Iterative Programming in ML with Type Classes and Associated Types

Bachelor's Thesis

Mark Cohen

Department of Computer Science
University of Chicago

5 June 2019

Motivations

Iterative programming is intuitive, expressive, and easy to work with.

Motivations

- Python:

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

return dict(res)
```

Motivations

- Python:

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

return dict(res)
```

- Rust:

```
for (k, v) in db.iter() {
    if query(k) {
        res.push(v);
    }
};

res.collect::<HashMap<K, V>>()
```

Motivations

- SML:

```
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
```

Motivations

How can we have our cake and eat it too?

First directions: traits

Can we use/modify Rust's trait implementation?

First directions: traits

Can we use/modify Rust's trait implementation?
No:

- The "type ownership" problem is baked in to the existing theory.

Schärli et al., "Traits: The Formal Model", 2003

First directions: traits

Can we use/modify Rust's trait implementation?
No:

- The "type ownership" problem is baked in to the existing theory.
- Traits are subordinate to *object-oriented* classes.

Schärli et al., "Traits: The Formal Model", 2003

First directions: type classes

Can we use type classes?

First directions: type classes

Can we use type classes?

- Type classes stand on their own as a complete infrastructure of reuse.

Wadler & Blott, "How to make *ad-hoc* polymorphism less *ad hoc*", 1983.

First directions: type classes

Can we use type classes?

- Type classes stand on their own as a complete infrastructure of reuse.
- The programmer can freely create instances of type classes for types not owned by them.

Wadler & Blott, "How to make *ad-hoc* polymorphism less *ad hoc*", 1983.

But...

First directions: type classes

Can we use type classes?

- Type classes stand on their own as a complete infrastructure of reuse.
- The programmer can freely create instances of type classes for types not owned by them.

Wadler & Blott, "How to make *ad-hoc* polymorphism less *ad hoc*", 1983.

But...

- We need *associated types*.

First directions: associated types

Associated types are the key to type-sound iteration.

First directions: associated types

Associated types are the key to type-sound iteration.

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

First directions: associated types

Associated types are the key to type-sound iteration.

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

```
struct Counter {
    count: u64
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

First directions: associated types

Associated types are the key to type-sound iteration.

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

impl Iterator for Counter {
    type Item = u64;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;
        Some(self.count)
    }
}
```

First directions: associated types

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

impl Iterator for List<T> {
  type Item = T;

  fn next(&mut self) -> Option<Self::Item> {
    match self.head {
      Some(node) => {
        let tmp = node.value;
        self.head = node.next;
        Some(tmp)
      },
      None => None
    }
  }
}
```

Final direction

Use type classes and associated types to develop a discipline of iteration with none of the drawbacks discussed thus far.

LUCA-lang

We will soon discuss the properties of a small theoretical language. Some of this language is currently built; some of it doesn't quite work yet.

- Language is named LUCA, after Luca Cardelli

Luca Cardelli, "Basic Polymorphic Typechecking", 1987.

Desired design

Goal: iterate over any collection, building any collection.

Desired design

Goal: iterate over any collection, building any collection.

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

instance Iterable α list where
  type item = α
  val next = fn xs =>
    if nil? xs
    then None
    else Some (hd xs, tl xs)
```

Desired design

Goal: iterate over any collection, building any collection.

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

instance Iterable α list where
  type item = α
  val next = fn xs =>
    if nil? xs
    then None
    else Some (hd xs, tl xs)

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

instance Collectible α list where
  type item = α
  val default = []
  val insert = listAppend
```

Desired design

Goal: iterate over any collection, building any collection.

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

instance Iterable α list where
  type item = α
  val next = fn xs =>
    if nil? xs
    then None
    else Some (hd xs, tl xs)

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

instance Collectible α list where
  type item = α
  val default = []
  val insert = listAppend

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

Desired design

How about `fold`?

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

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

Desired design

How about `filter`?

