ForML: Iterative Programming with Type Classes and Associated Types

Mark Cohen
Department of Computer Science
University of Chicago

5 June 2019

Abstract

Iterative programming is simply the best way to work with collections in terms of being powerful, highly intuitive, and imposing a low cognitive load. Thinking in terms of `map`, `filter`, and `fold` can certainly be just as intuitive, but only after substantially more training; especially for introductory-level programmers, iteration is more developer-friendly. We present a design for pure polymorphic iteration in the ML discipline. We base this design in a language modeled after ML, extended with type classes and associated types. This paper presents the motivation, design, intuition, and formalism behind this language. Advised by Adam Shaw.

1 Introduction

Collections are the basis of software development, but despite this fact, they are often rather painful to work with. Consider the following Python program:

```
for (k, v) in db:
    if query(k):
        res.append((k, v))
return dict(res)
```

This is highly intuitive: the gap between the programmer's mental model and the actual program is minimal. This particular snippet iterates over all key-value pairs in a dictionary, performs some query on each one, and appends all keys that pass the query to some response list. Then, it converts that response list to a dictionary with blank values.

Python requires minimal underlying infrastructure to enable overloading: all that is required of the programmer is that `db` have the `__iter__` method implemented.

Unfortunately, there are several pain points that Python can produce with this program. First, what if `db` is not something that can be iterated over? Since Python does not perform static type checking, that will cause a run-time error. Similarly, run-time errors will be caused if `res` is not something that can be appended onto or converted into a dictionary. Especially pernicious are run-time errors arising in the last case, when converting between collections.

Rust is a relatively young language that brings strong, static type checking and modern type theory to systems programming. Based on traits [5], Rust can produce a similar level of elegance to Python, but with the backing of a strong type system. Consider the following Rust translation of the above Python example:

```
for (k, v) in db.iter() {
    if query(k) {
        res.push((k, v));
    }
};
res.collect::<HashMap<K, V>>()
```

This program reads roughly the same as the equivalent program in Python. However, behind the scenes is an `Iterator` trait that provides the infrastructure necessary to verify the type of this program. Additionally, rather calling overloaded collection constructors, Rust opts for an explicitly annotated `collect` method, itself leveraging traits for the sake of type safety.

Contrast the above two examples with the following equivalent Standard ML (SML) program:

```
let
  fun arrToDict' d [] = d
    | arrToDict' d ((k, v) :: xs) =
        arrToDict' (dictAdd d (k, v
           )) xs
  fun arrToDict xs = arrToDict' {}
      xs
in
  arrToDict (filter (fn (k, _) =>
      query k) d)
end
```

In the land of ML, we still have the benefit of static type checking. However, there is no infrastructure for reusing code; the bulk of the example above is actually concerned with how to convert between lists and dictionaries. Additionally, as previously discussed, thinking in terms of `map`, `filter`, and `fold` often comes with a higher cognitive load than thinking in terms of iteration, especially for new programmers.

We seek to develop a discipline of iteration in ML by combining the positive aspects of the above-discussed disciplines, while avoiding the downsides.

2 Design

To enable iteration, we propose a structure built around a complementary pair of type classes and four keyword constructs. All of this lives at the surface level: the programmer is free to implement instances of these type classes, and to use the keyword constructs as specified. Before diving deep into the definitions, we will lay out the infrastructure and prior research on top of which these constructs are built.

2.1 Infrastructure

Present in this design are two well-developed concepts not present in ML: type classes and associated types.

Type classes, originally presented by Wadler & Blott [6], allow clean encapsulation of behaviors of individual collections. Chiefly, the presence of type classes in our language obviates the combinatorial explosion of "convert X to Y" functions that would otherwise be necessary in ML.

