Introducing Fixed-Point Iteration Early in a Compiler Course

Max Hailperin*
Gustavus Adolphus College
St. Peter, MN 56082 USA
max@gac.edu http://www.gac.edu/~max

Abstract

When teaching a course in compiler design, it is conventional to introduce the iterative calculation of least fixed points quite late in the course, in the guise of iterative data-flow analysis. In this paper I point out that the same mathematical and algorithmic ideas can be introduced much earlier, in the parsing portion of the course, as an explanation of the standard algorithm for computing the FIRST sets of a context-free grammar. Doing so not only renders these techniques more familiar when they re-appear in data-flow analysis, it also provides a more sound foundation for the FIRST algorithm than is typically offered. Moreover, these techniques deserve increased curricular prominence because they naturally lead to proofs of correctness for general non-deterministic algorithms that subsume multiple deterministic algorithms.

Introduction

By the time computer science students take a course in compiler design, they should be familiar with the notion of proving that the particular computational "path" taken by a deterministic algorithm has the correct result as its inevitable destination. However, they may well never before have seen a proof of correctness for an algorithm embodying "don't care" non-determinism. Such a proof shows that no matter which of an assortment of alternative paths the algorithm chooses to take, it winds up at the correct answer regardless—all roads lead to Rome. Yet despite the unfamiliarity of such proofs, they arise quite naturally in the iterative calculation of least (or greatest) fixed-points of functions on partially ordered sets, and hence arise naturally in the data-flow analysis portion of a compiler design course. Moreover this non-determinism is of great practical importance, because it allows one proof of correctness for a general non-deterministic algorithm to serve for many different deterministic algorithms that concretely instantiate it. As such, there is a desirable decoupling between correctness and efficiency: the correctness can be proven first, and then a deterministic instantiation chosen based on its efficiency.

The problem is that this approach is too fundamental to be addressed for the first time in the context of iterative data-flow analysis, late in the typical compiler design course. It would be preferable if the concepts could be introduced earlier in some other context, and then re-used for data-flow analysis. Much to my surprise, there is a natural opportunity early in the compiler design course, in the material on parsing. The standard compiler design texts, e.g. [1, 2], show a technique for computing the so-called FIRST sets of a grammar that amounts to an iterative computation of a least fixed-point. Yet none of the texts I am familiar with takes the opportunity to show how the correctness of such an algorithm can be verified. Therefore, this paper will remedy this omission by showing how the computation of FIRST sets can be used as an early introduction to non-deterministic least fixed-point iterations.

In the remaining sections, I will state the problem of computing FIRST sets for a context free grammar, re-express it as the quest for a least fixed point, then make the notion of a non-deterministic least fixed-point iteration precise and show how the FIRST problem can be solved through such an iteration, and finally sketch how this specific iteration can be proved correct. Although the focus is on this one problem, I'll point out the features that are relied upon for the proof, and hence govern the scope of its generalizability. This mirrors how I have taught this material and is an approach I advocate.

The FIRST Problem

Recall that a context-free grammar (CFG) possesses two disjoint finite sets of grammar symbols, known as terminal symbols and non-terminals; we will refer to these sets as T and N below. The CFG also possesses a finite set, P, of productions, each of which has a non-terminal as its left-hand side and a (possibly empty) finite string of grammar symbols as its right-hand side. A (possibly empty) finite string of grammar symbols is said to be derivable from an initial grammar symbol if one can start with the initial symbol and by successive replacements of the left-hand side of a production by the right-hand side arrive at the string.

We can now define the FIRST problem in this context. We will consider FIRST to be a function from $N \cup T$ to $T \cup \{\epsilon\}$, where ϵ is a special symbol not in $N \cup T$. (We will ignore the conventional, but easy, extension of FIRST's domain to longer strings of grammar symbols.) The fundamental definition of FIRST is

$$\begin{aligned}\text{FIRST}(x) \quad = \quad & \{y \in T \mid \text{a string starting with } y \text{ is} \\ & \qquad \text{derivable from } x\} \\ & \cup\, (\text{if the empty string is derivable} \\ & \qquad \text{from } x, \{\epsilon\}, \text{else } \{\})\end{aligned}$$

This function plays an important role in both LL and LR parser construction. Thus, any standard compiler construction text, such as [1] or [2], will contain some algorithm for computing the FIRST function of a CFG, or, as it is commonly termed, computing the FIRST sets.