```
for id in users → string list
  if registered? id
  then collect id
  else pass
```

Implementing type classes: translation

Goal: transform a program with `class` and `instance` expressions to one without

Wadler & Blott, "How to make *ad-hoc* polymorphism less *ad hoc*", 1983.

Implementing type classes: translation

Goal: transform a program with `class` and `instance` expressions to one without

- `class` expressions are thrown away

Wadler & Blott, "How to make *ad-hoc* polymorphism less *ad hoc*", 1983.

Implementing type classes: translation

Goal: transform a program with `class` and `instance` expressions to one without

- `class` expressions are thrown away
- `instance` expressions are transformed into `let` expressions

Wadler & Blott, "How to make *ad-hoc* polymorphism less *ad hoc*", 1983.

Implementing type classes: translation

Goal: transform a program with `class` and `instance` expressions to one without

- `class` expressions are thrown away
- `instance` expressions are transformed into `let` expressions
- Calls to overloaded functions are substituted away

Wadler & Blott, "How to make *ad-hoc* polymorphism less *ad hoc*", 1983.

Translation: example

```
class Eq α where
  eq :: α -> α -> bool

instance Eq Int where
  eq = intEq

instance Eq Char where
  eq = charEq

(eq 1 1) andalso (eq #"a" #"a")
```

Translation: example

```
over eq :: ∀α. α → α → bool in
inst eq :: int → int → bool = intEq in
inst eq :: char → char → bool = charEq in
(eq 1 1) andalso (eq #"a" #"a")
```

Damas & Milner, "Principal type-schemes for functional programs", 1982.

Translation: example

$\texttt{let eq}_{\texttt{int}}\texttt{ = intEq in}$
$\texttt{let eq}_{\texttt{char}}\texttt{ = charEq in}$

$\texttt{(eq}_{\texttt{int}}\texttt{ 1 1) andalso (eq}_{\texttt{char}}\texttt{ \#"a" \#"a")}$

Translation: predicated types

```
class Eq α where
  eq :: α -> α -> bool

instance Eq Int where
  eq = intEq

instance Eq Char where
  eq = charEq

instance Eq α, Eq β => Eq α * β where
  eq = fn x y =>
    (eq (fst x) (fst y)) andalso (eq (snd x) (snd y))

eq (1, #"a") (1, #"a")
```

Translation: predicated types

```
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")
```

Translation: making the **PRED** rule algorithmic

Per Wadler & Blott, the translation should be:

let $\texttt{eq}_{\text{int}}$ = intEq in
let $\texttt{eq}_{\text{char}}$ = charEq in
let $\texttt{eq}_{\alpha*\beta}$ = $\lambda\texttt{eq}_{\alpha}.\lambda\texttt{eq}_{\beta}.\lambda\texttt{x}.\lambda\texttt{y}.$
    ($\texttt{eq}_{\alpha}$ (fst x) (fst y)) andalso
      ($\texttt{eq}_{\beta}$ (snd x) (snd y))
    in

$\texttt{eq}_{\alpha*\beta}$ $\texttt{eq}_{\text{int}}$ $\texttt{eq}_{\text{char}}$ (1, #"a") (1, #"a")

Translation: making the **PRED** rule algorithmic

Per Wadler & Blott, the translation should be:

