# From Iterative Implementations to Single-Pass Functions

AZADEH FARZAN, University of Toronto
VICTOR NICOLET, University of Toronto

Algorithms can be written in many ways, which can be a problem when one want to design automated methods to reason about them. The designer of these automated methods can either explicitly restrict their input to be of a given form, or they need to design methods to normalize inputs into a suitable format. We propose a methodology that does the latter for a wide class of imperative and recursive programs. A reference iterative or recursive algorithm reducing an input collection is translated to a functionally equivalent single-pass algorithm.

## 1 MOTIVATING EXAMPLE

We use an example to illustrate an example to illustrate our problem. Consider a set of (input) points on a 2D plane. A point is *Pareto optimal* if all the other points are either below or to the left of this point. Let $p.x$ and $p.y$ denote the coordinates of point $p$. Formally:

$$p \succ p' \iff p.x \geq p'.x \lor p.y \geq p'.y$$
$$\mathrm{POP}(X) = \{p \in X \mid \forall p' \in X, p \succ p'\}.$$

```
List<Point> l = []
for(i = 0; i < n; i++) {
  bool b = true;
  for(j = 0; j < i; j++)
    b = b && X[i] ≻ X[j];
  if(b) l.append(X[i]);
}
```

Fig. 1. Pareto Optimal Points

The code in Figure 1 is an implementation of POP that is not a single pass implementation. The input of the implementation is the list of points `X`. At each step of the outer loop, the point `X[i]` is compared with every other point in the list by the inner loop. `X[i]` is optimal iff the boolean accumulator `b` used in the inner loop is true. In this case, `X[i]` is added to the list of points to be returned `l`. While this implementation is easy for a programmer to write and understand, it might not be well suited for representing the algorithm as a function of the input collection directly. The elements of the input collection are read multiple times, and it is not clear how the result of the function changes when the input collection changes.

The code in Figure 2 is another implementation of POP with the same input collection as in the previous implementation. At each iteration, the set of optimal points `l` is updated by maintaining the points that are still optimal with respect to the new input, and adding the new input if it is optimal with respect to all the current optimal points. The main difference with the code in Figure 1 is that this implementation is single-pass: each element `p` of the list `X` is read only once. This implementation is also easier to analyze, if the goal is to model the POP function as a function of its input collection.

```
List<Point> l = [];
List<Point> tmp = [];
for(p in X) {
  bool b = true;
  tmp = [];
  for(e in l) {
    if(e ≻ p)tmp.append(e);
    b = b && (p ≻ e); }
  if (b) tmp.append(p);
  l=tmp;
}
```

Fig. 2. Single pass POP

The choice of an implementation for an algorithm can be of great importance for further application of automated methods. For example, if one wanted to parallelize the POP example, one could aim to either parallelize the implementation in Figure 1 by breaking the iteration space of the outer loop, or derive a divide-and-conquer implementation where the inputs list needs to be divided [5], effectively breaking the outer loop of the implementation in Figure 2.

Authors' addresses: Azadeh Farzan, University of Toronto, azadeh@cs.toronto.edu; Victor Nicolet, University of Toronto, victorn@cs.toronto.edu.

## 2 BACKGROUND

Let $Sc$ be a type that stands for any scalar type used in typical programming languages, such as `int` and `bool`, whenever the specific type is not important in the context. Scalars are assumed to be of *constant* size, and conversely, any constant-size representable data type is assumed to be scalar. Consequently, all operations on scalars are assumed to have constant time complexity. Type $\mathcal{S}$ defines the set of all *sequences* of elements of type $Sc$. The concatenation operator $\bullet : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ defined over sequences is associative. The sequence type stands in for *arrays*, *lists*, or any collection data type that admits a linear iterator and an *associative* composition operator.

**Functions of Sequences.** A function $h : \mathcal{S} \rightarrow D$ is rightwards iff there exists a binary operator $\oplus : D \times Sc \rightarrow D$ such that for all $x \in \mathcal{S}$ and $a \in Sc$, we have $h(x \bullet [a]) = h(x) \oplus a$. A rightwards function can be defined by a left fold over a sequence: $h(x) = \textbf{foldl} \ \oplus \ h([]) \ x$. A leftward function is defined analogously using the recursive equation $h([a] \bullet x) = a \otimes h(x)$. A function is **single-pass** if it is leftwards or rightwards.

**Loops.** We forgo the formal definition of an imperative input language since an informal exposition provides enough clarity. Our input sequential programs can be written in a simple imperative language with basic constructs for branching and looping (in the style of the code snippets given as motivating examples); the syntax used in the examples is standard, and the technique is not language specific. A loop is defined by an iterator $i \in \mathcal{I}$ and its loop body B written in the imperative language. The set of variables that appear in the body is partitioned into *state variables*, the variable that are written in the loop body, and *input variables*. The tuple of state variables is of type $D$.