Traits were the logical first research direction, as Rust is the closest existing language to our proposed design. However, upon further investigation, the original presentation of traits by Schärli et al. [5] was unsuitable as a foundation for this research; Schärli et al. developed traits as subordinate to object-oriented classes, for the purpose of sharing signatures and methods between closely related but non-hierarchical class structures. This means that, for example, traits cannot imply one another like type classes can (think of the type signature for pair equality in Haskell, `Eq a, Eq b => Eq a * b`). Another limitation this imposes is that the programmer must own a type to implement a trait for it.

Associated types, as originally presented by Chakravarty et al. [2], allow for linking types together. For the purposes of this design, we only require associated types of kind `*`: the infrastructure presented by Chakravarty et al. is substantially more powerful, but only the simplest case of it is necessary here.

For the sake of consistency, we will embed the results of Chakravarty et al. [2] in the notation of Wadler & Blott [6]. An associated type, intuitively

```
class Iterable α where
    type item
    val next: α → (item * α) option

class Collectible α where
    type item
    val default: α
    val insert: α → item → α

for <var> in <collection> → <ty>
  <expr>

collect <expr>

pass

bail
```

Figure 1: Design for iteration constructs

speaking, will be treated as an "overloaded" type: in the notation of Wadler & Blott [6], we would write `item` $::_i$ α `list` $\backslash\, \alpha$ to signify that the associated type `item` can be instantiated for α `lists`, translating to α.

We would like to operate as much as possible at the surface level; accordingly, after discussing the above design, we will show its translation to the intermediate language of Wadler & Blott [6], at which point we will allow their system to take over the remaining translation down to ML.

With these definitions in mind, we can now dissect the design proposed in figure 1.

2.2 The `Iterable` type class

The `Iterable` type class, unsurprisingly, marks a type as iterable. The class contains an associated type called `item`, which corresponds to the element type of the collection. For an α `list`, `item` is α. For a `string`, `item` is `char`.

The `next` method takes a collection, and tries to produce a pair of the collection's next element with the rest of the collection. If the collection is empty, it returns `None`. This method encodes both loop variable bindings and loop bounds: at each iteration, the loop variable is bound to the first half of the pair returned by `next`. When `next` returns `None`, we conclude execution.

2.3 The `Collectible` type class

The `Collectible` type class is something a little different: it marks a type as able to be built up piece-by-piece. Like `Iterable`, the class contains an associated type `item`, which functions exactly the same as in `Iterable`.

`default` is something unusual: a non-function class member. Its sole purpose is to hold a value that represents an "empty" collection: for an α `list`, `default` is `[]`. For a `string`, `default` is `""`. The purpose of `default` is to allow the programmer to build up a result over individual iterations without having to carry an accumulator around in a larger scope.

The `insert` method takes a collection and an item and returns a new collection containing both the elements of the collection argument and the item argument.

2.4 The `for` keyword construct

The `for` keyword takes an identifier `<var>`, which represents the loop variable. `<var>` will be bound to each element of `<collection>` in succession before each evaluation of `<expr>`. After performing inference on the body of `<expr>`, `<var>` must have a type that unifies with `<collection>`'s `item` type. The notation $\rightarrow$ `<ty>` signifies that the programmer wishes to *build* a collection of type `<ty>`: all loops in this discipline must explicitly build a collection. This may seem restrictive, however given that the programmer can implement the `Collectible` type class for types that are not traditionally considered "collections", it does not impose undue restriction.

There are important type restrictions on the various forms in `for`: `<collection>` must be an `Iterable`, and `<ty>` must be a `Collectible`. Lastly, intuitively, `<expr>` must unify with `<ty>`'s `item` type (modulo a special "glue" type that we will soon discuss): that is, `<expr>` must be something that can be `insert`ed into an internal accumulator of type `<ty>`.

2.5 The `collect`, `pass`, and `bail` keyword constructs

The `collect` keyword takes only an expression `<expr>`, which implicitly is assumed to contain references to the loop variable `<var>`; though of course there are situations where it may not. `collect`, intuitively, signals to add the following `<expr>` to the collection being built.