```
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")
```

But the corresponding rule is non-algorithmic:

$$\textbf{PRED}\quad \frac{(x ::_o \sigma) \in A \qquad A, (x :: \tau \setminus x_\tau) \vdash e :: \rho \setminus \bar{e}}{A \vdash e :: (x :: \tau).\rho \setminus (\lambda x_\tau.\bar{e})}$$

Wadler & Blott, "How to make *ad-hoc* polymorphism less *ad hoc*", 1983.

Translation: making the **PRED** rule algorithmic

To make this rule algorithmic, we need to:

Translation: making the **PRED** rule algorithmic

To make this rule algorithmic, we need to:

- Define a relation that will allow us to strip predicates in a predictable manner;

Translation: making the **PRED** rule algorithmic

To make this rule algorithmic, we need to:

- Define a relation that will allow us to strip predicates in a predictable manner;
- Define a new **PRED** rule using the above to match a specific syntactic form.

Predicate stripping

Recall the predicated type of pair equality:

$$\begin{aligned}\forall\alpha.\forall\beta.&(\texttt{eq} :: \alpha \to \alpha \to \texttt{bool}).\\ &(\texttt{eq} :: \beta \to \beta \to \texttt{bool}).\\ &(\alpha * \beta \to \alpha * \beta \to \texttt{bool})\end{aligned}$$

Predicate stripping

Recall the predicated type of pair equality:

$$\begin{aligned}\forall\alpha.\forall\beta.&(\mathtt{eq} :: \alpha \to \alpha \to \mathtt{bool}).\\ &(\mathtt{eq} :: \beta \to \beta \to \mathtt{bool}).\\ &(\alpha * \beta \to \alpha * \beta \to \mathtt{bool})\end{aligned}$$

Define a stripping relation π as follows:

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

Predicate stripping

Recall the predicated type of pair equality:

$$\begin{aligned}\forall\alpha.\forall\beta.&(\mathsf{eq} :: \alpha \to \alpha \to \mathtt{bool}).\\ &(\mathsf{eq} :: \beta \to \beta \to \mathtt{bool}).\\ &(\alpha * \beta \to \alpha * \beta \to \mathtt{bool})\end{aligned}$$

Define a stripping relation π as follows:

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

So,

$$\pi(\forall\alpha.\forall\beta \ldots) = ((\mathsf{eq} :: \beta \to \beta \to \mathtt{bool}), (\mathsf{eq} :: \alpha \to \alpha \to \mathtt{bool}))$$

Translation: making the **PRED** rule algorithmic

We propose the following rule:

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

Translation: making the **PRED** rule algorithmic

$$
\begin{aligned}
\texttt{inst eq} :: \forall\alpha.\forall\beta.&(\texttt{eq} :: \alpha \to \alpha \to \texttt{bool}).\\
&(\texttt{eq} :: \beta \to \beta \to \texttt{bool}).\\
&(\alpha * \beta \to \alpha * \beta \to \texttt{bool})\\
&= \lambda x.\lambda y \ldots \texttt{ in e'}
\end{aligned}
$$

Translation: making the **PRED** rule algorithmic

$$
\begin{aligned}
\texttt{inst eq} :: \forall\alpha.\forall\beta.&(\mathsf{eq} :: \alpha \rightarrow \alpha \rightarrow \texttt{bool}).\\
&(\mathsf{eq} :: \beta \rightarrow \beta \rightarrow \texttt{bool}).\\
&(\alpha * \beta \rightarrow \alpha * \beta \rightarrow \texttt{bool})\\
&= \lambda x.\lambda y \ldots \ \texttt{in e'}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{inst eq} :: \forall\alpha.\forall\beta.&(\mathsf{eq} :: \alpha \rightarrow \alpha \rightarrow \texttt{bool}).\\
&(\alpha * \beta \rightarrow \alpha * \beta \rightarrow \texttt{bool})\\
&= \lambda \boldsymbol{eq}_\beta.\lambda x.\lambda y \ldots \ \texttt{in e'}
\end{aligned}
$$

Translation: making the **PRED** rule algorithmic

$$\begin{aligned}\texttt{inst eq} :: \forall\alpha.\forall\beta.&(\mathsf{eq} :: \alpha \to \alpha \to \texttt{bool}).\\ &(\mathsf{eq} :: \beta \to \beta \to \texttt{bool}).\\ &(\alpha * \beta \to \alpha * \beta \to \texttt{bool})\\ &= \lambda x.\lambda y \ldots \ \texttt{in e'}\end{aligned}$$

$$\begin{aligned}\texttt{inst eq} :: \forall\alpha.\forall\beta.&(\mathsf{eq} :: \alpha \to \alpha \to \texttt{bool}).\\ &(\alpha * \beta \to \alpha * \beta \to \texttt{bool})\\ &= \lambda \boldsymbol{eq}_\beta.\lambda x.\lambda y \ldots \ \texttt{in e'}\end{aligned}$$

$$\begin{aligned}\texttt{inst eq} :: \forall\alpha.\forall\beta.&(\alpha * \beta \to \alpha * \beta \to \texttt{bool})\\ &= \lambda \boldsymbol{eq}_\alpha.\lambda \boldsymbol{eq}_\beta.\lambda x.\lambda y \ldots \ \texttt{in e'}\end{aligned}$$

Associated types: formalisms

Chakravarty et al. define *higher-kinded* associated type synonyms.

Chakravarty et al., “Associated Type Synonyms”, 2005.

Associated types: formalisms

Chakravarty et al. define *higher-kinded* associated type synonyms.

- We don't need the full power of this infrastructure.

Chakravarty et al., "Associated Type Synonyms", 2005.

Associated types: formalisms

Chakravarty et al. define *higher-kinded* associated type synonyms.

- We don't need the full power of this infrastructure.
- Instead, we treat associated types as a special case of overloading.

Chakravarty et al., "Associated Type Synonyms", 2005.

Associated types: formalisms

We need to add to our language:

Associated types: formalisms

We need to add to our language:

- A form that allows us to overload a type

Associated types: formalisms

We need to add to our language:

- A form that allows us to overload a type
- A form that allows us to declare an instance of a type

Associated types: formalisms

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

Associated types: formalisms

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

- over next :: $\forall \texttt{item}.\forall\alpha.\ \alpha \to \big(\texttt{item} * \alpha\big)$ option in ...

Associated types: formalisms

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

- over next :: $\forall \texttt{item}. \forall \alpha.\ \alpha \rightarrow (\texttt{item} * \alpha)$ `option in ...`

