Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 325 (2016) 127–146

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Iteration and Labelled Iteration

Bram Geron[1] and Paul Blain Levy [2]

School of Computer Science, University of Birmingham, Edgbaston, Birmingham, B15 2TT, UK

Abstract

We analyse the conventional sum-based representation of iteration from the perspective of programmers, and show that the syntax they suggest is fundamentally not a good representation of Java-style iteration with `for`, `while`, `break`, and `continue`. We present an alternative syntax, which we call "labelled iteration", where loops are identified using labels.
The languages are analysed: we give denotational and operational semantics, adequacy proofs for both languages, and a translation function from sum-based iteration to labelled iteration.

Keywords: iteration, loops, lexical binding, operational semantics, denotational semantics, higher-order language, lambda calculus, de Bruijn indices

1 Introduction

1.1 Overview

Iteration is an important programming language feature.

- In imperative languages, it is best known in `for` and `while` loops. The meaning of such a loop is to iterate code until some condition is met, or if the condition is never met, the loop diverges. Such loops are often supplemented by `break` and `continue`.
- It has also been studied in the lambda calculus setting [13,19,21].
- In the categorical setting, iteration corresponds to complete Elgot monads [9]. They descend from iterative, iteration, and Elgot theories, and their algebras and monads [7,1,2,3,23], which study variants of the sum-based iteration $-^\dagger$. This field is related to Kleene monads [10,17,18].

[1] Email: bxg314@cs.bham.ac.uk
[2] Email: P.B.Levy@cs.bham.ac.uk

http://dx.doi.org/10.1016/j.entcs.2016.09.035

Iteration can be implemented using recursion, but it is simpler: semantics of recursion require a least fixpoint, where iteration has a simple set-based semantics. Also from the programmer's perspective, iteration and recursion are different: a program using a `for` or `while` loop can sometimes be clearer than the same program using recursion.

1.2 The sum-based representation of iteration

We study two representations of iteration. First, the classical sum-based construct $-^\dagger$ that turns a computation $\Gamma, A \vdash M : A + B$ into a computation $\Gamma, A \vdash M^\dagger : B$. Categorically, this representation of iteration corresponds to complete Elgot monads [9]. To understand the correspondence better, we introduce a term constructor iter for $-^\dagger$. (Details are in Section 2.)

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Gamma, x{:}A \vdash^{\mathsf{c}} M : A + B}{\Gamma \vdash^{\mathsf{c}} \mathsf{iter}\ V,\ x.\ M : B}$$

Imperative programs with `for` and `while` can now be encoded using iter. As an example, the program on the left corresponds to the term on the right:

imperative	λ-calculus-like
$x := V;$ $\mathsf{while}\ (p(x))\ \{$ $\quad x := f(x);$ $\}$ $\mathsf{return}\ g(x);$	$\mathsf{iter}\ V, x.$ $\quad \mathsf{if}\ \ p(x)$ $\quad \mathsf{then\ return\ inl}\ f(x)$ $\quad \mathsf{else\ \ return\ inr}\ g(x)$

This works as follows. The iter construct introduces a new identifier x, which starts at V. The body is evaluated. If the body evaluates to $\mathsf{inr}\ W$, then the loop is finished and its result is W. If the body evaluates to $\mathsf{inl}\ V'$, then we set x to V', and keep on evaluating the body until it evaluates to some $\mathsf{inr}\ W$.

1.3 The "De Bruijn index" awkwardness with the sum-based representation

Programmers using imperative languages regularly use nested loops, as well their associated `break` and `continue` statements, which may be labelled. Such statements are not essential for programming, and code using `break` or `continue` can be rewritten so it does not use either statement, but this usually comes at a price in readability. There is usually a labelled and an unlabelled form of `break` and `continue`.

On the left side of Figure 1, we show an program in a Java-like language with nested labelled loops, and labelled `continue` statements. The colours can be ignored for now. The program computes the formula $\sum_{\substack{0\le i\le 8 \\ \wedge\, a[i][0]\neq 5}} \prod_{\substack{0\le j\le 8 \\ \wedge\, a[i][j]\ \text{even}}} a[i][j]$,

```
        int sum = 0;
outer:  for (int i = 0; i ≤ 8; i++){

            if (a[i][0] == 5)
                continue outer;
            int prod = 1;
inner:      for (int j = 0; j ≤ 8; j++){

                if (¬isEven(a[i][j]))
                    continue inner;
                if (a[i][j] == 0)
                    // product will be 0.
                    continue outer;
                prod = prod * a[i][j];
            }
            sum = sum + prod;
        }
        int result = sum;
```

```
iter ⟨0, 0⟩, ℓ1.  let ⟨sum, i⟩ = ℓ1 in
    if i ≥ 9 then
        return inr sum
    else if a[i][0] == 5 then
        return inl ⟨sum, i + 1⟩
    else iter ⟨1, 0⟩, ℓ2.  let ⟨prod, j⟩ = ℓ2 in
        if j ≥ 9 then
            return inr inl ⟨sum + prod, i + 1⟩
        else if ¬isEven a[i][j] then
            return inl ⟨prod, j + 1⟩
        else if a[i][j] == 0 then

            return inr inl ⟨sum, i + 1⟩
        else
            return inl ⟨prod * a[i][j], j + 1⟩
```

Figure 1. Two programs programs that compute the same formula. The left hand program is written in Java; the right hand program uses fine-grain call-by-value with *sum-based* iteration. Related fragments have the same colour. Both programs compute $\sum_{\substack{0\le i\le 8\\ \wedge\, a[i][0]\neq 5}} \prod_{\substack{0\le j\le 8\\ \wedge\, a[i][j] \text{ even}}} a[i][j]$.

although the specific formula is not important for the example. Recall from Java that "continue inner" aborts the current iteration of the inner loop, and continues with a fresh iteration of the inner while loop. Statement "continue outer" does the analogous thing but for the outer loop. It will abort the inner loop implicitly.

On the right side, we have a similar program to compute the same formula, but using sum-based iteration.

We use colour to indicate fragments that are intuitively related, because control flows to the same place after those fragments:

- After continue inner and the two occurrences of return inl $\langle \cdots, j + 1\rangle$, control flows to the beginning of the inner loop. We have drawn a solid purple box around those fragments. The assignment to *prod* on the left also precedes the beginning of the inner loop, and we have coloured it purple.

- After both occurrences of continue outer, control flows to the beginning of the outer for loop. Similarly, after return inl $\langle sum, i+1 \rangle$ and after the two occurrences of return inr inl $\langle \cdots \rangle$, control flows to the beginning of the outer iter. We have drawn a thick dashed red box around those fragments. The assignment to *sum* on the left also precedes the beginning of the outer loop, and we have coloured it red.

Note that in the left (Java) program, both statements in red boxes are written the same: "continue outer".

Both programs work, but the syntax of the right hand program has two awkwardnesses for programmers:

(i) Continuing to the outer loop (red) is written return inl $\langle \cdots \rangle$ in one case, and return inr inl $\langle \cdots \rangle$ in the other cases. The same "control fragment" is written differently depending on where it occurs. This makes moving code into and out of the inner loop error-prone.

(ii) return inl $\langle \cdots \rangle$ is used to resume both the inner and the outer iteration. To find out where control resumes, a reader of the program must carefully look up the innermost enclosing iteration. In contrast, in the Java program there can be no mistake about where control resumes after continue outer.

This awkwardness is exacerbated when there are three or more nested loops with the same structure: on the right hand side,

- return inl $(\cdots)$ would continue the *innermost enclosing* iteration
- return inr inl $(\cdots)$ would continue the *second-innermost enclosing* iteration
- return inr inr inl $(\cdots)$ would continue the *third-innermost enclosing* iteration, which would always be the outer iteration.

