Bachelor thesis
Computer Science

Radboud University

Continuations in functional programming languages

Author:
Janne van den Hout
s4610431

First supervisor/assessor:
prof. dr. Herman Geuvers
herman@cs.ru.nl

Second assessor:
dr. Freek Wiedijk
freek@cs.ru.nl

June 25, 2018

Abstract

Continuations and operators that are able to use these continuations, provide an elegant way of manipulating the control flow of programs. In this thesis we define a language containing an operator callcc that can capture its own continuation. Using the language, we define and evaluate programs to show how exceptions and backtracking are achieved with callcc. Next, we give a translation into continuation-passing style based on Plotkin's CPS translation [12] which eliminates the callcc operator. Just as Plotkin, we also define a colon translation that reduces some of the administrative redexes introduced by the translation. In addition we evaluate a translated program and assert that it evaluates to the same as the untranslated program. We observe that Plotkin's indifference theorem does not hold for our translation and propose a solution.

Contents

Chapter 1

Introduction

Continuations represent the program state of a program at any point during its evaluation. A continuation can therefore be seen as a representation of the rest of the program that is yet to be evaluated.

The programming language Scheme treats continuations as first-class objects. This makes it possible to capture the current continuation using the operator *call/cc*. The captured continuation saves the current program state. If later on the captured continuation is invoked, the earlier saved state is restored.

```
(+ 2 (call/cc
       (lambda (continuation)
         (+ 5 (continuation 4)))))
```

Using the *call/cc* operator, programs like the one above can be written. This program evaluates to 6.

Because continuations can be used to restore an older program state, they make it possible to jump to different places in the program. As a result continuations can be used to encode control mechanisms such as exceptions and backtracking in a language.

In this thesis we will study continuations and operators that can manipulate them. We are therefore going to define a small language miniFP$^+$ which will contain an operator callcc that is similar to the operator *call/cc* in Scheme. Furthermore, we define an operator throw that is used to invoke a continuation.

In Chapter 4 we introduce continuations which will be highlighted by some examples. Furthermore, the operator callcc will be explained in detail. We will look at the program mentioned above again and show how the result 6 is obtained. We will also evaluate other programs containing the operator to demonstrate how callcc can be used to model exceptions and backtracking. Next, we will define a translation into continuation-passing style for the terms of miniFP$^+$. This translation will be based on the translation Plotkin defined in [12]. The translation eliminates the callcc- and throw-terms. However, it introduces a lot of abstractions that make it difficult to establish a relation between reductions in the source term and reductions in the translated term. We will look at Plotkin's colon translation that aims to solve this problem. Moreover, we extend the colon translation to the terms of miniFP$^+$. Using this extended colon translation we will translate a program into continuation-passing style and compare its evaluation with the evaluation of the untranslated program.

Chapter 2

Preliminaries

2.1 λ-calculus

In the 1930's Alonzo Church developed the λ-calculus initially as a foundation for mathematics [2]. Later it became evident that the λ-calculus provides a simple model for computation. The simplicity arises from the fact that functions in the λ-calculus are "anonymous", meaning the functions do not have names and also only have one input. We will briefly summarize the λ-calculus here. For a complete definition we refer to Barendregt [1].

Definition 2.1.1. The λ-calculus consists of λ-terms that are constructed recursively from an infinite set of variables and can take one of the following forms:

a variable x

an abstraction $\lambda x.M$ where M is a λ-term

an application MN where M and N are both λ-terms

The syntax of λ-terms M and N in the λ-calculus can therefore be given by:

$$M, N ::= x \mid \lambda x.M \mid MN$$

Definition 2.1.2 (β-rule)**.** The most basic form of computation in the λ-calculus is β-reduction. If an application is of the form $(\lambda x.M)\ N$ it can be rewritten using the β-rule.

$$(\lambda x.M)\ N \ \rightarrow\ M\ [x \quad N]$$

N is substituted for every occurrence of the variable x in M. The term $(\lambda x.M)\ N$ is called a *redex*, a reducible expression, because it can be rewritten by the β-rule to $M[x \quad N]$.

Example 2.1.1. Consider the term $(\lambda x.\lambda y.y\ x)((\lambda x.x)\ 1)(\lambda x.x)$ and assume we added integers to the λ-terms. Next, we will show the reduction steps. The redexes being reduced at each step are bold.

$$\begin{aligned} & \boldsymbol{(\lambda x.\lambda y.y\ x)((\lambda x.x)1)}(\lambda x.x) \\ \rightarrow\ & (\lambda y.y\ \boldsymbol{((\lambda x.x)1)})(\lambda x.x) \\ \rightarrow\ & \boldsymbol{(\lambda y.y\ 1)(\lambda x.x)} \\ \rightarrow\ & \boldsymbol{(\lambda x.x)1} \\ \rightarrow\ & 1 \end{aligned}$$

In the first step the redex $(\lambda x.\lambda y.y\ x)((\lambda x.x)\ 1)$ reduces to $(\lambda y.y\ ((\lambda x.x)\ 1))$ by substituting $((\lambda x.x)\ 1)$ for every occurrence of x in $(\lambda y.y\ x)$. In the second step $((\lambda x.x)\ 1)$ reduces to 1. $(\lambda x.x)$ is then substituted for y and finally $(\lambda x.x)\ 1$ reduces to 1 which does not reduce any further.

As the reader may have noticed there are other redexes that we could have chosen in the different reduction steps. For example, at the beginning $((\lambda x.x)\ 1)$ could have been reduced first. Also, in the second step we could have substituted $(\lambda x.x)$ for y instead of reducing $((\lambda x.x)\ 1)$ first. The choice which redex will be evaluated can be fixed in a reduction strategy. Next, we will look at the call-by-value and call-by-name reduction strategy.

2.2 Call-by-value

When evaluating call-by-value, the λ-terms are differentiated into *terms* and *values*.

$$\begin{array}{lll} \text{Terms:} & M, N & ::= x \mid \lambda x.M \mid MN \\ \text{Values:} & v & ::= \lambda x.M \end{array}$$

Definition 2.2.1 (Call-by-value SOS)**.** The call-by-value structural operational semantics (SOS) assure that β-reduction is only performed if the right part of the application is a value. Furthermore, there are no reductions under a λ-abstraction.

$$(\lambda x.M)\ v \ \rightarrow\ M[x := v] \qquad \frac{M \ \rightarrow\ M'}{MN \ \rightarrow\ M'N} \qquad \frac{N \ \rightarrow\ N'}{v\ N \ \rightarrow\ v\ N'}$$

If the left part M of an application $M\ N$ is not an abstraction, evaluation first proceeds with M. We write $M \ \rightarrow\ M'$ to express that M reduces to M' by one of the three rules above. The whole application $M\ N$ thereby reduces to $M'\ N$. The case that M is an abstraction is similar; N is reduced until it eventually becomes a value and the β-rule can be applied.