Without loss of generality, let us assume the loop has one input variable $x$ that is of type $\mathcal{S}$. In the body of the loop, this input sequence is accessed through the iterator $i \in \mathcal{I}$. We define $\zeta_{\mathcal{I}} : \mathcal{S} \rightarrow \mathcal{S}$ as the mapping of input sequences to the sequences of inputs accessed by the body with iterator $i \in \mathcal{I}$. Note that each distinct element of $x$ may appear multiple times in $\zeta_{\mathcal{I}}(x)$ if the loop visits it multiple times. Below, we discuss how a single-pass recursive function can be produced from the loop, depending on the relationship between $x$ and $\zeta_{\mathcal{I}}(x)$.

## 3 SYNTHESIZING SINGLE-PASS FUNCTIONS

The input to our synthesis routine can be in the form of an imperative loop $for(i \in \mathcal{I})\{\mathsf{B}\}$ or a recursive function $f$. We describe how to produce a single-pass recursive function from a given reference implementation, whether it is a `for` loop or a recursive function.

For any given imperative loop, there exist many recursive functions that perform and equivalent computation (i.e. produce the same desired output). The iteration space of a loop implicitly defines an input that is traversed and processed by the loop. The functional representation of the loop can maintain this iteration space as its input, independent of what the underlying data looks like.

The input data for the implementation of POP in Figure 1 is a list of points on a plane, while the iteration space defined by the loop is a quadratic traversal of this input list. Alternatively, the algorithm can be implemented as a single pass function directly over the original input data. The code in Figure 2 is an iterative implementation of such an algorithm.

The same dichotomy applies if the reference implementation is given a recursive function (instead of a loop). It may be a single-pass function on a given input, or it may traverse its input through an alternative traversal strategy where the same input cell is visited many times. In this section, we focus on loops as input, since they are the more complicated case. We discuss how various functional representations for loops may be produced. The same ideas and transformations are applicable to recursive functions.

## 3.1 Iteration Space as Input Sequence

A `for` loop with iterator $i \in \mathcal{I}$ can be translated to a single-pass function $f_I$ that reads the sequence of inputs as defined by iteration space $\mathcal{I}$. In other words, $\zeta_{\mathcal{I}}(x)$ is considered to be the input to the recursive function. One such translation is outlined in [3, 4]. To keep this manuscript self-contained, we give a quick overview of this translation here.

First, the body of the loop is translated to an operator that takes two inputs: the state of the loop and the mapping of the input sequence on the current iterator. This translation step is simply a translation to SSA followed by rewriting to a functional form; the similarity between both models is explained in [2]. Nested loops are treated recursively using the method described here. Then, the function itself is built as a single pass function using the operator constructed in the first step, with the iteration space as input sequence.

The function for the POP example of Section 1 is defined for any mapped input sequence $\zeta(A) = [(A[i], A)\ for\ i = 1..len(A)]$ (each element is the pair of $A[i]$ and $A$ itself), by:

$$\begin{aligned} f_{\mathcal{I}}(\zeta(A)) = &\quad \textbf{foldl}\ \circledvee\ [\,]\ \zeta(A) \\ s \circledvee (a, z) = &\quad \textbf{let}\ b = \textbf{foldl}\ \lambda(b', a').(b' \wedge a \gtrdot a')\ \mathsf{true}\ z\ \textbf{in} \\ &\quad\ \mathsf{if}\ b\ \mathsf{then}\ s \bullet [a]\ \mathsf{else}\ s \end{aligned}$$

## 3.2 Original Data as Input Sequence

In this case, the function $f$ is performing a single pass over the original input data sequence $x$ whose traversal is predetermined by the collection type, in contrast to a single pass over $\zeta_{\mathcal{I}}(x)$. Unlike $f_{\mathcal{I}}$ in Section 3.1, $f$ cannot be produced via a simple code-to-code transformation. Intuitively, $f$ implements a different algorithm compared to $f_{\mathcal{I}}$ and this new algorithm may need to be *synthesized.* We use syntax-guided synthesis (SyGuS) [1] to discover the new recursive definition of $f$. As standard in SyGuS, the problem is defined by the correctness specification for this synthesis task, and then the sketch and expression grammar that define the programs that can be synthesized.

Note that this synthesis step is necessary when the original loop or function reads the elements of the input data more than once, while $f$ is single-pass on the input data and reads every element exactly once. In such cases, a state variable, of the type of the input collection, should be added to signature of $f$ to let it remember the necessary parts of the input sequence. We denote the new variable by $s.l$. Therefore, $f$ has type $\mathcal{S} \longrightarrow D'$ where $D'$ is the domain extended with this new state. In the general case, $D' = D \times \mathcal{S}$. We can then formally specify $f$ through its relation to $f_{\mathcal{I}}$ as $\forall x \in \mathcal{S} :\ f(x){\downarrow} = f_{\mathcal{I}}(\zeta_{\mathcal{I}}(x))$ where $\downarrow$ is the projection from $D'$ to $D$. In some problem instances, for example POP, a variable of this type already exists in the original state, and therefore, no new variable needs to be added (i.e. $D' = D$).