The `pass` keyword's sole purpose is to make `filter` expressions well-founded. Consider the following program to filter out all zero elements in a list, which would be infeasible without `pass`:

```
for x in [1, 2, 0, 3] → int list
  if zero? x
  then pass
  else collect x
```

The `bail` keyword, like `pass`, is nothing more than a placeholder. Its responsibility is to make expressions that "bail" early (i.e. terminate iteration when a certain condition is reached) well-founded. Consider the following program to gobble tokens until a closing parenthesis is reached, which would not be feasible to write without `bail`:

```
val toks = [Num 4, Plus, Num 3,
   RParen, Times, Num 2]
for tok in toks -> token list
  if tok = RParen
  then bail
  else collect tok
```

`collect`, `pass`, and `bail` will be typed using a special "glue" type, to avoid certain nonsense situations - we will discuss the "glue" type later. For now, understand `pass` and `bail` to impose no type restrictions, and to be defined to unify with any `collect` expression.

3 Translation

Figure 2 shows the general translation of a loop construct to Wadler & Blott's [6] intermediate language.

Here in the type signatures of `collect`, `pass`, and `bail` we see the "glue" type alluded to earlier, α `collect`. Under the hood, an α `collect` is exactly isomorphic to an α `option`; the reason for its existence is to keep these syntactic forms distinct from, for example, the forms for `option`s.

The type predicates in `for` encode the requirement that γ be an `Iterable` and that ι be a `Collectible`: note the use of `next` and `insert` from each of those type classes. Under those requirements, the body of `for` is a simple recursive construct, relying on the other three keywords to control its execution.

`collect`, `pass`, and `bail` have rather uninteresting definitions: as discussed, all that is necessary to ensure here is that they remain distinct from

```
let
  for :: $\forall\gamma.\forall\iota.$
    (item $::_i$ $\gamma$ \ item$_\gamma$).
    (item $::_i$ $\iota$ \ item$_\iota$).
    (next $::_i$ $\gamma \rightarrow$ (item$_\gamma$ * $\gamma$) option \ next$_\gamma$).
    (insert $::_i$ $\iota \rightarrow$ item$_\iota$ $\rightarrow \iota$).
    $\gamma \rightarrow \iota \rightarrow$ (item$_\gamma$ $\rightarrow$ item$_\iota$) $\rightarrow \iota$
    = fn collection iota expr =>
     case (next collection)
       of None => iota
        | Some (item, rest) =>
            case (expr item)
              of Bail => iota
               | Pass => for rest iota expr
               | Collect item' => for rest (insert iota item') expr

  collect :: $\forall\iota.$ (item $::_i$ \ item$_\iota$). item$_\iota$ $\rightarrow$ item$_\iota$ collect
    = fn item => Collect item

  pass :: $\forall\iota.$ $\iota$ collect = Pass

  bail :: $\forall\iota.$ $\iota$ collect = Bail

  acc :: $\forall\iota.$ (default $::_i$ $\iota$ \ default$_\iota$). $\iota$ = default
in
  for <collection> acc (fn <var> => <expr>)
end
```

Figure 2: Translation of loop design to intermediate language

other syntactic forms (so that the execution of `for` is uniquely controlled by these three constructs).

Next, we define a value `acc` that simply serves to hold the value of `default`, from the instance of `Collectible` on `<ty>`.

The last important piece of this translation is that we wrap `<expr>` in a function, binding the loop variable. As discussed previously, we assume `<expr>` to contain references to the loop variable; here is where that open binding is closed.

Given these five definitions, we can translate any loop written in the surface language to an expression in the intermediate language. From the intermediate language, we leverage the existing infrastructure of Wadler & Blott [6] to translate the rest of the way down, at which point we are left with an ordinary ML expression.