However, it is not immediately obvious from the above fundamental definition of FIRST what form such an algorithm could take, because it is a completely non-effective definition: it suggests computing FIRST(x) by looking through the set of all strings derivable from x and seeing what terminals they start with. Yet in general infinitely many strings may be derivable from a non-terminal, rendering this approach impossible. This is why we redefine FIRST in the next section as the least fixed-point of a set of equations, which leads to an effective algorithm for its computation.

FIRST as a Least Fixed-Point

If there is a production with left-hand side A and all the symbols on the right hand side have ϵ in their FIRST sets, then ϵ must be in FIRST(A) as well, since A can derive the empty string by way of this production. Similarly, if the production is of the form $A \rightarrow X_1X_2 \cdots X_n$ and there is some k with $1 \leq k \leq n$ such that $y \in$ FIRST(X_k) and for all i in the range $1 \leq i < k$, $\epsilon \in$ FIRST(X_i), then y must also be in FIRST(A), since A can derive a string starting with y by way of this production.

These two observations both constrain FIRST(A) where A is a non-terminal. There is a similar pair of constraints for each production with left-hand side A. We can put together all these pairs of constraints in one equation for FIRST(A), namely FIRST$(A) = E(A) \cup L(A)$, where

$$E(A) = \begin{cases} \{\epsilon\} & \text{if there exists } A \rightarrow X_1X_2 \cdots X_n \in P \\ & \quad \text{such that } \bigwedge\limits_{1 \leq k \leq n} \epsilon \in \text{FIRST}(X_k) \\ \{\} & \text{otherwise} \end{cases}$$

and

$$\begin{aligned} L(A) = \quad & \{y \in T \mid (A \rightarrow X_1X_2 \cdots X_n) \in P \\ & \quad \wedge \bigvee_{1 \leq k \leq n} (y \in \text{FIRST}(X_k) \\ & \qquad\qquad \wedge \bigwedge_{1 \leq i < k} \epsilon \in \text{FIRST}(X_i))\} \end{aligned}$$

For a terminal symbol a, we have an even simpler constraint, namely that FIRST$(a) = \{a\}$. Taking all of these together, we wind up with a system of $|N| + |T|$ simultaneous equations governing the function FIRST.

In my experience, students are not intrinsically suspicious of the idea that such a system of equations can be a "definition" of FIRST. However, it generally does not require many leading questions regarding their past experiences with systems of equations for them to realize that in general such a system might have no solution or multiple solutions. It is then easy to show an example CFG where the system of equations does in fact have multiple solutions. For example, if the CFG consists of the productions $A \rightarrow Aa$ and $A \rightarrow b$, then does FIRST(A) equal $\{a, b\}$ or just $\{b\}$? Both are solutions to the system of equations. (We are using the convention that lower-case letters are terminals and upper-case letters are non-terminals.)

This illustration, that there can be more than one solution, leads to the notion of defining FIRST as being the *least* fixed-point (i.e., solution), where "least" is in the sense of pointwise set inclusion, i.e., F is "less than or equal to" G if for all $x \in N \cup T$, $F(x) \subseteq G(x)$. In other words, we define FIRST as being a fixed point such that each of the individual FIRST sets is a subset of the corresponding one from any other fixed-point, since intuitively that makes FIRST be "junk-free." (This intuition can be verified, though I haven't done so when teaching this.) We still need to show that the system of equations will always have a fixed point, that there exists among the fixed points a least one, and that we have an effective algorithm for finding it. (Note that uniqueness is not at issue: if there is a least fixed point, then it must be unique, because two distinct sets can't both be subsets of each other.) We will tackle these goals together: our proof that the algorithm finds the least fixed point will also serve as proof that one exists.

The Iterative Algorithm

In general, a non-deterministic least fixed-point iteration starts with the least element of a partially ordered set (poset), which we'll require to have such an element, and repeatedly chooses and applies any function from a finite set of functions, subject only to the constraint that the function "makes a difference" i.e, that $x_{i+1} = f_i(x_i) \neq x_i$. This continues until none of the functions makes a difference, at which point the algorithm terminates with the final x_n as its value. Subject to appropriate conditions on the poset and the functions, we can show that the algorithm terminates and that x_n is the least common fixed point of the functions. (The nomenclature and exact definitions vary; some authors call these iterations "chaotic" rather than "non-deterministic" and loosen the restriction that only functions that make a difference may be chosen to allow a finite number of useless function applications. See for example [3].)