We call this the "De Bruijn index awkwardness", because De Bruijn's indices [6] for identifiers in λ calculus also work by counting intermediate binders, and they have similar disadvantages for programmers. Indeed, De Bruijn [6] claims his notation to be good for a number of things, but does not claim that it is "easy to write and easy to read for the human reader". For a brief introduction to De Bruijn indices, we refer to [14]; the same issue has also been studied from a different angle in [4,22].

1.4 The solution: Labelled iteration

We solve the De Bruijn index awkwardness with a second iteration construct, which we call labelled iteration and which we will also spell iter. It binds a name $x : A$ with a dual purpose:

- It holds a *value* of type A.
- It serves as a label for *restarting the loop*, upon which a new value of type A must be supplied.

```
    int sum = 0;
outer: for (int i = 0; i ≤ 8; i++){
        if (a[i][0] == 5)
            continue outer;
        int prod = 1;
inner:  for (int j = 0; j ≤ 8; j++){
            if (¬isEven(a[i][j]))
                continue inner;
            if (a[i][j] == 0)
                // product will be 0.
                continue outer;
            prod = prod * a[i][j];
        }
        sum = sum + prod;
    }
    int result = sum;
```

```
iter ⟨0, 0⟩, ℓ1.  let ⟨sum, i⟩ = ℓ1 in
    if i ≥ 9 then
        return sum
    else if a[i][0] == 5 then
        raise_ℓ1 ⟨sum, i + 1⟩
    else iter ⟨1, 0⟩, ℓ2.  let ⟨prod, j⟩ = ℓ2 in
        if j ≥ 9 then
            raise_ℓ1 ⟨sum + prod, i + 1⟩
        else if ¬isEven a[i][j] then
            raise_ℓ2 ⟨prod, j + 1⟩
        else if a[i][j] == 0 then
            raise_ℓ1 ⟨sum, i + 1⟩
        else
            raise_ℓ2 ⟨prod * a[i][j], j + 1⟩
```

Figure 2. Two programs programs that compute the same formula. The left hand program is written in Java; the right hand program uses fine-grain call-by-value with *labelled* iteration. Related fragments have the same colour. Both programs compute $\sum_{\substack{0 \le i \le 8 \\ \wedge\, a[i][0] \neq 5}} \prod_{\substack{0 \le j \le 8 \\ \wedge\, a[i][j] \text{ even}}} a[i][j]$.

In Figure 2, we have put the same Java program side-by-side side with an implementation using labelled iteration.

Like sum-based iteration, labelled iteration has a set-based semantics, but the type system is more involved. We explain labelled iteration in more detail in Section 3. We chose the spelling raise because there is a similarity with raising an exception; see also the discussion in Section 4.

1.5 Contributions

We define both languages: we give a type system, denotational semantics, big-step operational semantics, and an adequacy theorem for both languages. We explain the De Bruijn index awkwardness with the first language, and give a realistic example.

We show that the first construct can be macro-expressed in terms of the second construct.

For both types of iteration, we study only loops with `continue`: we omit `break` because we believe it is a straightforward extension.

We define the language with sum-based iteration in Section 2, and the language with labelled iteration in Section 3.

2 Sum-based iteration

We define both our constructs in terms of fine-grain call-by-value or FGCBV [20], which is a variant of call-by-value lambda calculus that has a syntactic separation between values and computations, and in which the evaluation order is made explicit.

We explain FGCBV and sum-based iteration here. The syntax and type system of FGCBV is given in Figure 3. We give a simple set-based semantics with divergence:

$$\begin{aligned} [\![1]\!] &= \{\star\} \\ [\![\mathrm{nat}]\!] &= \mathbb{N} \\ [\![A + B]\!] &= [\![A]\!] + [\![B]\!] \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![A \to B]\!] &= [\![A]\!] \to ([\![B]\!] + \{\bot\}) \end{aligned} \qquad [\![\Gamma]\!] = \prod_{(x:A)\in\Gamma} [\![A]\!]$$

$$[\![\Gamma \vdash^{\mathsf{v}} V : A]\!] \in [\![\Gamma]\!] \to [\![A]\!]$$
$$[\![\Gamma \vdash^{\mathsf{c}} M : A]\!] \in [\![\Gamma]\!] \to ([\![A]\!] + \{\bot\})$$

The semantics of plain FGCBV and FGCBV with sum-based iteration are the same, except of course that the latter has an extra construct. We give big-step operational semantics for both languages in Figure 5. The adequacy statements are simple:

Proposition 2.1 (adequacy)

(i) *For each closed term M of plain FGCBV without iteration, there is a unique V such that $M \Downarrow \mathsf{return}\ V$, and $[\![M]\!]_\varnothing = \mathsf{inl}\ [\![V]\!]_\varnothing$.*

(ii) *For each closed term M of FGCBV with sum-based iteration, either*

- *there is a unique V such that $M \Downarrow \mathsf{return}\ V$, and $[\![M]\!]_\varnothing = \mathsf{inl}\ [\![V]\!]_\varnothing$, or*
- *M does not reduce to a terminal, and $[\![M]\!]_\varnothing = \mathsf{inr}\ \bot$.*

3 Labelled iteration with pure function types

3.1 Introduction

To fix the De Bruijn index awkwardness indicated in Section 1.3, we now give a language that has an effectful "labelled iteration" construct instead. The judgements

$$
\begin{array}{lrcl}
\text{values} & V, W & ::= & x \mid \langle\rangle \mid 0 \mid \mathsf{succ}\ V \mid \mathsf{inl}\ V \mid \mathsf{inr}\ V \mid \langle V, W\rangle \mid \lambda x.\ M \\
\text{computations} & M, N & ::= & \mathsf{return}\ V \mid \mathsf{let}\ V\ \mathsf{be}\ x.\ M \mid M\ \mathsf{to}\ x.\ N \\
 & & & \mid V\ W \mid \mathsf{case}\ V\ \mathsf{of}\ \{0.\ M; \mathsf{succ}\ x.\ N\} \\
 & & & \mid \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{inl}\ x.\ M; \mathsf{inr}\ y.\ N\} \mid \mathsf{case}\ V\ \mathsf{of}\ \langle x, y\rangle.\ M \\
\text{types} & A, B, C & ::= & 1 \mid \mathrm{nat} \mid A + B \mid A \times B \mid A \to B
\end{array}
$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash^{\mathsf{v}} x : A} \qquad \frac{}{\Gamma \vdash^{\mathsf{v}} \langle\rangle : 1} \qquad \frac{}{\Gamma \vdash^{\mathsf{v}} 0 : \mathrm{nat}} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : \mathrm{nat}}{\Gamma \vdash^{\mathsf{v}} \mathsf{succ}\ V : \mathrm{nat}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{v}} \mathsf{inl}\ V : A + B} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : B}{\Gamma \vdash^{\mathsf{v}} \mathsf{inr}\ V : A + B} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Gamma \vdash^{\mathsf{v}} W : B}{\Gamma \vdash^{\mathsf{v}} \langle V, W\rangle : A \times B}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{c}} \mathsf{return}\ V : A} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Gamma, x : A \vdash^{\mathsf{c}} M : B}{\Gamma \vdash^{\mathsf{c}} \mathsf{let}\ V\ \mathsf{be}\ x.\ M : B} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : A \qquad \Gamma, x : A \vdash^{\mathsf{c}} N : B}{\Gamma \vdash^{\mathsf{c}} M\ \mathsf{to}\ x.\ N : B}$$