Example 2.2.1. We will consider the term from Example 2.1.1 again and evaluate it call-by-value assuming that the integer 1 is a value.

$$\begin{array}{ll} & (\lambda x.\lambda y.y\ x)\mathbf{((\lambda x.x)1)}(\lambda x.x) \\ \rightarrow & \mathbf{(\lambda x.\lambda y.y\ x)1}(\lambda x.x) \\ \rightarrow & \mathbf{(\lambda y.y\ 1)(\lambda x.x)} \\ \rightarrow & \mathbf{(\lambda x.x)1} \\ \rightarrow & 1 \end{array}$$

Using the call-by-value semantics we can see that the only redex in the original term is $(\lambda x.x)\ 1$. This term is not a value yet and therefore the β-rule cannot be applied to $(\lambda x.\lambda y.y\ x)\ ((\lambda x.x)1)$.

2.3 Call-by-name

Call-by-name is an other reduction strategy where β-reduction is performed as soon as possible. There is no notion of terms that are values. The syntax of the λ-terms is therefore the same as in Definition 2.1.1.

$$\text{Terms: } M, N ::= x \mid \lambda x.M \mid MN$$

Definition 2.3.1 (Call-by-name SOS)**.** β-reduction can be performed whenever the left part of an application is an abstraction. If this is not the case then the left part is evaluated.

$$(\lambda x.M)\ N \ \rightarrow\ M[x := N] \qquad \frac{M \ \rightarrow\ M'}{MN \ \rightarrow\ M'N}$$

Example 2.3.1. We will look at Example 2.1.1 again and evaluate the term call-by-name where the integer 1 is added to the terms of the λ-calculus.

$$\begin{aligned} & \mathbf{(\lambda x.\lambda y.y\ x)((\lambda x.x)1)}(\lambda x.x) \\ \rightarrow\ & \mathbf{(\lambda y.y\ ((\lambda x.x)1))(\lambda x.x)} \\ \rightarrow\ & \mathbf{(\lambda x.x)((\lambda x.x)1)} \\ \rightarrow\ & \mathbf{(\lambda x.x)1} \\ \rightarrow\ & 1 \end{aligned}$$

2.4 Evaluation contexts

Another way to define the call-by-value semantics of λ-terms is by using evaluation contexts described by Felleisen and Hieb in [4].

Definition 2.4.1 (Evaluation contexts)**.** An evaluation context is "a term with a hole".

$$E ::= [\,] \mid E\ M \mid v\ E$$

We write $E[a]$ to indicate that the hole $[\,]$ in the evaluation context E has been replaced with the term a.

To define reduction we will use the naming "head reduction" Leroy [9] uses in his lecture slides.

Definition 2.4.2 (Head reduction)**.** We define head reduction as the reduction of a term a using a set of head reduction rules. As the terms only consists of λ-abstractions and applications, the only head reduction rule is the β-rule.

$$(\lambda x.a)\ v \ \rightarrow\ a\ [x := v]$$

Because the evaluation is call-by-value, the argument to the abstraction has to be a value.

Definition 2.4.3 (Reduction)**.** We define reduction as the head reduction of a term inside the hole of a context.

$$\frac{a \ \rightarrow\ a'}{E[a] \ \rightarrow\ E[a']}$$

If the term a, that is plugged into the hole of the context E, reduces by head reduction to a term a' then we can replace the hole in E by a'.

Example 2.4.1. We will evaluate the term from Example 2.1.1 using evaluation contexts.

$$
\begin{array}{lll}
 & E[((\lambda x.x)\ 1)] & \text{with } E = (\lambda x.\lambda y.y\ x)[\,](\lambda x.x) \\
! & E[1] & \text{with } E = (\lambda x.\lambda y.y\ x)[\,](\lambda x.x) \\
= & E_1[(\lambda x.\lambda y.y\ x)\ 1] & \text{with } E_1 = [\,](\lambda x.x) \\
! & E_1[\lambda y.y\ 1] & \text{with } E_1 = [\,](\lambda x.x) \\
= & E_2[(\lambda y.y\ 1)(\lambda x.x)] & \text{with } E_2 = [\,] \\
! & E_2[(\lambda x.x)\ 1] & \text{with } E_2 = [\,] \\
! & E_2[1] & \text{with } E_2 = [\,]
\end{array}
$$

At the beginning the term $(\lambda x.\lambda y.y\ x)((\lambda x.x)\ 1)(\lambda x.x)$ corresponds to the evaluation context $E = (\lambda x.\lambda y.y\ x)[\,](\lambda x.x)$. The hole $[\,]$ denotes the redex we are reducing, in this case $((\lambda x.x)\ 1)$. Using β-reduction the term reduces to 1. Since we cannot reduce 1 any further we take the following redex $(\lambda x.\lambda y.y\ x)\ 1$. However, this means that the evaluation context has changed. The hole in the context now denotes the new redex $(\lambda x.\lambda y.y\ x)\ 1$. After the fourth reduction we are finished because 1 is a value and does not reduce further and E_2 only consists of a hole.

Chapter 3

miniFP$^+$

In this chapter the language miniFP$^+$ is defined. We will first give its syntax and call-by-value operational semantics. We then continue with typing miniFP$^+$. The syntax of miniFP$^+$ and the operational semantics are an extension of the definitions in the lecture slides of Leroy [9, 10]. Given miniFP$^+$ we will define a new language miniFP.

3.1 Syntax

We will now give the syntax of miniFP$^+$. Since we want miniFP$^+$ to evaluate call-by-value, terms and values have to be distinguished.

Definition 3.1.1 (miniFP$^+$)**.** We define the terms and values of miniFP$^+$ as follows.

Terms:	$a, b, c ::= 0$		
		$\mid \mathsf{Succ}(a)$	successor of a
		$\mid ()$	
		$\mid \mathsf{None}$	
		$\mid \mathsf{Some}\ a$	
		$\mid \mathsf{true}$	
		$\mid \mathsf{false}$	
		$\mid x$	variable
		$\mid \lambda x.a$	abstraction
		$\mid \mu f.\lambda x.a$	recursive function
		$\mid a\ b$	function application
		$\mid a\ +\ b$	addition
		$\mid \mathsf{Cons}(a, b)$	list constructor
		$\mid \mathsf{Nil}$	empty list
		$\mid \mathsf{let}\ x = v\ \mathsf{in}\ a$	
		$\mid \mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b$	
		$\mid \mathsf{match}\ a\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ b \mid \mathsf{Cons}(c_1, c_2)\ \ !\ \ c$	pattern matching
		$\mid a\ ;\ b$	composition

j zero? a	check for zero
j callcc a	
j throw a b	

Values: $v ::= 0 \mid \mathsf{Succ}(v) \mid () \mid \mathsf{None} \mid \mathsf{Some}\ v \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{Nil} \mid \mathsf{Cons}(v_1, v_2) \mid \lambda x.a \mid \mu f.\lambda x.a$

The language miniFP is defined as the language miniFP$^+$ without the terms callcc a and throw a b.