Specifically in the case of the FIRST problem, we will find our least fixed point by starting at F_0 defined by $F_0(x) = \{\}$, for all $x \in N \cup T$. The set of functions available for choosing each f_i from will have $|N| + |T|$ members, one for each grammar symbol. We will name them after the grammar symbols, e.g. f_A for the non-terminal A and f_a for the terminal a. We'll use A as a representative non-terminal and a as a representative terminal: everything should be interpreted as applying to all the other symbols as well.

The functions for the terminals are particular simple; we have

$$f_a(F)(x) = \begin{cases} \{a\} & \text{if } x = a \\ F(x) & \text{otherwise} \end{cases}$$

In other words, $f_a(F)$ is identical to F except possibly for at a, which it maps to $\{a\}$ regardless of whether F did so.

The functions for the terminals are more complex, though again we have the property that $f_A(F)$ is identical to F except possibly at A. This time, however, $f_A(F)(A)$ is not independent of F, unlike $f_a(F)(a)$:

$$f_A(F)(x) = \begin{cases} E_A(F) \cup L_A(F) & \text{if } x = A \\ F(x) & \text{otherwise} \end{cases}$$

where

$$E_A(F) = \begin{cases} \{\epsilon\} & \text{if there exists } A \to X_1 X_2 \cdots X_n \in P \\ & \quad \text{such that } \bigwedge_{1 \le k \le n} \epsilon \in F(X_k) \\ \{\} & \text{otherwise} \end{cases}$$

and

$$\begin{aligned} L_A(F) = \quad & \{y \in T \mid (A \to X_1 X_2 \cdots X_n) \in P \\ & \wedge \bigvee_{1 \le k \le n} (y \in F(X_k) \\ & \qquad \wedge \bigwedge_{1 \le i < k} \epsilon \in F(X_i))\} \end{aligned}$$

Notice that finding a fixed point of these functions is the same as solving the earlier equations for FIRST. This is because these functions come directly from the equations, but rather than expressing FIRST in terms of itself, they express one approximation to FIRST, $f_A(F)$, in terms of the previous approximation, F.

Verifying the Algorithm

Clearly if the algorithm terminates it does so at a fixed point, since that is the stopping condition for the iteration. So the remaining questions are whether the fixed point found on termination is the least one, and whether we can guarantee termination.

We'll use $\sqsubseteq$ for our partial order; recall that for the FIRST problem we are defining $F \sqsubseteq G$ to mean that $F(x) \subseteq G(x)$ for all $x \in N \cup T$.

Suppose G is some arbitrary fixed point of the functions f, i.e., for all $x \in N \cup T$, $f_x(G) = G$. Since F_0 is the least element of the poset, $F_0 \sqsubseteq G$. Next we rely on a property of the functions, namely that they are all monotonic. This means that if $F \sqsubseteq G$, then $f(F) \sqsubseteq f(G)$. It isn't hard to verify that the f_a and f_A given above are monotonic. Since all the functions are monotonic, whichever is chosen as f_0 is. So, since $F_0 \sqsubseteq G$, by monotonicity $F_1 = f_0(F_0) \sqsubseteq f_0(G)$. But as G is a fixed point, $f_0(G) = G$, so $F_1 \sqsubseteq G$. By the same argument $F_2 \sqsubseteq G$, $F_3 \sqsubseteq G$, etc. By induction, we can show that $F_n \sqsubseteq G$. Thus, if the algorithm terminates with a fixed point F_n, we can be sure that it is the least one, since $F_n \sqsubseteq G$, where G is a completely arbitrary fixed point.

To show termination we'll show that the F iterates form a chain, i.e., that $F_0 \sqsubseteq f_0(F_0) = F_1 \sqsubseteq f_1(F_1) = F_2 \sqsubseteq \cdots \sqsubseteq f_{n-1}(F_{n-1}) = F_n$. Since our poset doesn't have any infinite strictly increasing chains, eventually a fixed point must be reached. In terms of the FIRST sets, we are saying they can't grow forever, which is obvious given that each FIRST set is a subset of the finite set $T \cup \{\epsilon\}$.