$$\frac{\Gamma, x : A \vdash^{\mathsf{c}} M : B}{\Gamma \vdash^{\mathsf{v}} \lambda x.\ M : A \to B} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \to B \qquad \Gamma \vdash^{\mathsf{v}} W : A}{\Gamma \vdash^{\mathsf{c}} V\ W : B}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : \mathrm{nat} \qquad \Gamma \vdash^{\mathsf{c}} M : C \qquad \Gamma, x : \mathrm{nat} \vdash^{\mathsf{c}} N : C}{\Gamma \vdash^{\mathsf{c}} \mathsf{case}\ V\ \mathsf{of}\ \{0.\ M; \mathsf{succ}\ x.\ N\} : C}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A + B \qquad \Gamma, x : A \vdash^{\mathsf{c}} M : C \qquad \Gamma, y : B \vdash^{\mathsf{c}} N : C}{\Gamma \vdash^{\mathsf{c}} \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{inl}\ x.\ M; \mathsf{inr}\ y.\ N\} : C}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \times B \qquad \Gamma, x : A, y : B \vdash^{\mathsf{c}} M : C}{\Gamma \vdash^{\mathsf{c}} \mathsf{case}\ V\ \mathsf{of}\ \langle x, y\rangle.\ M : C}$$

Addition for sum-based iteration

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Gamma, x : A \vdash^{\mathsf{c}} M : A + B}{\Gamma \vdash^{\mathsf{c}} \mathsf{iter}\ V,\ x.\ M : B}$$

Figure 3. Above: syntax of plain fine-grain call-by-value. Sum-based iteration adds only one term construct and no values or types; the type derivation of this term is given below.

in this language are

$$\begin{array}{rl}
\Delta; \Gamma \vdash^{\mathsf{c}} M : A & \text{for computations} \\
\Gamma \vdash^{\mathsf{v}} V : A & \text{for values}
\end{array}$$

We give the typing rules in Figure 6. Γ is a context of identifiers bound to values, as usual. Δ exists only for computations; it is a context of *typed labels*. Denotations

Fine-grain call-by-value

$$\begin{aligned}
[\![x]\!]_\rho &= \rho(x) & [\![\langle V, W\rangle]\!]_\rho &= \langle [\![V]\!]_\rho, [\![W]\!]_\rho\rangle \\
[\![\langle\rangle]\!]_\rho &= \langle\rangle & [\![\lambda x.\ M]\!]_\rho &= \lambda(a{\in}[\![A]\!]).[\![M]\!]_{(\rho,x\mapsto a)} \\
[\![0]\!]_\rho &= 0 & [\![\mathsf{return}\ V]\!]_\rho &= \mathsf{inl}\ [\![V]\!]_\rho \\
[\![\mathsf{succ}\ V]\!]_\rho &= 1 + [\![V]\!]_\rho & [\![\mathsf{let}\ V\ \mathsf{be}\ x.\ M]\!]_\rho &= [\![M]\!]_{(\rho,x\mapsto[\![V]\!]_\rho)} \\
[\![\mathsf{inl}\ V]\!]_\rho &= \mathsf{inl}\ [\![V]\!]_\rho & [\![M\ \mathsf{to}\ x.\ N]\!]_\rho &= \begin{cases} [\![N]\!]_{(\rho,x\mapsto v)} & \text{if } [\![M]\!]_\rho = \mathsf{inl}\ v \\ \mathsf{inr}\ \bot & \text{if } [\![M]\!]_\rho = \mathsf{inr}\ \bot \end{cases} \\
[\![\mathsf{inr}\ V]\!]_\rho &= \mathsf{inr}\ [\![V]\!]_\rho & [\![V\ W]\!]_\rho &= [\![V]\!]_\rho\ [\![W]\!]_\rho
\end{aligned}$$

$$[\![\mathsf{case}\ V\ \mathsf{of}\ \{0.\ M;\ \mathsf{succ}\ x.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_\rho & \text{if } [\![V]\!]_\rho = 0 \\ [\![N]\!]_{(\rho,x\mapsto n)} & \text{if } [\![V]\!]_\rho = 1+n \end{cases}$$

$$[\![\mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{inl}\ x.\ M;\ \mathsf{inr}\ y.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_{(\rho,x\mapsto a)} & \text{if } [\![V]\!]_\rho = \mathsf{inl}\ a \\ [\![N]\!]_{(\rho,y\mapsto b)} & \text{if } [\![V]\!]_\rho = \mathsf{inr}\ b \end{cases}$$

$$[\![\mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle.\ M\}]\!]_\rho = [\![M]\!]_{(\rho,x\mapsto a,y\mapsto b)} \qquad \text{if } [\![V]\!]_\rho = \langle a, b\rangle$$

Addition for sum-based iteration

$$[\![\mathsf{iter}\ V, x.\ M]\!]_\rho = \begin{cases} \mathsf{inl}\ w & \text{if } \exists v_{0..k} \text{ s.t. } v_0 = [\![V]\!]_\rho \\ & \quad \wedge \forall i : [\![M]\!]_{(\rho,x\mapsto v_i)} = \mathsf{inl}\ \mathsf{inl}\ v_{i+1} \\ & \quad \wedge [\![M]\!]_{(\rho,x\mapsto v_k)} = \mathsf{inl}\ \mathsf{inr}\ w \\ \mathsf{inr}\ \bot & \text{if no such } v_{0..k} \text{ exists} \end{cases}$$

Figure 4. Denotational semantics of values and terms in fine-grain call-by-value, and semantics of the sum-based iteration construct.

of judgements are

$$[\![\Delta; \Gamma \vdash^{\mathsf{c}} A]\!] = \Big(\prod_{(x:B)\in\Gamma} [\![B]\!]\Big) \to \Big(\sum_{(y:C)\in\Delta} [\![C]\!] + [\![A]\!] + \{\bot\}\Big)$$

$$[\![\Gamma \vdash^{\mathsf{v}} A]\!] = \Big(\prod_{(x:B)\in\Gamma} [\![B]\!]\Big) \to [\![A]\!]\ .$$

Γ is used to form values.

$$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash^{\mathsf{v}} x : A}$$

Δ is used to form computations, much like raising an exception. However, conventionally, exception names come from a global set. Our "exception names", which we call labels, will be bound in the same way that identifiers are bound by λ.

Furthermore, when a label is raised, it must be parametrised by a value of the

Fine-grain call-by-value

$T ::= \mathsf{return}\ V$

$$\frac{}{\mathsf{return}\ V \Downarrow \mathsf{return}\ V} \qquad \frac{M[V/x] \Downarrow T}{\mathsf{let}\ V\ \mathsf{be}\ x.\ M \Downarrow T} \qquad \frac{M \Downarrow \mathsf{return}\ V \qquad N[V/x] \Downarrow T}{M\ \mathsf{to}\ x.\ N \Downarrow T}$$

$$\frac{M[W/x] \Downarrow T}{(\lambda x.\ M)\ W \Downarrow T} \qquad \frac{M[V/x, W/y] \Downarrow T}{\mathsf{case}\ \langle V, W\rangle\ \mathsf{of}\ \{\langle x, y\rangle.\ M\} \Downarrow T}$$

$$\frac{M_0 \Downarrow T}{\mathsf{case}\ 0\ \mathsf{of}\ \{0.\ M_0; \mathsf{succ}\ x.\ M_{\mathsf{succ}}\} \Downarrow T} \qquad \frac{M_{\mathsf{succ}}[V/x] \Downarrow T}{\mathsf{case}\ (\mathsf{succ}\ V)\ \mathsf{of}\ \{0.\ M_0; \mathsf{succ}\ x.\ M_{\mathsf{succ}}\} \Downarrow T}$$

$$\frac{M_{\mathsf{inl}}[V/x] \Downarrow T}{\mathsf{case}\ (\mathsf{inl}\ V)\ \mathsf{of}\ \{\mathsf{inl}\ x.\ M_{\mathsf{inl}}; \mathsf{inr}\ x.\ M_{\mathsf{inr}}\} \Downarrow T} \qquad \frac{M_{\mathsf{inr}}[V/x] \Downarrow T}{\mathsf{case}\ (\mathsf{inr}\ V)\ \mathsf{of}\ \{\mathsf{inl}\ x.\ M_{\mathsf{inl}}; \mathsf{inr}\ x.\ M_{\mathsf{inr}}\} \Downarrow T}$$