3.2 Operational semantics

We now define the call-by-value operational semantics of miniFP$^+$ using evaluation contexts.

Definition 3.2.1 (Evaluation contexts)**.** For miniFP$^+$ we define the following evaluation contexts.

$$E ::= [\,] \mid E\ b \mid v\ E \mid E + b \mid v + E \mid \mathsf{Succ}(E) \mid \mathsf{Some}\ E \mid \mathsf{Cons}(E, b) \mid \mathsf{Cons}(v, E)$$
$$\mid \mathsf{match}\ E\ \mathsf{with\ Nil}\ \to\ b \mid \mathsf{Cons}(c_1, c_2)\ \to\ c \mid \mathsf{if}\ E\ \mathsf{then}\ a\ \mathsf{else}\ b \mid \mathsf{zero?}\ E$$

Definition 3.2.2 (Reduction rules)**.** We define head reduction rules and context reduction rules for miniFP$^+$. The context reduction rules are defined for the terms callcc and throw as they operate on the whole context. The composition $a\ ;\ b$ is evaluated by rewriting it to $(\lambda x.\lambda y.y)\ a\ b$. This way a is evaluated for possible side effects that might occur when using operators such as callcc and throw. The result of a, however, will be thrown away.

Head reduction rules:

$(\lambda x.a)v$	$\to\ a[x := v]$
$(\mu f.\lambda x.a)v$	$\to\ a[f := \mu f.\lambda x.a, x := v]$
$0 + 0$	$\to\ 0$
$0 + \mathsf{Succ}(v)$	$\to\ \mathsf{Succ}(v)$
$\mathsf{Succ}(v) + 0$	$\to\ \mathsf{Succ}(v)$
$\mathsf{Succ}(v_1) + \mathsf{Succ}(v_2)$	$\to\ \mathsf{Succ}(\mathsf{Succ}(v_1 + v_2))$
$\mathsf{let}\ x = v\ \mathsf{in}\ a$	$\to\ a[x := v]$
$\mathsf{if\ true\ then}\ a\ \mathsf{else}\ b$	$\to\ a$
$\mathsf{if\ false\ then}\ a\ \mathsf{else}\ b$	$\to\ b$
$\mathsf{match\ Nil\ with\ Nil}\ \to\ a \mid \mathsf{Cons}(c_1, c_2)\ \to\ c$	$\to\ a$
$\mathsf{match\ Cons}(b_1, b_2)\ \mathsf{with\ Nil}\ \to\ a \mid \mathsf{Cons}(c_1, c_2)\ \to\ c$	$\to\ c[c_1 := b_1, c_2 := b_2]$
$a\ ;\ b$	$\to\ (\lambda x.\lambda y.y)\ a\ b$
$\mathsf{zero?}\ 0$	$\to\ \mathsf{true}$
$\mathsf{zero?}\ v$	$\to\ \mathsf{false}$ if $v \neq 0$

Context reduction rules:

$E[\mathsf{callcc}\ a]$	$\to\ E[a\ (\lambda x.E[x])]$
$E[\mathsf{throw}\ k\ a]$	$\to\ k\ a$

Definition 3.2.3 (Reduction)**.** For miniFP$^+$ we define reduction as either the reduction of the whole context using one of the context reduction rules, or as the head reduction of a term inside the hole of a context:

$$\frac{a \ ! \ a^{\prime}}{E[a] \ ! \ E[a^{\prime}]}$$

The operational semantics for miniFP are the same as for miniFP$^+$ except miniFP does not have the context reduction rules regarding callcc and throw.

3.3 Typing miniFP$^+$

In this section we will type the terms of miniFP$^+$. That is for each term a we want to have $a : \sigma$ where σ is the type of a. Therefore the types possible in miniFP$^+$ are defined first.

Definition 3.3.1 (Types)**.**

Types:	τ, σ	::=	nat j bool j list j unit j Maybe τ j τ cont j var j τ_1 ! τ_2
Variables:	var	::=	α j β
Polymorphism:	$poly$	::=	$\forall var.poly$ j τ
Context:	Γ	::=	j $\Gamma, x : \tau$

nat is the type of 0 and the natural numbers being generated from applying the successor function to 0. The terms true and false have type bool. Nil and Cons(a, b) are of type list and () has type unit. We use Maybe τ to give a type to None and Some a. Furthermore, we introduce a continuation type τ cont which indicates that the continuation expects something of type τ and which also hides the return type of the continuation. This type is used when defining the types of callcc and throw. Since miniFP$^+$ contains let-expressions which we want to have polymorphic types, type variables and quantification over types are defined. Furthermore, we define a function type τ_1 ! τ_2 where τ_1 is the type of the argument and τ_2 the return type of the function.

The types for the terms of miniFP are the same except that the type τ cont is not needed since there are no callcc- or throw-terms in miniFP.

We now introduce typing contexts Γ which are sets of $x : \tau$; mappings from variables to types. Using the typing context the typing relation between the terms of miniFP$^+$ and the types is defined as follows.

$$\Gamma \vdash a : \tau$$

The typing relation indicates that a has type τ in the context Γ. a is called *well-typed* if $a : \tau$ can be produced using *typing rules*. A typing rule has the following form.

$$\frac{\Gamma_1 \vdash a_1 : \tau_1 \ ... \ \Gamma_n \vdash a_n : \tau_n}{\Gamma \vdash a : \tau}$$

The statements above the line are the *premises* that have to be fulfilled in order to obtain the conclusion $\Gamma \vdash a : \tau$. The typing rules for miniFP$^+$ are defined as follows.

Definition 3.3.2 (Typing rules)**.** The typing rules can be divided into *axioms* and *inference rules*. Axioms are typing rules without premises whereas inference rules have one or more premises.

