRTHIS(X_k at line 2 of loop). (It also takes k iterations of the loop for this link to materialze at run-time!)

Q.E.D.

What are the consequences of the above results? One such consequence is that there does not exist a graph-theoretic bound on the number of iterations required by the type finder, or by the value-flow analysis. Note that even though the actual implementations of these analyses use an "optimized" approach of propagating data only along bfrom links and across instructions, this does not reduce the number of iterations required in the above example. Thus there exist codes which will cause the type-finder to iterate arbitrarily many times before types are stabilized. This is, however, not of major concern to us, since such cases will be extremely rare, and in any case, convergence is ensured.

A second, more important consequence relates to inter-procedural analysis, and implies that unless formulated carefully our currently implemented algorithms could either diverge, or provide us with wrong information. To see this, consider the case of the code given in Proposition 3, slightly modified as follows:

```
/* all variables are global */
          A := 1;
          x1:= x2:= ... := xk := 1;
          iterate( );
          end;
          proc iterate;
          if cond then return;
          else
```

```
            A := {A};
            x_k: = x_{k-1};
                 :
            x_2 := x_1;
            x_1 := arb A;
            iterate( );
            return;
    end if;
    end proc;
```

In the above code, we will have to generate the RC-paths L, LL, LLL, ..., $\underbrace{\text{LL}\cdots\text{L}}_{N+k}$ (where L denotes the recursive call to iterate), and assign to each of them different type information (i.e. $f(\alpha)$, $f^2(\alpha),\ldots,f^{N+k}(\alpha)$ respectively).

Thus we cannot simply ignore RC-paths containing more than a given number of repetitions, since this would lose information (in the sense of obtaining under-estimated types rather than over-estimated). However, if we keep track of all generated RC-paths, we may run into problems of divergence.

Two approaches to solve this problem are possible:

I. Modify the algorithm so that it will keep track of all generated RC-paths, but will be able to realize when information has stabilized, in the sense that even though more RC paths can be generated, tracing information along them will not change the already available information. This approach is likely to complicate the algorithm.

II. Compromise, and define the space of RC-paths in such a way that information might be traced also along non-interprocedurally valid execution paths (hopefully much fewer than all static paths), but will definitely be traced along all interprocedurally valid paths. NL 201 describes such an approach, but even it has to be modified, since we now must avoid involvement with infinitely many

RC-paths, before compacting them into only finitely many possible values.

Finally, we conjecture that the above problems that our inter-procedural analysis algorithms face do not stem from the special structure of these algorithms, but are of a more intrinsic nature, and that it is undecidable to find a solution to the (forward) type-finding analysis in which information is traced along all static inter-procedurally valid execution paths, and only along those paths.