Addition for sum-based iteration

$T ::= \mathsf{return}\ V$

$$\frac{\exists k \geq 0\ \exists (V_1, \cdots, V_k)\ \forall i \in \{1..k\} : M[V_{i-1}/x] \Downarrow \mathsf{return}\ \mathsf{inl}\ V_i \qquad M[V_k/x] \Downarrow \mathsf{return}\ \mathsf{inr}\ Z}{\mathsf{iter}\ V_0,\ x.\ M \Downarrow \mathsf{return}\ Z}$$

Figure 5. Big-step operational semantics of plain fine-grain call-by-value and of sum-based iteration. In our operational semantics, closed terms reduce to "terminal" terms of the same type, or they do not reduce at all. We use metavariable T for terminals. For FGCBV and its extension with sum-based iteration, terminal terms are always of the form $\mathsf{return}\ V$. Introducing a separate notion of terminals might seem odd for now, but in Figure 8 we extend the rules for FGCBV and add another form of terminal. So T above may come to stand for something other than $\mathsf{return}\ V$ further in the paper.

corresponding type. The typing rule is as follows:

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad (x{:}A) \in \Delta}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{raise}_x\ V : B}$$

We thus have these judgements.

$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \vdash^{\mathsf{c}} \mathsf{raise}_x\ \langle 3, \mathrm{true}\rangle : \mathrm{string}$$
$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \vdash^{\mathsf{c}} \mathsf{raise}_x\ \langle y, z\rangle : 0$$
$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \vdash^{\mathsf{c}} \mathsf{return}\ y : \mathrm{nat}$$

But we cannot raise identifiers:

$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \nvdash^{\mathsf{c}} \mathsf{raise}_y\ 3$$

And we can also not use labels for their value:

$$(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \nvdash^{\mathsf{c}} \mathsf{return}\ x : \mathrm{nat} \times \mathrm{bool}$$

Indeed, the typing rule of return (see Figure 6 on the next page) shows that x is not

Values and types are the same as in fine-grain call-by-value in Figure 3 on page 133.

$$\text{computations} \quad M, N ::= \cdots \mid \mathsf{iter}\ V, x.\ M \mid \mathsf{raise}_x\ V$$

$$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash^{\mathsf{v}} x : A} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Delta;\ \Gamma, x{:}A \vdash^{\mathsf{c}} M : B}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{let}\ V\ \mathsf{be}\ x.\ M : B}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{return}\ V : A} \qquad \frac{\Delta; \Gamma \vdash^{\mathsf{c}} M : A \qquad \Delta;\ \Gamma, x{:}A \vdash^{\mathsf{c}} N : B}{\Delta; \Gamma \vdash^{\mathsf{c}} M\ \mathsf{to}\ x.\ N : B}$$

$$\frac{\cdot\ ;\ \Gamma, x{:}A \vdash^{\mathsf{c}} M : B}{\Gamma \vdash^{\mathsf{v}} \lambda x.\ M : A \to B} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \to B \qquad \Gamma \vdash^{\mathsf{v}} W : A}{\Delta; \Gamma \vdash^{\mathsf{c}} V\ W : B}$$

$$\frac{}{\Gamma \vdash^{\mathsf{v}} \langle\rangle : 1} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{v}} \mathsf{inl}\ V : A + B} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : B}{\Gamma \vdash^{\mathsf{v}} \mathsf{inr}\ V : A + B}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : \mathrm{nat} \qquad \Delta; \Gamma \vdash^{\mathsf{c}} M : C \qquad \Delta;\ \Gamma, x{:}A \vdash^{\mathsf{c}} N : C}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{case}\ V\ \mathsf{of}\ \{0.\ M; \mathsf{succ}\ x.\ N\} : C}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A + B \qquad \Delta;\ \Gamma, x{:}A \vdash^{\mathsf{c}} M : C \qquad \Delta;\ \Gamma, y{:}B \vdash^{\mathsf{c}} N : C}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{inl}\ x.\ M; \mathsf{inr}\ y.\ N\} : C}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \times B \qquad \Delta;\ \Gamma, x{:}A, y{:}B \vdash^{\mathsf{c}} M : C}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{case}\ V\ \mathsf{of}\ \langle x, y\rangle.\ M : C}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Delta, x{:}A;\ \Gamma, x{:}A \vdash^{\mathsf{c}} M : B}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{iter}\ V, x.\ M : B} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad (x{:}A) \in \Delta}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{raise}_x\ V : B}$$

Figure 6. Syntax of labelled iteration.

available in the context of the argument to return:

$$\frac{y{:}\mathrm{nat}, z{:}\mathrm{bool} \ \vdash^{\mathsf{v}}\ V : \mathrm{nat} \times \mathrm{bool}}{(x : \mathrm{nat} \times \mathrm{bool})\ ;\ (y{:}\mathrm{nat}, z{:}\mathrm{bool}) \ \vdash^{\mathsf{c}}\ \mathsf{return}\ V : \mathrm{nat} \times \mathrm{bool}}$$

Our use of a syntactically separate kind of names bears resemblance to the use of function names by Kennedy [16] for control.

Labelled iteration

We now wish to use labels to generalise the $\mathsf{iter}\ V, x.\ M$ from last section. Remember that previously when M reduces to

- $\mathsf{return}\ \mathsf{inl}\ V'$, then the loop should be re-tried with value V',
- $\mathsf{return}\ \mathsf{inr}\ W$, then the result of the loop is W.

Our new notation will *also* be $\mathsf{iter}\ V, x.\ M$. However, here x is *both* an identifier and a label:

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Delta, x{:}A;\ \Gamma, x{:}A \vdash^{\mathsf{c}} M : B}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{iter}\ V, x.\ M : B}$$

Now similarly when writing $\mathsf{iter}\ V, x.\ M$, when M reduces to

- $\mathsf{raise}_x\ V'$, then the loop should be re-tried with value V',
- $\mathsf{raise}_y\ W$, $(y \neq x)$ then the loop should be aborted and loop y should be re-tried with value W,
- $\mathsf{return}\ W$, then the result of the loop is W.

We wish to repeat that the same name x can appear in *both* Δ and Γ. We pose no general syntactic restriction on $(x{:}A) \in \Delta$ and $(x{:}B) \in \Gamma$ to have the same type. However, to be able to form $\mathsf{iter}\ V, x.\ M$, we must have x in both Δ and Γ of the same type.

We also wish to note at this point that we define the semantics of our language on the *binding diagrams*[8], that is, on the abstract syntax modulo α-equivalence.

Labelled iteration and λ

Now that contexts for computations are different from contexts for values, the conventional fine-grain call-by-value judgements have to be tweaked to work in this setting. The typing rule for return in Figure 6 is simple: when we move upwards from a computation to a value judgement we just forget about Δ.