Axioms:

$$\frac{}{\Gamma \vdash 0 : \mathsf{nat}} :_{0} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} :_{true} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} :_{false}$$

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} :_{unit} \qquad \frac{}{\Gamma \vdash \mathsf{None} : \mathsf{Maybe}\ \tau} :_{none} \qquad \frac{}{\Gamma \vdash \mathsf{Nil} : \mathsf{list}} :_{nil}$$

Inference rules:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} :_{var} \qquad \frac{\Gamma \vdash a : \tau}{\Gamma \vdash \mathsf{Some}\ a : \mathsf{Maybe}\ \tau} :_{some}$$

$$\frac{\Gamma \vdash a : \mathsf{nat}}{\Gamma \vdash \mathsf{Succ}(a) : \mathsf{nat}} :_{succ} \qquad \frac{\Gamma \vdash a : \mathsf{nat}}{\Gamma \vdash \mathsf{zero?}\ a : \mathsf{bool}} :_{zero?}$$

$$\frac{\Gamma \vdash a : \mathsf{nat} \qquad \Gamma \vdash b : \mathsf{nat}}{\Gamma \vdash a + b : \mathsf{nat}} :_{add} \qquad \frac{\Gamma \vdash a : \mathsf{nat} \qquad \Gamma \vdash b : \mathsf{list}}{\Gamma \vdash \mathsf{Cons}(a, b) : \mathsf{list}} :_{cons}$$

$$\frac{\Gamma, x : \sigma \vdash a : \tau}{\Gamma \vdash \lambda x.a : \sigma \ ! \ \tau} :_{abs} \qquad \frac{\Gamma \vdash a : \sigma \ ! \ \tau \qquad \Gamma \vdash b : \sigma}{\Gamma \vdash a\ b : \tau} :_{app}$$

$$\frac{\Gamma, f : \sigma \ ! \ \tau, x : \sigma \vdash a : \tau}{\mu f.\lambda x.a : \sigma \ ! \ \tau} :_{rec} \qquad \frac{\Gamma \vdash a : \sigma \qquad \Gamma \vdash b : \tau}{\Gamma \vdash a\ ;\ b : \tau} :_{comp}$$

$$\frac{\Gamma \vdash a : \tau\ \mathsf{cont} \ ! \ \tau}{\Gamma \vdash \mathsf{callcc}\ a : \tau} :_{callcc} \qquad \frac{\Gamma \vdash a : \sigma\ \mathsf{cont} \qquad \Gamma \vdash b : \sigma}{\Gamma \vdash \mathsf{throw}\ a\ b : \tau} :_{throw}$$

$$\frac{\Gamma \vdash \forall\alpha.\sigma}{\Gamma \vdash a : \sigma[\alpha \quad \tau]} :_{elim} \qquad \frac{\Gamma \vdash a : \sigma \qquad \alpha \notin \Gamma}{\Gamma \vdash a : \forall\alpha.\sigma} :_{intro}$$

$$\frac{\Gamma \vdash c : \mathsf{bool} \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash b : \tau}{\Gamma \vdash \mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b : \tau} :_{if}$$

$$\frac{\Gamma \vdash v : \forall\alpha.\sigma \qquad \Gamma, x : \forall\alpha.\sigma \vdash b : \tau \qquad \alpha \notin \Gamma}{\Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ b : \tau} :_{let\_poly}$$

$$\frac{\Gamma \vdash l : \mathsf{list} \qquad \Gamma \vdash e : \tau \qquad \Gamma, a : \mathsf{nat}, b : \mathsf{list} \vdash f : \tau}{\Gamma \vdash \mathsf{match}\ l\ \mathsf{with}\ \mathsf{Nil} \ ! \ e \mid \mathsf{Cons}(a, b) \ ! \ f : \tau} :_{match}$$

The typing rules for miniFP are the same as for miniFP^{+} except for the callcc- and throw-typing rules.

Harper and Lillibridge state in [6] that polymorphic let-terms in a language containing callcc are only well-typed if the bound expression is a value. In Appendix A.1 we give the example of Harper and Lillibridge of a term $e_0 : \mathsf{bool}$ with the property that the term evaluates to 0 which is, however, of type nat. The term e_0 is a term of the form $\mathsf{let}\ x = a\ \mathsf{in}\ b$ where a is not a value and contains a callcc.

Having typed miniFP$^+$ we can prove that every term can be uniquely decomposed into a context and redex.

Proposition 3.3.1. *For all terms t 2* miniFP$^+$*, if $\vdash t : \sigma$ and t 62 Values then there exists a unique E with either*

1. *$t = E[e]$ and e reduces by head reduction*
2. *$t = E[\mathsf{callcc}\ u]$ and $E[\mathsf{callcc}\ u]$ reduces by context reduction*
3. *$t = E[\mathsf{throw}\ u\ v]$ and $E[\mathsf{throw}\ u\ v]$ reduces by context reduction.*

Proof. The proof is by induction on t and a case analysis of t.
As an example we will give the proof for the case $t = \mathsf{Cons}(a, b)$. For the complete proof see Appendix A.2.

Case $\mathsf{Cons}(a, b)$:

- a 62 **Values** : From induction there exists a unique E^0 with either 1. $a = E^0[e^0]$ and e^0 reduces by head reduction or 2. $a = E^0[\mathsf{callcc}\ c]$ and $E^0[\mathsf{callcc}\ c]$ reduces by context reduction or 3. $a = E^0[\mathsf{throw}\ c\ d]$ and $E^0[\mathsf{throw}\ c\ d]$ reduces by context reduction.
  1. Then there exists a unique E and e such that $\mathsf{Cons}(a, b) = E[e]$ and e reduces by head reduction. Because $E = \mathsf{Cons}(E^0, b)$ and $e = e^0$.
  2. Then there exists a unique E such that $\mathsf{Cons}(a, b) = E[\mathsf{callcc}\ u]$ and $E[\mathsf{callcc}\ u]$ reduces by context reduction. Because $E = \mathsf{Cons}(E^0, b)$ and $u = c$, we have $E[\mathsf{callcc}\ c] = \mathsf{Cons}(E^0[\mathsf{callcc}\ c], b)$ which reduces by context reduction to $\mathsf{Cons}(E^0[c\ (\lambda x.E[x])], b)$.
  3. Then there exists a unique E such that $\mathsf{Cons}(a, b) = E[\mathsf{throw}\ u\ v]$ and $E[\mathsf{throw}\ u\ v]$ reduces by context reduction. Because $E = \mathsf{Cons}(E^0, b)$, $u = c$ and $v = d$, we have $E[\mathsf{throw}\ c\ d] = \mathsf{Cons}(E^0[\mathsf{throw}\ c\ d], b)$ which reduces by context reduction to $c\ d$.
- a 2 **Values** : The proof is the same as for the case a 62 Values only now the induction is over b.

□

In Proposition 3.3.1 for every term it was proved that it can be uniquely decomposed into a context and a redex. For every term this was also proved for the case that the term does not contain a callcc or throw. Therefore it follows that the unique decomposition property also holds for the terms of miniFP.

Chapter 4

Continuations

In this chapter we will introduce continuations. The content presented is based on the lecture slides of Leroy [10]. To introduce the concept of continuations we will, just as Leroy, use an example. Consider the small program $p = (1+2)+3$. When evaluating p, we will first evaluate the subexpression $(1 + 2)$ and then *continue* by adding 3 to the result. The continuation of $(1 + 2)$ is the rest of the computation that needs to be done to obtain the overall result of p. In the case of our example the rest that needs to be done is to add 3.