$f$ is sketched as a single pass function with operator $\oplus$ defined with unknown functions $init$, $\boxtimes$ and $post$. For $x \in \mathcal{S}$, $a \in Sc$ and $s \in D'$, the (functional) sketch for $f$ is:

$$\begin{aligned} f(x) &= \textbf{foldl}\ \oplus\ f_I([\,])\ x \\ s \oplus a &= post(a, \textbf{foldl}\ \boxtimes\ \ init(s, a)\ s.l) \end{aligned}$$

We limit the solution space for $init$, $\boxtimes$, and $post$ such that $f$ is in the same asymptotic complexity class as the reference function (and $f_{\mathcal{I}}$). This is done through a standard technique of bounding the definition of the synthesizable program grammar.

***Example* 3.1.** The implementation of POP given in Figure 2 is a solution of this sketch. The functional form of this implementation is the function $f$ above, with the following synthesized implementations for *init*, ⊠ and *post*:

$$\begin{aligned} init(s, a) &= ([], \text{true}) \\ s \boxtimes e &= (e > a\ ?\ s.l \bullet [e] : s.l,\ s.b \wedge a > e) \\ post(a, s) &= (s.b\ ?\ s.l \bullet [a] : s.l, s.b) \end{aligned}$$

Intuitively, the *init* operation corresponds to the code before the inner loop in Figure 2, the ⊠ operator is the body of the inner loop and the *post* operation corresponds to the update of the list after the inner loop. ⌟

## 3.3 Other Traversal Strategies

Given raw input data, there may be several different traversal strategies for processing the data to compute the desired function. One is captured by the reference imperative loop, as outlined in Section 3.1. Another is a single-pass over the input data, as outlined in Section 3.2. There may be many more possibilities. Note that a specific iteration strategy is part of the algorithmic design of the reference imperative code. In our search for a divide-and-conquer solution, we do not set out to search for all possible alternative traversals of the input data that could potentially lead to other divide-and-conquer algorithms. Beyond the traversal strategy that is already present in the reference implementation, we only produce the one that is a single-pass over the input data, because a divide-and-conquer algorithm for this version will have to divide the input data.

Note that as a result of a small change in the iteration strategy, for example traversing a list backwards instead of forwards, the signature of function $f$ may have to change substantially [3, 7]. Therefore, given a fixed new iterator $j \in \mathcal{J}$ and a mapping $\zeta_{\mathcal{J}}$ of the input collection on this iterator, the problem of synthesizing a new function $g$ that computes the original values devised by $f$ is entirely non-trivial and generally unsolved. Now, consider performing a search along all possible candidates for $\mathcal{J}$ to discover a particular recursive definition for the loop in the form of a new function $g$. For each such given $g$, one needs to search for a divide-and-conquer implementation. The search space for the possible solutions will be huge.

## 4 EXPERIMENTS

The single-pass synthesis from iterative implementations has been implemented as part of the tool Parsynt [5]. The tool is implemented in OCaml [6] and uses Rosette [8] as a syntax-guided synthesis solver in the background.

### 4.1 Experimental Results

Table 1 presents the synthesis times for some of the benchmarks used in [5]. The table reports the synthesis time required to translate each benchmark written in an iterative loop to a single-pass function that takes the original data as an input sequence (Section 3.2). For the *closest pair* and the *intersecting intervals* benchmarks, the output is a scalar variable, so the tool adds a list that temporarily memorizes the inputs. The time complexity of the result of the functional translation is always the same, but the single-pass version might require more space. For the other benchmarks, time and spatial complexity for both single-pass and iterative implementations are the same.

| | Synt. (s) |
|---|---|
| Sorting | 1.6 |
| $k$-largest | 1.3 |
| Closest pair | 2.0 |
| Intersecting intervals | 0.7 |
| Histogram | 1.3 |
| POP | 4.3 |
| Min. points | 4.5 |
| Quadrant convex hull | 4.7 |

Table 1. Synthesis times for benchmarks from [5]. Benchmarks running on a laptop with 6-core Intel Core i7-8750H CPU @ 2.20GHz and 16GB RAM running Ubuntu 20.04.

## REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. 1–25.

[2] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Not.* 33, 4 (April 1998), 17–20.

[3] Azadeh Farzan and Victor Nicolet. 2017. Synthesis of Divide and Conquer Parallelism for Loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 540–555.

[4] Azadeh Farzan and Victor Nicolet. 2019. Modular Divide-and-conquer Parallelization of Nested Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, 610–624.

[5] Azadeh Farzan and Victor Nicolet. 2021. Phased Synthesis of Divide and Conquer Programs. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2021)*. ACM.

[6] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. The OCaml system release 4.07: Documentation and user's manual. (2018).

[7] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic Inversion Generates Divide-and-conquer Parallel Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 146–155.

[8] Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. 135–152.