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Delta; \Gamma \vdash^{\mathsf{c}} \mathsf{return}\ V : A}$$

But reversely, for λ, we have a choice: what should Δ be? We take what seems to be the only reasonable choice: to reset Δ to the empty context, $\cdot$.

$$\frac{\cdot;\ \Gamma, x{:}A \vdash^{\mathsf{c}} M : B}{\Gamma \vdash^{\mathsf{v}} \lambda x.\ M : A \to B}$$

Java agrees with this choice: it is a syntax error to write a labelled `continue` or `break` with a label outside of the current method [11]. From a programmer's perspective, this means that all functions are pure.

3.2 Denotational semantics

Recall that the semantics of term and value judgements is as follows.

$$[\![\Delta;\Gamma \vdash^{\mathsf{c}} A]\!] = \Big(\prod_{(x:B)\in\Gamma} [\![B]\!]\Big) \to \Big(\sum_{(y:C)\in\Delta} [\![C]\!] + [\![A]\!] + \{\bot\}\Big)$$
$$[\![\Gamma \vdash^{\mathsf{v}} A]\!] = \Big(\prod_{(x:B)\in\Gamma} [\![B]\!]\Big) \to [\![A]\!]$$

The denotation of types is as follows.

$$\begin{aligned} [\![1]\!] &= \{\star\} \\ [\![\mathrm{nat}]\!] &= \mathbb{N} \\ [\![A + B]\!] &= [\![A]\!] + [\![B]\!] \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![A \to B]\!] &= [\![A]\!] \to ([\![B]\!] + \{\bot\}) \end{aligned}$$

We give the semantics of terms and values in Figure 7. We use the following notation for elements of the ternary sum $(\sum_{(x:B)\in\Delta}[\![B]\!] + [\![A]\!] + \{\bot\})$:

(i) *return* a (for $a \in [\![A]\!]$) (compare to the term notation: $\mathsf{return}\ V$),

(ii) *raise*$_x$ b (for $b \in [\![B]\!]$) (compare to the term notation: $\mathsf{raise}_x\ V$),

(iii) $\bot$.

Definition 3.1 [weakening] We say that $\Delta';\Gamma'$ is *stronger* than $\Delta;\Gamma$ when $\Delta' \subseteq \Delta$ and $\Gamma' \subseteq \Gamma$. Alternatively, we say that $\Delta;\Gamma$ is *weaker* than $\Delta';\Gamma'$.

A term in a context is also a term in a weaker context, with the same derivation. A value in a context is also a value in a weaker context, with the same derivation.

Definition 3.2 [closedness]

(i) When $\cdot \vdash^{\mathsf{v}} V : A$, then we say that V is *closed*.

(ii) When $\Delta;\ \cdot \vdash^{\mathsf{c}} M : A$, then we say that M is *closed*.

Definition 3.3 A *substitution* (between two-zone contexts) $\sigma : \Delta';\Gamma' \to \Delta;\Gamma$ consists of two parts,

- for every label $(x : A) \in \Delta'$, a *label* $\sigma_{\mathrm{lab}}(x)$ of type A in Δ, and
- for every identifier $(x : A) \in \Gamma'$, a *value* $\sigma_{\mathrm{id}}(x)$ $(\Gamma \vdash^{\mathsf{v}} \sigma_{\mathrm{id}}(x) : A)$.

Remark 3.4 From a two-zone substitution $\sigma : \Delta';\Gamma' \to \Delta;\Gamma$ we can trivially obtain a one-zone substitution $\Gamma' \to \Gamma$. By abuse of notation, we also write σ for this obtained substitution on one-zone contexts. Similarly, from a one-zone substitution $\sigma : \Gamma' \to \Gamma$, we obtain trivially a two-zone substitution $\cdot\,;\Gamma' \to \cdot\,;\Gamma$, for which we also write σ.

$$
\begin{aligned}
[\![x]\!]_\rho &= \rho(x) \\
[\![\langle\rangle]\!]_\rho &= \langle\rangle \\
[\![0]\!]_\rho &= 0 \\
[\![\mathsf{succ}\ V]\!]_\rho &= 1 + [\![V]\!]_\rho \\
[\![\mathsf{inl}\ V]\!]_\rho &= \mathsf{inl}\ [\![V]\!]_\rho \\
[\![\mathsf{inr}\ V]\!]_\rho &= \mathsf{inr}\ [\![V]\!]_\rho \\
[\![\langle V, W\rangle]\!]_\rho &= \langle [\![V]\!]_\rho, [\![W]\!]_\rho\rangle \\
[\![\lambda x.\ M]\!]_\rho &= \lambda(a \in [\![A]\!]).[\![M]\!]_{(\rho, x\mapsto a)}
\end{aligned}
\qquad
\begin{aligned}
[\![\mathsf{return}\ V]\!]_\rho &= \mathit{return}\ [\![V]\!]_\rho \\
[\![\mathsf{raise}_x\ V]\!]_\rho &= \mathit{raise}_x\ [\![V]\!]_\rho \\
[\![\mathsf{let}\ V\ \mathsf{be}\ x.\ M]\!]_\rho &= [\![M]\!]_{(\rho, x\mapsto [\![V]\!]_\rho)} \\
[\![M\ \mathsf{to}\ x.\ N]\!]_\rho &= \begin{cases} [\![N]\!]_{(\rho, x\mapsto v)} & \text{if } [\![M]\!]_\rho = \mathit{return}\ v \\ \mathit{raise}_y\ w & \text{if } [\![M]\!]_\rho = \mathit{raise}_y\ w \\ \bot & \text{if } [\![M]\!]_\rho = \bot \end{cases} \\
[\![V\ W]\!]_\rho &= [\![V]\!]_\rho\ [\![W]\!]_\rho
\end{aligned}
$$

$$
[\![\mathsf{case}\ V\ \mathsf{of}\ \{0.\ M;\ \mathsf{succ}\ x.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_\rho & \text{if } [\![V]\!]_\rho = 0 \\ [\![N]\!]_{(\rho, x\mapsto n)} & \text{if } [\![V]\!]_\rho = 1 + n \end{cases}
$$

$$
[\![\mathsf{case}\ V\ \mathsf{of}\ \{\mathsf{inl}\ x.\ M;\ \mathsf{inr}\ y.\ N\}]\!]_\rho = \begin{cases} [\![M]\!]_{(\rho, x\mapsto a)} & \text{if } [\![V]\!]_\rho = \mathsf{inl}\ a \\ [\![N]\!]_{(\rho, y\mapsto b)} & \text{if } [\![V]\!]_\rho = \mathsf{inr}\ b \end{cases}
$$

$$
[\![\mathsf{case}\ V\ \mathsf{of}\ \{\langle x, y\rangle.\ M\}]\!]_\rho = [\![M]\!]_{(\rho, x\mapsto a, y\mapsto b)} \qquad \text{if } [\![V]\!]_\rho = \langle a, b\rangle
$$

$$
[\![\mathsf{iter}\ V, x.\ M]\!]_\rho = \begin{cases} \mathit{return}\ w & \text{if } \exists v_{0..k} \text{ s.t. } v_0 = [\![V]\!]_\rho \\ & \quad \wedge \forall i : [\![M]\!]_{(\rho, x\mapsto v_i)} = \mathit{raise}_x\ v_{i+1} \\ & \quad \wedge [\![M]\!]_{(\rho, x\mapsto v_k)} = \mathit{return}\ w \\ \mathit{raise}_y\ w & \text{if } \exists v_{0..k} \text{ s.t. } v_0 = [\![V]\!]_\rho \\ & \quad \wedge \forall i : [\![M]\!]_{(\rho, x\mapsto v_i)} = \mathit{raise}_x\ v_{i+1} \\ & \quad \wedge [\![M]\!]_{(\rho, x\mapsto v_k)} = \mathit{raise}_y\ w \\ \bot & \text{if no other case matches} \end{cases}
$$

Figure 7. Denotational semantics of terms and values of the language with labelled iteration. See also Section 3.2.

