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
1571-0661/

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

$x := V$;
while $(p(x))$ {
  $x := f(x)$;
}
return $g(x)$;
	iter V, x.
	  if $p(x)$
	  then return inl $f(x)$
	  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 inr W, then the loop is finished and its result is W. If the body evaluates to inl V', then we set x to V', and keep on evaluating the body until it evaluates to some 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 $\sum_{\substack{0 \le i \le 8 \\ \wedge a[i][0] \ne 5}} \prod_{\substack{0 \le j \le 8 \\ \wedge a[i][j] \text{ even}}} a[i][j]$,