Definition 4.0.1 (Continuation)**.** In general, the continuation of a subexpression e of a program p is the computation that needs to be done after e has been evaluated in order to obtain the result of p. The continuation is a function that takes as argument the value of the subexpression and yields the value of the whole program.

$$\text{continuation} : \text{value of } e \rightarrow \text{value of } p$$

Looking back at our example the continuation of $(1 + 2)$ is the function $(\lambda x.x + 3)$. After $(1 + 2)$ has been evaluated its value 3 will be passed as argument to the continuation which will add 3 and so compute the overall value 6 of the program.

We can also think about continuations using evaluation contexts. Let the program p be equal to the evaluation context E with a subexpression e that replaces the hole.

$$p = E[e] \quad \text{with } E = ... [\,] ...$$

If e reduces then the continuation of e is $\lambda x.E[x]$. The result of e will be passed as argument to the continuation which in turn will pass it to the hole of the evaluation context.

Example 4.0.1. Consider again the program $p = (1 + 2) + 3$.

$$\begin{array}{ll} p = (1 + 2) + 3 & = E[1 + 2] \text{ with } E = [\,] + 3 \\ \rightarrow\ p_1 = 3 + 3 & = E_1[3 + 3] \text{ with } E_1 = [\,] \\ \rightarrow\ 6 & \end{array}$$

The continuation of $(1+2)$ is the function $\lambda x.E[x]$ which is the same as $\lambda x.x+3$. The continuation takes the result of $(1 + 2)$ as input and passes it to the hole of the evaluation context thereby leaving the program $p_1 = 3 + 3$ over for evaluation. The continuation of p_1 takes

the result of $(3 + 3)$ and passes it to the hole of the evaluation context. So the continuation becomes $\lambda x.E_1[x]$ and since the evaluation context is only the hole, the continuation is equal to the identity function $\lambda x.x$. The continuation gets passed the result 6 of p_1 and because it is the identity function it just returns 6 which is the overall result of the program p.

4.1 callcc

The functional programming language Scheme has an operator *call-with-current-continuation* or *call/cc* for short. *call/cc* takes as argument a function. This function is then applied to the continuation of the *call/cc*. If during evaluation of the function body the continuation is applied to an argument the current continuation is thrown away and the continuation of the *call/cc* is restored. The argument passed to the continuation becomes the result value of the *call/cc*.

For the language miniFP$^+$ we defined a similar operator callcc that gets hold of its continuation. This continuation is then passed to the argument of callcc which is a function expecting a continuation. Furthermore, we defined the operator throw to invoke a continuation on an argument. The semantics for callcc and throw were defined as follows.

$$\begin{array}{ll} E[\mathsf{callcc}\ a] & !\ E[a\ (\lambda x.E[x])] \\ E[\mathsf{throw}\ k\ a] & !\ k\ a \end{array}$$

callcc is evaluated by supplying a continuation to the function a. The continuation supplied is the current continuation of the callcc, which is $(\lambda x.E[x])$. This way callcc saves the current context and makes it possible to invoke it again later. throw, on the other hand, throws away the current context. By passing a to the continuation k, throw reinstates the context saved by the continuation k.

Example 4.1.1. We will consider the program from the introduction again. The program has slightly changed to fit the syntax of miniFP$^+$.

$$2 + \mathsf{callcc}\ (\lambda k.\ 5 + (\mathsf{throw}\ k\ 4))$$

We will show that the program evaluates to 6.

$$\begin{array}{lll} & E[2 + \mathsf{callcc}\ (\lambda k.\ 5 + (\mathsf{throw}\ k\ 4))] & \text{with } E = [\,] \\ = & E_1[\mathsf{callcc}\ (\lambda k.\ 5 + (\mathsf{throw}\ k\ 4))] & \text{with } E_1 = 2 + [\,] \\ ! & E_1[(\lambda k.\ 5 + (\mathsf{throw}\ k\ 4))\ (\lambda x.E_1[x])] & \text{with } E_1 = 2 + [\,] \\ ! & E_1[5 + (\mathsf{throw}\ (\lambda x.E_1[x])\ 4)] & \text{with } E_1 = 2 + [\,] \\ = & E_2[\mathsf{throw}\ (\lambda x.E_1[x])\ 4] & \text{with } E_2 = 2 + 5 + [\,] \\ ! & (\lambda x.E_1[x])\ 4 & \\ ! & E_1[4] & \text{with } E_1 = 2 + [\,] \\ = & E_3[2 + 4] & \text{with } E_3 = [\,] \\ ! & E_3[6] & \text{with } E_3 = [\,] \end{array}$$

By evaluating callcc its continuation is passed as argument to the function of callcc. The continuation ($\lambda x.E_1[x]$) or ($\lambda x.\ 2+x$) is to add the result of the callcc to 2. When evaluating throw the current context E_2 is thrown away. And the context E_1 that was initially saved by the callcc is restored with 4 being passed as argument. This way the result value of callcc is 4. And the overall result becomes 6.

Type of callcc and throw

We will now look in more detail at the types of callcc and throw. callcc has the type (τ cont $!$ τ) $!$ τ. It expects a function of type τ cont $!$ τ. The argument to this function is the continuation of the callcc. Because the return type of callcc is of type τ the continuation expects an argument of this type. The type of the continuation is therefore τ cont.

throw has the type σ cont $!$ σ $!$ τ. Two arguments will be passed to throw, a continuation with type σ cont and an element of type σ. throw will apply the continuation to the element which yields a type τ that is the type of the result of the further computation.

Example 4.1.2. To demonstrate the effect of using callcc we will consider the list_iter and find program Leroy [10] defines in his lecture slides. The two programs and their types are defined as follows.

$$
\begin{aligned}
&\text{list\_iter} : (\text{nat}\ \ !\ \ \text{unit})\ \ !\ \ \text{list}\ \ !\ \ \text{unit}\\
&\text{list\_iter} := \mu list\_iter.\lambda f.\lambda list.\ \text{match}\ list\ \text{with Nil}\ \ !\ \ ()\\
&\qquad \mid \text{Cons}(hd, tl)\ \ !\ \ f\ hd\ ;\ list\_iter\ f\ tl
\end{aligned}
$$

$$
\begin{aligned}
&\text{find} : (\text{nat}\ \ !\ \ \text{bool})\ \ !\ \ \text{list}\ \ !\ \ \text{Maybe nat}\\
&\text{find} := \lambda p.\lambda l.\ \text{callcc}\ (\lambda k.\ \text{list\_iter}\ (\lambda x.\ \text{if}\ p\ x\ \text{then}\\
&\qquad \text{throw}\ k\ (\text{Some}\ x)\ \text{else}\ ())\ l\ ;\ \text{None})
\end{aligned}
$$

The list_iter program takes a function and a list as input. It will iterate over the list and apply the function first to the head of the list and then continue by recursively calling the list_iter program again with the tail of the list. In case the list supplied to the program is empty (Nil), the program returns (). This corresponds to the return type of list_iter which is unit and () is of type unit. If the list is not empty, the composition $f\ hd\ ;\ list\_iter\ f\ tl$ is executed. As defined in the semantics the first part is only evaluated for possible side effects. So for the overall type to be unit, the second part of the composition $list\_iter\ f\ tl$ also has to be of type unit. This is the case, because of the recursion $list\_iter$ will eventually be called with an empty list and return () of type unit.