4 Sample programs

Given the design-based nature of this research, we would be remiss to not discuss some of the more elegant programs this design can produce. The simplest form of iteration to encode is a `map`, which our design can do readily:

```
for x in [1, 2, 3, 4] → string list
  collect (toString x)
```

In fact, `map` can even be implemented in this discipline:

```
val map = fn f xs =>
  for x in xs → $\alpha$ list
```

```
    collect (f x)
```

`filter` expressions can also be expressed, as we saw in the motivation of the `pass` and `bail` keywords. Below is the implementation of `filter` itself:

```
val filter = fn pred xs =>
  for x in xs → α list
    if pred x
    then collect x
    else pass
```

`fold` itself cannot be implemented in this discipline, due to the use of a type to provide the initial accumulator value (through the `default` type class member) rather than requiring that the programmer pass the value themselves. However, we can readily express `fold` expressions:

```
instance Collectible string where
  type item = string
  val default = ""
  val insert = concat

val strs = ["hi ", "world", "!"]
for x in strs → string
  collect x
```

In this discipline, `fold` expressions will usually have a very simple body: the interesting parts come in the type annotations for the result collection. This imposes a much lower cognitive load than ordinary `fold` expressions, which are notoriously hard to grasp (especially for new programmers).

Now with a command of the power of this discipline, let us build a small lexer for a prefix calculator language. The grammar is as follows:

$$\nu = 1, 2, 3, \ldots \qquad \text{(numbers)}$$

$$\chi = + \mid - \mid * \mid \div \qquad \text{(operators)}$$

$$e = (\chi\ \nu\ \nu) \mid \nu \qquad \text{(expressions)}$$

Assume an intuitive ML datatype representing expressions in this language. Now, we can implement our scanner:

```
val scan = fn s =>
  for c in s → token list
    case c
      of #"(" => collect LParen
       | #")" => collect RParen
       | #"+" => collect Add
       | #"-" => collect Sub
       | #"*" => collect Mul
       | #"÷" => collect Div
       | (numeric n) =>
           collect (Num n)
       | #"␣" => pass
       | _ => bail
```

5 Limitations

The use of type classes imposes one major limitation on this design: it is not possible to have multiple instances of a class on the same type in scope simultaneously. For example, a programmer desiring to iterate over a search tree in both preorder and in-order fashion must work around our design; each kind of traversal would require its own instance of `Iterable`. The same limitation is present with `Collectible`: for example, one cannot have a definition of `collect` on lists that represents both `cons` and `append` (i.e. operating on an α and an α `list` respectively) in scope simultaneously. Likely, this problem can be solved by simply limiting the scope of instances to allow overriding.

Another limitation is that in this current design, nested loops are not entirely well-founded. Consider the following program to take the Cartesian product of two lists:

```
for x in xs → (α * β) list
  for y in ys → ?
    collect (x, y)
```

What is the collection type of the inner loop? The trouble is that in nested loops, the programmer usually does not want to build up an inner collection, but rather use *all* levels of nesting in tandem to build up one outer collection. Nested iteration will need a careful treatment in future work.

References

[1] Luca Cardelli, *Basic polymorphic typechecking*, Science of Computer Programming **8** (1988).

[2] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones, *Associated type synonyms*, ICFP (2005).

[3] Luis Damas and Robin Milner, *Principal type-schemes for functional programs*, PoPL (1982).

[4] Edsger W. Dijkstra, *Making a translator for algol-60* (1961).

[5] Natanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black, *Traits: The formal model* (2003).

[6] Philip Wadler and Stephen Blott, *How to make ad-hoc polymorphism less ad hoc*, PoPL (1989).

A `INST-PRED`: Making translation algorithmic

While investigating type classes as a research foundation, we discovered that one of the inference rules in Wadler & Blott's paper [6] - `PRED` - is non-algorithmic: it matches any term with a predicated type, rather than a specific syntactic form. As a consequence, the rule has to be attempted at every point in the inference process, lest the translated output be incorrect. Take the formalized definition of pair equality in figure 3: intuitively, it should translate to the representation in figure 5. However, following Wadler & Blott's rules strictly [6], the `SPEC`, `ABS`, and `COMB` rules are triggered first, and `PRED` is not invoked until deep inside the expression, at which point the dictionary-passing bindings are put in the wrong place (figure 4).