```
instance Iterable string where
  type item = char
  val next = fn xs =>
    if xs = ""
    then None
    else Some (hd xs, tl xs)
```

Associated types: formalisms

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

- over next :: $\forall$item.$\forall\alpha.\ \alpha \to \big($item $*\ \alpha\big)$ option in ...

```
instance Iterable string where
  type item = char
  val next = fn xs =>
    if xs = ""
    then None
    else Some (hd xs, tl xs)
```

- inst next :: string $\to \big($char $*$ string$\big)$ option $= \lambda$s... in ...

Associated types: formalisms

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

Associated types: formalisms

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

- `over default` $:: \forall$`item.`$\forall\alpha.\ \alpha$ `in ...`
- `over insert` $:: \forall$`item.`$\forall\alpha.\ \alpha \rightarrow$ `item` $\rightarrow \alpha =$ `... in ...`

Associated types: formalisms

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

- `over default` $::\ \forall$`item`$.\forall\alpha.\ \alpha$ `in ...`
- `over insert` $::\ \forall$`item`$.\forall\alpha.\ \alpha \to$ `item` $\to \alpha =$ `... in ...`

```
instance Collectible string where
  type item = char
  val default = ""
  val next = fn xs =>
    if xs = ""
    then None
    else Some (hd xs, tl xs)
```

Associated types: formalisms

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

- over default :: $\forall$item.$\forall\alpha.\ \alpha$ in ...
- over insert :: $\forall$item.$\forall\alpha.\ \alpha \rightarrow$ item $\rightarrow \alpha =$ ... in ...

```
instance Collectible string where
  type item = char
  val default = ""
  val next = fn xs =>
    if xs = ""
    then None
    else Some (hd xs, tl xs)
```

- inst default :: string $=$ "" in ...
- inst insert :: string $\rightarrow$ char $\rightarrow$ string $= \lambda$s.λc ... in ...

Iteration: formalisms

Recall our desired designs:

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

for x in ["hi", " ", "world", "!"] → string
  collect x

for id in users → string list
  if registered? id
  then collect id
  else pass
```

Iteration: formalisms

Recall our desired designs:

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

for x in ["hi", " ", "world", "!"] → string
  collect x

for id in users → string list
  if registered? id
  then collect id
  else pass
```

- We need to formalize `for`, `collect`, and `pass`.

Iteration: `for` formalism

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

Iteration: for formalism

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

Typing constraints:

- `<collection>` must be `Iterable`

Iteration: for formalism

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

Typing constraints:

- `<collection>` must be `Iterable`
- `<ty>` must be a `Collectible`

Iteration: `for` formalism

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

Typing constraints:

- `<collection>` must be `Iterable`
- `<ty>` must be a `Collectible`

Evaluation:

- Bind internal variable ι to `default` (from `<ty> Collectible`)

Iteration: `for` formalism

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

Typing constraints:

- `<collection>` must be `Iterable`
- `<ty>` must be a `Collectible`

Evaluation:

- Bind internal variable ι to `default` (from `<ty> Collectible`)
- Try: let `(item, rest) = next(collection)` (from `Iterable`)
    - If None, return ι

Iteration: for formalism

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

Typing constraints:

- `<collection>` must be `Iterable`
- `<ty>` must be a `Collectible`

Evaluation:

- Bind internal variable ι to `default` (from `<ty> Collectible`)
- Try: let `(item, rest) = next(collection)` (from `Iterable`)
  - If None, return ι
- Bind `<var>` to `item`, evaluate `<expr>`
  - If `<expr>` returns a `collect`, insert that returned value into ι

Iteration: `for` formalism

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

Typing constraints:

- `<collection>` must be `Iterable`
- `<ty>` must be a `Collectible`

Evaluation:

- Bind internal variable ι to `default` (from `<ty>` `Collectible`)
- Try: let `(item, rest) = next(collection)` (from `Iterable`)
  - If None, return ι
- Bind `<var>` to `item`, evaluate `<expr>`
  - If `<expr>` returns a `collect`, insert that returned value into ι
- Bind `<collection>` to `rest`

Iteration: `for` formalism

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

Typing constraints:

- `<collection>` must be `Iterable`
- `<ty>` must be a `Collectible`

Evaluation:

- Bind internal variable ι to `default` (from `<ty>` `Collectible`)
- Try: let `(item, rest) = next(collection)` (from `Iterable`)
    - If None, return ι
- Bind `<var>` to `item`, evaluate `<expr>`
    - If `<expr>` returns a `collect`, insert that returned value into ι
- Bind `<collection>` to `rest`
- Recurse

Iteration: `collect` formalism

```
collect <expr>
```

Iteration: `collect` formalism

```
collect <expr>
```

Typing constraints:

- `<expr>` must be the same type as `item` (from `<ty> Collectible`)

Iteration: `collect` formalism

```
collect <expr>
```

Typing constraints:

- `<expr>` must be the same type as `item` (from `<ty> Collectible`)

Evaluation:

- Return `<expr>`

Iteration: pass formalism

```
pass
```

Iteration: pass formalism

```
pass
```

- Just do nothing

Iteration: pass formalism

```
pass
```

- Just do nothing
- Do not pass go

Iteration: pass formalism

```
pass
```

- Just do nothing
- Do not pass go
- Do not collect $200

Iteration: big payoff!

```
let
  val pass = None
  val collect = fn x => Some x
  val for = fn collection ι expr =>
    case (next collection)
      of None => ι
       | Some (item, rest) =>
           case (expr item)
             of Some item' => for rest (insert ι item') expr
              | None => for rest ι expr
in
  for <collection> default (fn <var> => <expr>)
end
```

Iteration: big payoff!

```
for id in users → string list
  if registered? id
  then collect id
  else pass
```

Iteration: big payoff!

```
for id in users → string list
  if registered? id
  then collect id
  else pass
```

```
let
  val pass = None
  val collect = fn x => Some x
  val for = fn collection ι expr =>
    case (next collection)
      of None => ι
       | Some (item, rest) =>
           case (expr item)
             of Some item’ => for rest (insert ι item’) expr
              | None => for rest ι expr
in
  for users [] (fn id => if registered? id then collect id else
      pass)
end
```

Iteration: big payoff!

$$
\begin{aligned}
\texttt{for} :: \forall\gamma.\forall\iota.\\
&\big(\texttt{item} ::_i \gamma \setminus \texttt{item}_\gamma\big).\\
&\big(\texttt{item} ::_i \iota \setminus \texttt{item}_\iota\big).\\
&\big(\texttt{next} ::_i \gamma \rightarrow \big(\texttt{item}_\gamma * \gamma\big)\ \texttt{option} \setminus \texttt{next}_\gamma\big).\\
&\big(\texttt{insert} ::_i \iota \rightarrow \texttt{item}_\iota \rightarrow \iota\big).\\
&\gamma \rightarrow \iota \rightarrow \big(\texttt{item}_\gamma \rightarrow \texttt{item}_\iota\big) \rightarrow \iota
\end{aligned}
$$

Thank you!