We will now look at the find program. The find program takes two arguments. The first argument is a predicate function that checks whether a natural number fulfills that predicate. The function is therefore of type nat $!$ bool. The second argument is the list that possibly contains an element fulfilling the predicate. The body consists of a callcc. Inside the callcc function the list_iter program is called on the list and on a function that checks whether the current element of the list fulfills the predicate. If this is the case, throw will pass that element to the continuation which will make the element the result of the callcc. The overall return type of find is therefore Maybe nat. This means the type of callcc has to be of type Maybe nat and its continuation k of type Maybe nat cont. Following the callcc type inference rule

the function passed to callcc is of type Maybe nat cont ! Maybe nat. The argument of the function will become the continuation of callcc and is therefore of type Maybe nat cont. The body of the function then has type Maybe nat. It consists of a composition of a call to the list_iter program and None. The second part of the composition needs to correspond to the overall type Maybe nat of the body. This holds because None is of type Maybe nat.

Now for the first part of the composition recall the type of the list_iter program.

$$\text{list\_iter: (nat ! unit) ! list ! unit}$$

Because list_iter has unit as return type the first part of the composition is of type unit as well. However, we also want to be able to return an element, other than None, of type Maybe nat that we have found in the list. The callcc and the throw in the function supplied to list_iter make it possible to return an element of type Maybe nat from inside the list_iter which would normally return () of type unit.

$$(\lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } ())$$

Because of the type of list_iter the function needs to have type nat ! unit. This means that throw k (Some x) inside the if statement is of type unit. However, if we look at the type inference rule for throw we see that this type is not relevant. Because Some x is of type Maybe nat and k is of type Maybe nat cont we can just pass Some x to k which recall is the continuation of the callcc. This way callcc returns the Some x which is of type Maybe nat from within the list_iter program. For the complete type derivation for list_iter and find see Appendix A.3.

Furthermore, in Appendix A.4 we evaluate a program $find\_one$ that executes the find program on a list and a predicate function that checks if an element is Succ(0).

$$\begin{aligned} find\_one = \ & \text{let list\_iter} = \mu list\_iter.\lambda f.\lambda list.\ \text{match } list \text{ with Nil ! } () \\ & \qquad \text{| Cons}(hd, tl)\ \text{!}\ f\ hd\ ;\ list\_iter\ f\ tl \\ & \text{in (let find} = \lambda p.\lambda l.\ \text{callcc } (\lambda k.\text{list\_iter } (\lambda x.\ \text{if } p\ x \text{ then} \\ & \qquad \text{throw } k\ (\text{Some } x) \text{ else } ())\ l;\ \text{None}) \\ & \text{in find } (\lambda x.\text{zero? Pred } x)\ \text{Cons(Succ(Succ(0)), Cons(Succ(0), Nil)))} \end{aligned}$$

The evaluation steps show how once Succ(0) is encountered throw restores the context saved by the callcc. The result Some Succ(0) is passed to this context, thereby making Some Succ(0) the overall result of the program.

Example 4.1.2 as well as the complete evaluation of the program $find\_one$ in Appendix A.4 show that callcc can be used to model *exceptions*. If an element that fulfills the predicate is found the current context and thereby the rest of the list is thrown away and the context saved by the callcc is restored. The result value of callcc becomes the element found in the list.

Furthermore, callcc can be used to implement *backtracking*. With regard to the $find\_one$ program this would mean resuming iteration over the list after an element fulfilling the predicate is found. This is achieved by adding a second *inner* callcc to the find program.

$$\begin{aligned} \text{find} := & \lambda p.\lambda l.\ \text{callcc } (\lambda k.\ \text{list\_iter } (\lambda x.\ \text{if } p\ x \text{ then} \\ & \quad \text{callcc } (\lambda k^{\emptyset}.\ \text{throw } k\ (\text{Some } (x,\ k^{\emptyset})) \text{ else } ())\ l;\ \text{None}) \end{aligned}$$

The continuation captured by this inner callcc saves the current context during iteration and thereby the current position in the list. By invoking this continuation, iteration over the list resumes starting from the saved position.

In Appendix A.5 we evaluate another program *print_all* that Leroy [10] defines in his lecture slides and that makes use of the modified find program to print all the elements fulfilling the predicate function.

$$
\begin{aligned}
\mathit{print\_all} = \ & \mathsf{let\ list\_iter} = \mu \mathit{list\_iter}.\lambda f.\lambda \mathit{list}.\ \mathsf{match}\ \mathit{list}\ \mathsf{with\ Nil}\ \ !\ \ () \\
& \qquad\qquad \mathsf{j\ Cons}(\mathit{hd}, \mathit{tl})\ \ !\ \ f\ \mathit{hd}\ ;\ \mathit{list\_iter}\ f\ \mathit{tl} \\
& \mathsf{in\ (let\ find} = \lambda p.\lambda l.\ \mathsf{callcc}\ (\lambda k.\ \mathsf{list\_iter}\ (\lambda x.\ \mathsf{if}\ p\ x\ \mathsf{then} \\
& \qquad\qquad \mathsf{callcc}\ (\lambda k'.\ \mathsf{throw}\ k\ (\mathsf{Some}\ (x, k'))\ \mathsf{else}\ ())\ l; \mathsf{None}) \\
& \mathsf{in\ (let\ printall} = \lambda p.\lambda l.\mathsf{match\ find}\ p\ l\ \mathsf{with\ None}\ \ !\ \ () \\
& \qquad\qquad \mathsf{j\ Some}\ (x, k)\ \ !\ \ \mathsf{print\_string}\ x\ ;\ \mathsf{throw}\ k\ () \\
& \mathsf{in\ printall}\ (\lambda x.\mathsf{zero?\ Pred}\ x)\ \mathsf{Cons}(\mathsf{Succ}(0), \mathsf{Cons}(\mathsf{Succ}(0), \mathsf{Nil}))))
\end{aligned}
$$

The evaluation shows how the two continuations captured by the callcc's make it possible to jump between printing and iterating over the list. If an element fulfilling the predicate is found throw invokes the continuation of the outer callcc with the element passed as argument. However, at the moment an element was found the inner callcc captured another continuation which saved the context and thereby the position in the list. After a found element is printed this continuation is invoked and iteration over the list resumes from the saved position. This way both Succ(0) values in the list are printed.