Showing that the F iterates form a chain is the trickiest part. We want to show that for any arbitrary i, $F_i \sqsubseteq f_i(F_i) = F_{i+1}$. We will do this by inductively assuming that F_0 up through F_i form a chain, and use that assumption to show that the chain continues on to F_{i+1}. Suppose that $f_i = f_\alpha$ for some grammar symbol α. Looking at the definitions of f_a and f_A, we see that for all $x \in N \cup T$ other than α, $F_i(x) = F_{i+1}(x)$. So, the only issue is whether $F_i(\alpha) \subseteq F_{i+1}(\alpha)$. If f_i is the first use of f_α, i.e., there is no k such that $0 \le k < i$ and f_k also is f_α, then clearly $F_i(\alpha) = \{\}$, since $F_0(\alpha) = \{\}$ and only f_α changes this FIRST set. Thus in this case we have $F_i(\alpha) = \{\} \subseteq F_{i+1}(\alpha)$, and need only to consider the remaining case, that f_α had been previously used. Let k be the most recent step where it was used, i.e., we have $0 \le k < i$ and $f_k = f_i = f_\alpha$, but for no j in the range $k < j < i$ does $f_j = f_\alpha$. Because the other functions leave α's FIRST set unchanged, $F_i(\alpha) = F_{k+1}(\alpha)$. From our inductive hypothesis that we have a chain up to F_i, we have $F_k \sqsubseteq F_i$, and then by monotonicity of f_α, $f_\alpha(F_k) \sqsubseteq f_\alpha(F_i)$. In other words, $F_{k+1} \sqsubseteq F_{i+1}$. In particular, then, $F_{k+1}(\alpha) \subseteq F_{i+1}(\alpha)$. But recall that $F_{k+1}(\alpha) = F_i(\alpha)$. Thus we've shown, as desired, that $F_i(\alpha) \subseteq F_{i+1}(\alpha)$ and so $F_i \sqsubseteq F_{i+1}$. Thus the chain, which we inductively assumed reached at least as far as F_i, is shown to continue to F_{i+1}. By induction, then, all the Fs form a chain, completing our proof that the non-deterministic algorithm must terminate with the least fixed point as its result.

Any deterministic algorithm, then, which performs one of the various specific computations that the general non-deterministic iteration is capable of, must also compute the least fixed point of our collection of functions, i.e., FIRST.

Generalization

In teaching this material, it is important to point out what features of the poset and of the set of functions we relied upon, in order that the scope for generalization is apparent.

Our proof relied on the poset having a least element and no infinitely ascending chains. In fact, our poset was finite, so it was clear that it had no infinitely ascending chains. Even some infinite posets, however, have this property, and sometimes these "infinitely wide but finitely high" posets arise in data-flow analysis. We also relied on two properties of the functions f_a and f_A. These functions are monotonic, and they don't interfere, because each changes only one of the FIRST sets, and each changes a different FIRST set. Again, these properties recur in many data-flow analysis contexts, where the functions are monotonic and non-interfering.

Finally, it is important when teaching this material to point out that these properties are essential to make this particular form of proof work, but not necessarily to make a non-deterministic least fixed-point iteration work. Other properties can be used as the basis for other kinds of proofs.

Conclusions

The material presented in this paper, which I am advocating others incorporate into their compiler design courses, is admittedly more sophisticated than much of what undergraduates encounter. Yet if there is anywhere in the curriculum where students can be expected to handle such material, it is in the compiler design course. My own experience has been that in a highly interactive small class, where the students can actively participate in the construction of the proof, their minds can expand to accommodate this proof without snapping. Those compelled by class size to use a more conventional lecture format might have more difficulty successfully incorporating this material, however.

Assuming this difficulty can be overcome, my experience is that the atmosphere of magic that otherwise surrounds the computation of FIRST is dispelled and iterative data-flow analysis becomes less mysterious as well, since it becomes the application of a (somewhat) familiar technique to a new domain, rather than introducing both technique *and* domain. A few of the students even see non-deterministic least fixed-point iteration as aesthetically pleasing or "cool."

References

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] FISCHER, C. N., AND LEBLANC, JR., R. J. *Crafting a Compiler*. Benjamin/Cummings, 1988.

[3] GESER, A., KNOOP, J., LÜTTGEN, G., STEFFEN, B., AND RÜTHING, O. Chaotic fixed point iterations. Tech. Rep. MIP-9403, Universität Passau, Fakultät für Mathematik und Informatik, Oct. 1994.