In order to arrive at the correct translation, one either must make precarious decisions about the order of rule application, or create an algorithmic rule to augment the system. We propose the latter.

First, to formulate this rule, we propose a predicate-stripping relation π, defined below:

$$\begin{aligned} \pi(\varnothing) &= \varnothing \\ \pi(\rho_1.\rho_2 \ldots \rho_n) &= (\rho_n,\ \rho_1.\rho_2 \ldots \rho_{n-1}) \end{aligned}$$

Intuitively, this rule simply returns the last predicate in a list with the remaining predicates in a list.

Next, and again to ease the formulation of our new rule, we propose a slight modification to the type grammar presented by Wadler & Blott [6]:

$$\begin{aligned} \tau &= \tau \to \tau' \mid \alpha \mid \chi(\tau_1 \ldots \tau_n) && \text{(bare types)} \\ \rho &= \varnothing \mid (x :: \tau).\rho && \text{(type predicates)} \\ \sigma &= \varnothing \mid \forall\alpha.\sigma && \text{(type schemes)} \\ & \sigma\rho\tau && \text{(types)} \end{aligned}$$

All that is important to note here is the lack of fall-through cases for ρ and σ, and the necessity of including all three σ, ρ, and τ when expressing a type.

Now, to arrive at the desired translation, we propose a `PRED-INST` rule that specifically matches `inst` declarations with predicated types, defined in figure 6.

B LUCA-lang

The bulk of the working hours spent on this research revolved around building an experimental compiler for the language we propose. As the academic quarter drew to a close, efforts on this front had to be stopped in favor of finalizing the research, but the language served as an indispensable tool for shaping understanding, especially of topics in type inference and compiler design.

The language is named **LUCA**, after Luca

```
over eq :: ∀α. α → α → bool in
inst eq :: int → int → bool = intEq in
inst eq :: char → char → bool = charEq in
inst eq :: ∀α.∀β.
              (eq :: α → α → bool).
              (eq :: β → β → bool).
              (α * β → α * β → bool)
              = λx.λy.
                (eq (fst x) (fst y)) andalso
                  (eq (snd x) (snd y))
              in
eq (1, #"a") (1, #"a")
```

Figure 3: Pair equality, formalized (from Wadler & Blott [6])

```
let eq_int = intEq in
let eq_char = charEq in
let eq_α*β = λx.λy.
            (λeq_α.eq_α (fst x) (fst y)) andalso
              (λeq_β.eq_β (snd x) (snd y))
            in

eq_α*β eq_int eq_char (1, #"a") (1, #"a")
```

Figure 4: Pair equality, translated using PRED (from Wadler & Blott [6])

Cardelli, whose paper [1] explaining the seminal work in polymorphic type inference by Damas & Milner [3] was instrumental in the genesis of this research. Though the compiler is not currently in a functioning state, the work that went into it was of paramount importance. Below is an overview of the most edifying experiences of this process.

B.1 Polymorphic type inference

The process of type unification (that is, the process by which $\forall\alpha.\alpha \to \alpha$ is deemed to be the same as `int` $\to$ `int`) constitutes a substantial leap of understanding beyond the standard undergraduate curriculum in programming language theory. The implementation of Algorithm W [3] eventually grew into a full-fledged set of compiler phases.

Annotation walks an abstract syntax tree and produces an annotated AST; the process itself is "dumb" in that it does not perform any inference, but rather serves as an aid for the next phase, collection. Collection walks an annotated AST and produces a constraint set, containing pairs of types that must unify for the program to have a valid type. Unification is the final phase, which folds over a constraint set, generating a substitution which, when applied, will generate the *most general* type for the program [3]. Each of these three phases is implemented in their own SML structure, and can be found in `annotate.sml`, `collect.sml`, and `unify.sml`, respectively.