4.2 Continuation-passing style

Using continuations we can transform every term t into a function whose argument will hold the continuation of the term. After the computation of t is finished the result is passed to the continuation. The term t is said to be written in *continuation-passing style* or *CPS* for short.

Example 4.2.1. Consider the factorial function and assume that we have defined a Pred function that returns the predecessor of its argument and a Mult function that multiplies its two arguments.

$$\mathsf{FAC} = \mu f.\lambda x.\ \mathsf{if\ zero?}\ x\ \mathsf{then\ Succ}(0)\ \mathsf{else\ Mult}\ x\ f(\mathsf{Pred}\ x)$$

Translated into continuation-passing style the factorial function becomes:

$$
\begin{aligned}
\mathsf{FAC}_{cps} = & \mu f.\lambda x.\lambda k.\ \mathsf{if\ zero?}\ x\ \mathsf{then}\ k\ \mathsf{Succ}(0) \\
& \qquad \mathsf{else}\ f(\mathsf{Pred}\ x)(\lambda v.k\ (\mathsf{Mult}\ x\ v))
\end{aligned}
$$

The translation into continuation-passing style has given the function f an extra argument k which holds the continuation. In the case that $x = 0$, we can just pass Succ(0) to the continuation k. In the case that $x \neq 0$, we recursively call the factorial function. As f takes two arguments we supply a new x namely Pred x and a new continuation $(\lambda v.k\ (\mathsf{Mult}\ x\ v))$. This continuation will get passed the result of f Pred x. The result will have to be multiplied with the current value of x first, before the overall result is being passed to the continuation k.

We are now going to evaluate $\mathsf{FAC}_{cps}\ 3\ (\lambda i.i)$. For readability we will abbreviate the natural numbers.

$$1 = \mathsf{Succ}(0)$$
$$2 = \mathsf{Succ}(\mathsf{Succ}(0))$$
$$3 = \mathsf{Succ}(\mathsf{Succ}(\mathsf{Succ}(0)))$$
$$6 = \mathsf{Succ}(\mathsf{Succ}(\mathsf{Succ}(\mathsf{Succ}(\mathsf{Succ}(\mathsf{Succ}(0))))))$$

Only the essential reduction steps will be shown. We use $!$ to denote that we are doing multiple reductions.

$E[(\mu f.\lambda x.\lambda k.\ \mathsf{if\ zero?}\ x\ \mathsf{then}\ k\ 1\ \mathsf{else}\ f(\mathsf{Pred}\ x)(\lambda v.k\ (\mathsf{Mult}\ x\ v)))\ 3\ (\lambda i.i)]$
with $E = [\,]$

$!$ $E_1[\mathsf{if\ zero?}\ 3\ \mathsf{then}\ (\lambda i.i)\ 1\ \mathsf{else}\ \mathsf{FAC}_{cps}(\mathsf{Pred}\ 3)(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))]$
with $E_1 = [\,]$

$!$ $E_3[\mathsf{FAC}_{cps}(\mathsf{Pred}\ 3)(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))]$
with $E_3 = [\,]$

$!$ $E_4[\mathsf{if\ zero?}\ 2\ \mathsf{then}\ (\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ 1\ \mathsf{else}$
$\mathsf{FAC}_{cps}(\mathsf{Pred}\ 2)(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))]$
with $E_4 = [\,]$

$!$ $E_5[\mathsf{FAC}_{cps}(\mathsf{Pred}\ 2)(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))]$
with $E_5 = [\,]$

$!$ $E_6[\mathsf{if\ zero?}\ 1\ \mathsf{then}\ (\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))\ 1\ \mathsf{else}$
$\mathsf{FAC}_{cps}(\mathsf{Pred}\ 1)(\lambda v_2.(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))\ (\mathsf{Mult}\ 1\ v_2))]$
with $E_6 = [\,]$

$!$ $E_7[\mathsf{FAC}_{cps}(\mathsf{Pred}\ 1)(\lambda v_2.(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))\ (\mathsf{Mult}\ 1\ v_2))]$
with $E_7 = [\,]$

$!$ $E_8[\mathsf{if\ zero?}\ 0\ \mathsf{then}\ (\lambda v_2.(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))\ (\mathsf{Mult}\ 1\ v_2))\ 1\ \mathsf{else}$
$\mathsf{FAC}_{cps}(\mathsf{Pred}\ 0)(\lambda v_3.(\lambda v_2.(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))\ (\mathsf{Mult}\ 1\ v_2))\ (\mathsf{Mult}\ 0\ v_3))]$
with $E_8 = [\,]$

$!$ $E_9[(\lambda v_2.(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))\ (\mathsf{Mult}\ 1\ v_2))\ 1]$
with $E_9 = [\,]$

$!$ $E_9[(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))\ (\mathsf{Mult}\ 1\ 1)]$
with $E_9 = [\,]$

$!$ $E_9[(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ 1)]$
with $E_9 = [\,]$

$!$ $E_9[(\lambda i.i)\ (\mathsf{Mult}\ 3\ 2)]$
with $E_9 = [\,]$

! $E_9[(\lambda i.i)\ 6]$
with $E_9 = [\,]$

! $E_9[6]$
with $E_9 = [\,]$

We can see that the last call to FAC_{cps} gets passed the continuation $(\lambda v_2.(\lambda v_1.(\lambda v.(\lambda i.i)\ (\mathsf{Mult}\ 3\ v))\ (\mathsf{Mult}\ 2\ v_1))\ (\mathsf{Mult}\ 1\ v_2))$ which multiplies all the intermediate results. In the end, we see that the result 6 of the FAC_{cps} program gets passed to $(\lambda i.i)$ which was the initially supplied continuation.

4.2.1 CPS translation

In this section we are going to extend the translation into continuation-passing style from Leroy's [10] lecture slides to the terms of miniFP^+. The translation is based on Plotkin's [12] CPS translation and eliminates callcc- and throw-terms. Therefore the translated terms are terms of miniFP.