We can use a substitution $\sigma : \Delta'; \Gamma' \to \Delta; \Gamma$ as follows on terms. Given a term $\Delta'; \Gamma' \vdash^{\mathsf{c}} M : A$, we obtain the term $\Delta; \Gamma \vdash^{\mathsf{c}} M\sigma : A$ by

- for any $x \in \Delta$, replacing all occurrences of $\mathsf{raise}_x\ V$ (where x is free) by $\mathsf{raise}_{\sigma_{\mathrm{lab}}(x)}\ (V\sigma)$, where $V\sigma$ is given similarly by induction. And
- for any $x \in \Gamma$, replacing all value occurrences of identifiers by $\sigma_{\mathrm{id}}(x)$.

For one-zone contexts Γ we have the usual notion of substitution $\sigma : \Gamma' \to \Gamma$ that assigns a value (over Γ) to each identifier of Γ'. And given $\Gamma' \vdash^{\mathsf{v}} V : A$, we obtain similarly $\Gamma \vdash^{\mathsf{v}} V\sigma : A$.

Two-zone contexts and their substitutions form a category, and one-zone contexts and their substitutions form another category. That is, substitutions can be composed associatively and composition has an identity.

Lemma 3.5 (substitution lemma)

$T ::= \mathsf{return}\ V \mid \mathsf{raise}_x\ V$

$$\frac{M \Downarrow \mathsf{raise}_x\ V}{M\ \mathsf{to}\ x.\ N \Downarrow \mathsf{raise}_x\ V}$$

$$\frac{\exists k \geq 0\ \exists (V_1, \cdots, V_k)\ \forall i \in \{1..k\} : M[V_{i-1}/x] \Downarrow \mathsf{raise}_x\ V_i \qquad M[V_k/x] \Downarrow \mathsf{return}\ Z}{\mathsf{iter}\ V_0,\ x.\ M \Downarrow \mathsf{return}\ Z}$$

$$\frac{\exists k \geq 0\ \exists (V_1, \cdots, V_k)\ \forall i \in \{1..k\} : M[V_{i-1}/x] \Downarrow \mathsf{raise}_x\ V_i \qquad M[V_k/x] \Downarrow \mathsf{raise}_y\ Z}{\mathsf{iter}\ V_0,\ x.\ M \Downarrow \mathsf{raise}_y\ Z}\ (x \neq y)$$

Figure 8. Big-step operational semantics for labelled iteration. This figure extends Figure 5. Namely, we add rules, and we add a new form of terminal: $\mathsf{raise}_x\ V$.

(i) *Let one-zone substitution $\sigma : \Gamma' \to \Gamma$ be given. If $\Gamma' \vdash^{\vee} V : A$, then $[\![V\sigma]\!]_\rho = [\![V]\!]_{(x \mapsto [\![\sigma(x)]\!]_\rho)_{x \in \Gamma'}}$.*

(ii) *Let two-zone substitution $\sigma : \Delta'; \Gamma' \to \Delta; \Gamma$ be given. If $\Delta'; \Gamma' \vdash^{\mathsf{c}} M : A$, then $[\![M\sigma]\!]_\rho = f([\![M]\!]_{(x \mapsto [\![\sigma_{\mathrm{id}}(x)]\!]_\rho)_{x \in \Gamma'}})$, where f maps $\mathit{raise}_x\ v$ to $\mathit{raise}_{\sigma_{\mathrm{lab}}(x)}\ v$.*

3.3 Operational semantics

We define a big-step "reduction" relation $M \Downarrow T$ between closed terms $\Delta;\ \cdot\ \vdash^{\mathsf{c}} M : A$ and (closed) terminals $\Delta;\ \cdot\ \vdash^{\mathsf{c}} T : A$ of the same type, such that for every such M either

(i) $M \Downarrow T = \mathsf{return}\ V$, or

(ii) $M \Downarrow T = \mathsf{raise}_x\ V$, $x \in \mathrm{dom}\,\Delta$, or

(iii) M does not reduce.

Derivation rules are given in Figure 8, and the reduction relation is defined as their least fixed point.

Theorem 3.6 (adequacy)

(i) *If $M \Downarrow \mathsf{return}\ V$, then $[\![M]\!]_\varnothing = \mathit{return}\ [\![V]\!]_\varnothing$.*

(ii) *If $M \Downarrow \mathsf{raise}_x\ V$, then $[\![M]\!]_\varnothing = \mathit{raise}_x\ [\![V]\!]_\varnothing$.*

(iii) *If M does not reduce, then $[\![M]\!]_\varnothing = \bot$.*

3.4 Translation from sum-based iteration

Let $\Gamma \vdash^{\mathsf{c}} M : A$ or $\Gamma \vdash^{\mathsf{v}} V : A$ be a term or value in the language with sum-based iteration. We define a *translation* $\mathrm{translate}(M)$, $\mathrm{translate}(V)$ from sum-based iteration, such that $\cdot\,;\ \Gamma \vdash^{\mathsf{c}} \mathrm{translate}(M) : A$ or $\Gamma \vdash^{\mathsf{v}} \mathrm{translate}(V) : A$, respectively, in the language with labelled iteration. The translation macro-expands sum-based iter as follows. The other constructs are left unchanged.

$$\mathrm{translate}(\mathsf{iter}\ V, x.\ M) = \mathsf{iter}\ V, x.\ \big(\mathrm{translate}(M)\ \mathsf{to}\ \mathit{result}.\ \mathsf{case}\ \mathit{result}\ \mathsf{of}\ \{\mathsf{inl}\ y.\ \mathsf{raise}_x\ y; \mathsf{inr}\ x'.\ \mathsf{return}\ x'\}\big)$$

where $\mathrm{translate}(M)$ is implicitly weakened by adding x to Δ.

Theorem 3.7 (translation preserves semantics)

(i) *Let $\Gamma \vdash^{\mathsf{c}} M : A$ a term of the language with sum-based iteration, and $\rho \in [\![\Gamma]\!]$. Then $[\![M]\!]_\rho = [\![\mathrm{translate}(M)]\!]_\rho$.*

(ii) *Let $\Gamma \vdash^{\mathsf{v}} V : A$ a value of the language with sum-based iteration, and $\rho \in [\![\Gamma]\!]$. Then $[\![V]\!]_\rho = [\![\mathrm{translate}(V)]\!]_\rho$.*

Corollary 3.8 *If $M \Downarrow T$ in the language with sum-based iteration, then there is T' such that* $\mathrm{translate}(M) \Downarrow T'$ *in the language with labelled iteration, and $[\![T]\!]_\varnothing = [\![T']\!]_\varnothing$. And if M does not reduce to a terminal, then* $\mathrm{translate}(M)$ *does not reduce to a terminal.*

4 Discussion and related work

In our presentation of labelled iteration, we have chosen to only consider pure functions. It is an important future task to extend the present system so as to allow for functions that raise an iteration.