B.2 Monadic programming

At a certain point, the repetitious nature of recursive-descent parsing became too much to bear. Monadic programming offered an elegant solution. Below is a snippet of positively beautiful

```
let eq_int = intEq in
let eq_char = charEq in
let eq_α*β = λeq_α.λeq_β.λx.λy.
                (eq_α (fst x) (fst y)) andalso
                  (eq_β (snd x) (snd y))
                in

eq_α*β eq_int eq_char (1, #"a") (1, #"a")
```

Figure 5: Pair equality, translated using PRED-INST

$$\frac{\begin{array}{c}(x ::_o \_\_\_) \in A \\ (x' ::_o \_\_\_) \in A \\ \pi(\rho) = ((x' :: \tau'), \bar{\rho}) \\ A, (x' :: \tau' \setminus x_{\tau'}) \vdash e :: \sigma\bar{\rho}\,\tau \setminus \bar{e} \\ A, (x' :: \tau' \setminus x_{\tau'}) \vdash e' :: \sigma''\rho''\tau'' \setminus \bar{e}'\end{array}}{A \vdash (\texttt{inst}\ x :: \sigma\rho\tau = e\ \texttt{in}\ e') :: \sigma''\rho''\tau'' \setminus (\texttt{inst}\ x :: \sigma\bar{\rho}\tau = \lambda x'_{\tau'}.\bar{e}\ \texttt{in}\ \bar{e}')}$$

Figure 6: PRED-INST: an algorithmic update of PRED

code from the **LUCA** parser:

```
and absExpr ts = (
  litKey "fn"          >>
  (many1 identifier) >>= (fn xs =>
  lit Scan.FatArrow    >>
  expr                 >>= (fn e =>
  pure (foldr AST.Abs e xs)))) ts
```

The monadic sequencing operators `>>=`, `>>`, and `pure` allow computations to be built "by description". This parser reads as a very minimal translation of the term grammar for abstractions (`fn` $x \ldots \Rightarrow e$). Listing out the steps in plain English, one might write:

- Gobble the "fn" keyword. If that works,
- Parse at least one identifier (the function arguments). If that works,
- Gobble an arrow. If that works,
- Parse the function body. If that works,
- Assemble a curried stack of function ASTs, using the list of identifiers and function body parsed above.

The beauty of monadic programming is that each step of plain English translates directly - almost word for word - to a line of monadic code. In addition to this novel method of describing algorithms, reimplementing Haskell's `Control.Monad` library provided a glimpse into some of the more powerful constructs available to ML programmers, chiefly `Functor`. Monadic programming is used throughout the entire **LUCA** compiler, with several different monadic types. The most interesting uses of monadic programming can be found in `parse.sml` and `collect.sml`.

B.3 Shunting yard algorithm

When the time came to parse type annotations, the primary limitation of monadic parsing became a major roadblock: the inability to cleanly parse infix and postfix grammars. The **LUCA** compiler has a separate parser for type expressions, which uses Dijkstra's shunting yard algorithm [4], so named for its visual resemblance to a railroad shunting yard. The algorithm encodes operator precedence - not an easy task in monadic parsing - by maintaining a separate output queue and operator stack. Dijkstra's original presentation [4] described a conversion from mixed-fixity notation to RPN. However, since we required conversion to an AST, we modified the algorithm, changing the output queue to an output stack. "Pushing" to the output queue, then, instead means applying the current operator to the top n elements of the output stack (where n is the operator's arity), then placing the result of that application on the top of the output stack. By the time execution terminates, the output stack should be a singleton of one AST, representing the parsed expression. The **LUCA** compiler's type expression parser can be found in `tyexpr.sml`.