We have noticed the *De Bruijn index awkwardness* in settings other than iteration. For instance, it is customary in functional languages such as Haskell to use monad transformers [12] to embed imperative programs with multiple side-effects, but they suffer from a similar De Bruijn index awkwardness: the i^{th} monad transformer is addressed by writing "lift^i *effect*". This issue and proposed solutions have been studied in the literature [15,24,5], but addressing effects using labels seems yet unexplored. Imperative languages address mutable cells using identifiers, and it is possible that addressing effects with labels might benefit the readability of similar functional programs as well.

Many programming languages have not just unlabelled and labelled `continue`, after which we have modelled our combination of iter and raise, but also unlabelled and labelled `break`. It should be straightforward to introduce a construct that binds a label like iter, but when the label is raised with parameter a, the result of that construct is a, so that raise of that label resembles `break`. Such a construct, together

with the raise we used in this paper, resembles an intra-procedural form of exception handling. If we wrap an iter inside this new construct and use one label for breaking and one for continuing, we can "`break`" and "`continue`" from this combination of constructs, to deepen the resemblence with Java-style loops.

5 Conclusion

In the present article we summarize the essence of the sum-based representation of iteration, and evaluate it from a programming perspective. Although it might work well for a semantics standpoint, it is inadequate for programmers to program in. We propose an alternative representation of iteration that is suitable for programmers, but still has relatively clean semantics.

References

[1] Peter Aczel, Jiří Adámek, Stefan Milius, and Jiří Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theoretical Computer Science*, 300(1–3):1–45, May 2003.

[2] Jiří Adámek, Stefan Milius, and Jiří Velebil. Elgot Algebras. *Logical Methods in Computer Science*, 2(5), November 2006.

[3] Jiří Adámek, Stefan Milius, and Jiří Velebil. Elgot theories: a new perspective on the equational properties of iteration. *Mathematical Structures in Computer Science*, 21(Special Issue 02):417–480, April 2011.

[4] Stefan Berghofer and Christian Urban. A Head-to-Head Comparison of de Bruijn Indices and Names. *Electronic Notes in Theoretical Computer Science*, 174(5):53–67, June 2007.

[5] Edwin Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144, New York, NY, USA, 2013. ACM.

[6] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972.

[7] Calvin C. Elgot. Monadic Computation and Iterative Algebraic Theories. In H. E. Rose and J. C. Shepherdson, editors, *Studies in Logic and the Foundations of Mathematics*, volume 80 of *Logic Colloquium '73 Proceedings of the Logic Colloquium*, pages 175–230. Elsevier, 1975.

[8] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Symposium on Logic in Computer Science, 1999. Proceedings*, pages 193–202, 1999.

[9] Sergey Goncharov, Christoph Rauch, and Lutz Schröder. Unguarded Recursion on Coinductive Resumptions. In *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 183–198. Elsevier, December 2015.

[10] Sergey Goncharov, Lutz Schröder, and Till Mossakowski. Kleene Monads: Handling Iteration in a Framework of Generic Effects. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science*, number 5728 in Lecture Notes in Computer Science, pages 18–33. Springer Berlin Heidelberg, September 2009. DOI: 10.1007/978-3-642-03741-2_3.

[11] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java® language specification*. Addison-Wesley, Upper Saddle River, NJ, java se 8 edition, 2014.

[12] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925, pages 97–136. Springer, 1995.

[13] Yoshihiko Kakutani. Duality between Call-by-Name Recursion and Call-by-Value Iteration. In Julian Bradfield, editor, *Computer Science Logic*, number 2471 in Lecture Notes in Computer Science, pages 506–521. Springer Berlin Heidelberg, September 2002. DOI: 10.1007/3-540-45793-3_34.

[14] Fairouz Kamareddine and Alejandro Ríos. A λ-calculus à la de Bruijn with explicit substitutions. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics and Programs*, number 982 in Lecture Notes in Computer Science, pages 45–62. Springer Berlin Heidelberg, September 1995. DOI: 10.1007/BFb0026813.

[15] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM.

[16] Andrew Kennedy. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 177–190, New York, NY, USA, 2007. ACM.

[17] Dexter Kozen and Konstantinos Mamouras. Kleene Algebra with Products and Iteration Theories. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 415–431, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[18] Dexter Kozen and Konstantinos Mamouras. Kleene Algebra with Equations. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming*, number 8573 in Lecture Notes in Computer Science, pages 280–292. Springer Berlin Heidelberg, July 2014. DOI: 10.1007/978-3-662-43951-7_24.

[19] J. Laird. The Elimination of Nesting in SPCF. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, number 3461 in Lecture Notes in Computer Science, pages 234–245. Springer Berlin Heidelberg, April 2005. DOI: 10.1007/11417170_18.

[20] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, December 2006.

[21] John Longley. The recursion hierarchy for PCF is strict. Informatics Research Report EDI-INF-RR-1421, School of Informatics, University of Edinburgh, July 2015.

[22] Conor McBride and James McKinna. Functional Pearl: I Am Not a Number–I Am a Free Variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 1–9, New York, NY, USA, 2004. ACM.

[23] Stefan Milius and Tadeusz Litak. Guard Your Daggers and Traces: On The Equational Properties of Guarded (Co-)recursion. *Electronic Proceedings in Theoretical Computer Science*, 126:72–86, August 2013.

[24] Dominic Orchard and Tomas Petricek. Embedding Effect Systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 13–24, New York, NY, USA, 2014. ACM.

[25] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(02):198–212, August 1967.

A Appendix: proofs

We first prove adequacy of fine-grain call-by-value without iteration. The adequacy of FGCBV + sum-based iteration and the adequacy of the language with labelled iteration are then minor modifications. All our adequacy proofs are in the style of Tait [25].

We use the following substitution lemma for both plain FGCBV and FGCBV with sum-based iteration.

Lemma A.1 *Assume a substitution* $\sigma : \Gamma' \to \Gamma$ *and an environment* $\rho \in [\![\Gamma]\!]$.

(i) *Let* $\Gamma' \vdash^{\mathrm{V}} V : A$ *be a value. Then* $[\![V]\!]_{(x \mapsto [\![\sigma_x]\!]\rho)_{x \in \Gamma'}} = [\![V\sigma]\!]\rho$.

(ii) *Let* $\Gamma' \vdash^{\mathrm{C}} M : A$ *be a term. Then* $[\![M]\!]_{(x \mapsto [\![\sigma_x]\!]\rho)_{x \in \Gamma'}} = [\![M\sigma]\!]\rho$.

The proofs of both substitution lemmas, Lemma A.1 and Lemma 3.5, are routine and we omit them.

A.1 Adequacy of FGCBV without iteration

We prove adequacy with the help of the following type-indexed predicate on closed values and terms.

Definition A.2 By mutual induction on the type of V and M, respectively.

$$
\begin{array}{ll}
\text{when } \vdash^{\mathsf{v}} V : 1 : & P(V) \equiv \text{true} \\
\text{when } \vdash^{\mathsf{v}} V : \text{nat} : & P(V) \equiv \text{true} \\
\text{when } \vdash^{\mathsf{v}} V : A + B : & P(\mathsf{inl}\ V) \equiv P(V) \\
 & P(\mathsf{inr}\ V) \equiv P(V) \\
\text{when } \vdash^{\mathsf{v}} V : A \times B : & P(\langle V, W\rangle) = P(V) \wedge P(W) \\
\text{when } \vdash^{\mathsf{v}} V : A \to B : & P(\lambda x.\, M) \equiv \forall(\vdash^{\mathsf{v}} W : A) : P(W) \Rightarrow P(M[W/x]) \\
\text{when } \vdash^{\mathsf{c}} M : A : & P(M) \equiv \exists(\vdash^{\mathsf{v}} V : A) : \Big(P(V) \ \wedge\ M \Downarrow \mathsf{return}\ V \ \wedge\ [\![M]\!]_{\varnothing} = \mathsf{inl}\ [\![V]\!]_{\varnothing}\Big)
\end{array}
$$

Observe that $P(M)$ implies adequacy of M.

Proposition A.3

(i) *If* $\Gamma \vdash^{\mathsf{v}} V : A$, *and if for all* $(x{:}B) \in \Gamma$ *we have a closed* $\vdash^{\mathsf{v}} \sigma_x : B$ *satisfying* $P(\sigma_x)$, *then* $P(V\sigma)$.

(ii) *If* $\Gamma \vdash^{\mathsf{c}} M : A$, *and if for all* $(x{:}B) \in \Gamma$ *we have a closed* $\vdash^{\mathsf{v}} \sigma_x : B$ *satisfying* $P(\sigma_x)$, *then* $P(M\sigma)$.

Proof

By induction on the value or term. Here are some interesting and less interesting cases.

$V = x$) Then $V\sigma = \sigma_x$, which was assumed to satisfy P.

$M = \mathsf{return}\ V$) Trivially by induction.

$V = \lambda y.\, M$) We have to show that if $\vdash^{\mathsf{v}} W : A$ satisfies P, then $M[\sigma, W/y]$ satisfies P. By induction.

$M = \mathsf{let}\ V\ \mathsf{be}\ x.\, N$) We are allowed to assume $P(V\sigma)$, so the induction hypothesis gives us $P(N[\sigma, (V\sigma)/x])$. We know that $M\sigma$ and $N[\sigma, (V\sigma)/x]$ reduce to the same terminal. We know $[\![M\sigma]\!]_{\varnothing} = [\![N\sigma]\!]_{x \mapsto [\![V\sigma]\!]_{\varnothing}}$, which we know is equal to $[\![N[\sigma, (V\sigma)/x]]\!]_{\varnothing}$ by the substitution lemma. Now $P(N[\sigma, (V\sigma)/x])$ implies $P(M\sigma)$.

$M = V\ W$) Similarly.

$M = M'$ to $x.\ N$) From the induction, we get V such that $P(V)$ and $M'\sigma \Downarrow \mathsf{return}\ V$ and $[\![M'\sigma]\!]_\varnothing = \mathsf{inl}\ [\![V]\!]_\varnothing$. From the derivation rule and the induction, we get V' such that $P(V')$ and $N[\sigma, V/x] \Downarrow \mathsf{return}\ V'$, and $[\![N[\sigma, V/x]]\!]_\varnothing = \mathsf{inl}\ [\![V']\!]_\varnothing$.

By the substitution lemma, $[\![N\sigma[V/x]]\!]_\varnothing = [\![N\sigma]\!]_{x \mapsto [\![V]\!]_\varnothing}$, and because we know $[\![M'\sigma]\!]_\varnothing = \mathsf{inl}\ [\![V]\!]_\varnothing$, we know that by definition

$$[\![(M'\sigma)\ \mathsf{to}\ x.\ (N\sigma)]\!]_\varnothing = [\![N\sigma]\!]_{x \mapsto [\![V]\!]_\varnothing}\ .$$

This completes the proof for this case.

$M = \mathsf{case}\ V\ \mathsf{of}\ \cdots$) Depending on the type of V, but for every type trivially by case analysis on $V\sigma$.

□

Corollary A.4 *All closed values and terms satisfy P.*

Adequacy directly follows from this.

Observe that the only cases in which we essentially looked at the normal form of $M\sigma$ are $\mathsf{return}\ V$ and M to $x.\ N$. Specifically, we did not use the normal form of $M\sigma$ in the let case. This means that we can reuse most of the proof for FGCBV with sum-based iteration.

A.2 Adequacy of FGCBV + sum-based iteration

Similar structure. We redefine $P(M)$:

$$P(\vdash^{\mathsf{c}} M : A) = \exists(\vdash^{\mathsf{v}} V : A) : \Big((P(V)\ \wedge\ M \Downarrow \mathsf{return}\ V\ \wedge\ [\![M]\!]_\varnothing = \mathsf{inl}\ [\![V]\!]_\varnothing))$$
$$\vee\ (M \Downarrow\!\!\!\!/\ \wedge\ [\![M]\!]_\varnothing = \mathsf{inr}\ \bot)\Big)$$

We have the same proposition as Proposition A.3 in this case:

- The case $M = \mathsf{return}\ V$ is still trivial.
- For $M = M'$ to $x.\ N$, we have to consider the alternative case that $M'\sigma \Downarrow\!\!\!\!/$ and $[\![M'\sigma]\!]_\varnothing = \mathsf{inr}\ \bot$. This case is trivial.
- For iter, observe that every sequence $V_1, \cdots, V_k$ in the operational semantics corresponds uniquely to a sequence

  $$v_0 = [\![V\sigma]\!]_\varnothing, v_1 = [\![V_1]\!]_\varnothing, v_2 = [\![V_2]\!]_\varnothing, \cdots, v_k = [\![V_k]\!]_\varnothing$$

  for the denotational semantics, and the proof in that case is analogous to the proof for let.

  To prove that non-existence of a valid sequence $V_1, \cdots, V_k$ for the operational semantics implies the non-existence of a valid sequence $v_0, \cdots, v_k$, we instead prove

the contrapositive. Indeed, we have our initial $V\sigma$ already, and by induction on a valid sequence $v_0, \cdots, v_k$ together with the induction hypothesis, we obtain step by step our sequence $V_1, \cdots, V_k$. So now we also know that $\mathsf{iter}\ V, x.M \not\Downarrow$ implies $[\![\mathsf{iter}\ V, x.M]\!]_\varnothing = \mathsf{inr}\ \bot$.

A.3 Adequacy of the language with labelled iteration

Similar structure.

We redefine $P(M)$ again. Recall that M closed means that $\Delta; \cdot \vdash^{\mathsf{c}} M : A$.

$$
\begin{aligned}
P(M) \equiv \Big(& \big(\exists(\vdash^{\mathsf{v}} V : A) : \big(P(V) \ \wedge\ M \Downarrow \mathsf{return}\ V \ \wedge\ [\![M]\!]_\varnothing = \mathit{return}\ [\![V]\!]_\varnothing\big)\big) \\
& \vee \big(\exists((x{:}B) \in \Delta) : \exists(\vdash^{\mathsf{v}} V : B) : \big(P(V) \ \wedge\ M \Downarrow \mathsf{raise}_x\ V \ \wedge\ [\![M]\!]_\varnothing = \mathit{raise}_x\ [\![V]\!]_\varnothing\big)\big) \\
& \vee \big(M \not\Downarrow \ \wedge\ [\![M]\!]_\varnothing = \bot\big) \Big)
\end{aligned}
$$

We have a proposition analogous to Proposition A.3.

Proposition A.5

(i) *If* $\Gamma \vdash^{\mathsf{v}} V : A$, *and if for all* $(x{:}B) \in \Gamma$ *we have a closed* $\vdash^{\mathsf{v}} \sigma_x : B$ *satisfying* $P(\sigma_x)$, *then* $P(V\sigma)$.

(ii) *If* $\Delta; \Gamma \vdash^{\mathsf{c}} M : A$, *and if we have a substitution* $\sigma : \Delta'; \Gamma' \to \Delta; \cdot$ *such that* $P(\sigma_{\mathrm{id}}(x))$ *on all identifiers, then* $P(M\sigma)$.

- The additional case for sequencing is trivial.
- The case $P(\mathsf{raise}_x\ V)$ is trivial.
- The additional case for iteration is analogous.