Definition 4.2.1 (Plotkin's CPS translation)**.** The translation function $\mathtt{J\,K}$ takes a term of miniFP^+ and translates it into a term in continuation-passing style. Furthermore, we define a function $\mathit{j}\ \mathit{j}$ that translates values.

$$
\begin{aligned}
\mathit{j}0\mathit{j} &= 0 \\
\mathit{j}\mathsf{true}\mathit{j} &= \mathsf{true} \\
\mathit{j}\mathsf{false}\mathit{j} &= \mathsf{false} \\
\left|()\right| &= () \\
\mathit{j}\mathsf{None}\mathit{j} &= \mathsf{None} \\
\mathit{j}\mathsf{Some}\ v\mathit{j} &= \mathsf{Some}\ \mathit{j}v\mathit{j} \\
\left|\mathsf{Succ}(v)\right| &= \mathsf{Succ}(\mathit{j}v\mathit{j}) \\
\mathit{j}\mathsf{Nil}\mathit{j} &= \mathsf{Nil} \\
\left|\mathsf{Cons}(v_1, v_2)\right| &= \mathsf{Cons}(\mathit{j}v_1\mathit{j}\ ,\mathit{j}v_2\mathit{j}) \\
\mathit{j}\lambda x.a\mathit{j} &= \lambda x.\mathtt{J}a\mathtt{K} \\
\mathit{j}\mu f.\lambda x.a\mathit{j} &= \mu f.\lambda x.\mathtt{J}a\mathtt{K} \\
\mathtt{J}v\mathtt{K} &= \lambda k.\ k\ \mathit{j}v\mathit{j} \\
\mathtt{J}x\mathtt{K} &= \lambda k.\ k\ x \\
\mathtt{J}\mathsf{Some}\ a\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ k\ \mathsf{Some}\ v_a) \\
\mathtt{J}\mathsf{zero?}\ a\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ k\ \mathsf{zero?}\ v_a) \\
\mathtt{J}\mathsf{Succ}(a)\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ k\ \mathsf{Succ}(v_a)) \\
\mathtt{J}\mathsf{Cons}(a, b)\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ k\ \mathsf{Cons}(v_a, v_b))) \\
\mathtt{J}a\ +\ b\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ k\ (v_a\ +\ v_b)) \\
\mathtt{J}a\ ;\ b\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ k\ v_b)) \\
\mathtt{J}a\ b\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b\ k)) \\
\mathtt{J}\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b\mathtt{K} &= \lambda k.\ \mathtt{J}c\mathtt{K}\ (\lambda v_c.\ \mathsf{if}\ v_c\ \mathsf{then}\ \mathtt{J}a\mathtt{K}\ k\ \mathsf{else}\ \mathtt{J}b\mathtt{K}\ k)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{J}\mathsf{let}\ x = v\ \mathsf{in}\ a\mathtt{K} &= \lambda k.\ \mathsf{let}\ x = \mathit{jvj}\ \mathsf{in}\ \mathtt{J}a\mathtt{K}\ k \\
\mathtt{J}\mathsf{match}\ l\ \mathsf{with}\ \mathsf{Nil}\ \ \mathit{!}\ \ e\ \mathit{j}\ \mathsf{Cons}(a,b)\ \ \mathit{!}\ \ f\mathtt{K} &= \lambda k.\ \mathtt{J}l\mathtt{K}\ (\lambda v_l.\ \mathsf{match}\ v_l\ \mathsf{with}\ \mathsf{Nil}\ \ \mathit{!}\ \ \mathtt{J}e\mathtt{K}\ k \\
&\qquad \mathit{j}\ \mathsf{Cons}(a,b)\ \ \mathit{!}\ \ \mathtt{J}f\mathtt{K}\ k) \\
\mathtt{J}\mathsf{callcc}\ a\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda f.\ f\ k\ k) \\
\mathtt{J}\mathsf{throw}\ a\ b\mathtt{K} &= \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b))
\end{aligned}
$$

As mentioned before, the function a supplied to callcc expects a continuation as argument. When translated to continuation-passing style, a therefore becomes a function expecting two arguments. Its own argument and a continuation, which for the case of callcc are the same. This is why $\mathtt{J}a\mathtt{K}$ is applied to $(\lambda f.\ f\ k\ k)$.

In [6] Harper and Lillibridge showed that let-terms have to be translated to $\lambda k.\mathsf{let}\ x = \mathit{jvj}\ \mathsf{in}\ a$ in order to be well-typed. We will therefore also use this translation instead of the more intuitive translation $\lambda k.\ \mathtt{J}v\mathtt{K}\ (\lambda x.\ \mathtt{J}a\mathtt{K}\ k)$.

Example 4.2.2. To demonstrate Plotkin's CPS translation we are going to translate the term $(\lambda x.x)\ 0$ into continuation-passing style.

$$
\begin{aligned}
\mathtt{J}(\lambda x.x)\ 0\mathtt{K} &= \lambda k.\ \mathtt{J}\lambda x.x\mathtt{K}\ (\lambda v_a.\ \mathtt{J}0\mathtt{K}\ (\lambda v_b.v_a\ v_b\ k)) \\
&= \lambda k.\ (\lambda k_1.\ k_1\ \left|(\lambda x.\ x)\right|)\ (\lambda v_a.\ (\lambda k_2.\ k_2\ \mathit{j0j})\ (\lambda v_b.v_a\ v_b\ k)) \\
&= \lambda k.\ (\lambda k_1.\ k_1\ (\lambda x.\ \mathtt{J}x\mathtt{K}))\ (\lambda v_a.\ (\lambda k_2.\ k_2\ 0)\ (\lambda v_b.v_a\ v_b\ k)) \\
&= \lambda k.\ (\lambda k_1.\ k_1\ (\lambda x.\ (\lambda k_3.\ k_3\ \mathit{jxj})))\ (\lambda v_a.\ (\lambda k_2.\ k_2\ 0)\ (\lambda v_b.v_a\ v_b\ k)) \\
&= \lambda k.\ (\lambda k_1.\ k_1\ (\lambda x.\ (\lambda k_3.\ k_3\ x)))\ (\lambda v_a.\ (\lambda k_2.\ k_2\ 0)\ (\lambda v_b.v_a\ v_b\ k))
\end{aligned}
$$

To execute this program we apply it to the identity function and then perform β-reductions. We omit the evaluation context because we are merely performing β-reductions and the context therefore only corresponds to a hole.

$$
\begin{aligned}
& (\lambda k.\ (\lambda k_1.\ k_1\ (\lambda x.\ (\lambda k_3.\ k_3\ x)))\ (\lambda v_a.\ (\lambda k_2.\ k_2\ 0)\ (\lambda v_b.\ v_a\ v_b\ k)))(\lambda i.i) \\
\mathit{!}\ & (\lambda k_1.\ k_1\ (\lambda x.\ (\lambda k_3.\ k_3\ x)))\ (\lambda v_a.\ (\lambda k_2.\ k_2\ 0)\ (\lambda v_b.\ v_a\ v_b\ (\lambda i.i))) \\
\mathit{!}\ & (\lambda v_a.\ (\lambda k_2.\ k_2\ 0)\ (\lambda v_b.\ v_a\ v_b\ (\lambda i.i)))(\lambda x.\ (\lambda k_3.\ k_3\ x)) \\
\mathit{!}\ & (\lambda k_2.\ k_2\ 0)\ (\lambda v_b.\ (\lambda x.\ (\lambda k_3.\ k_3\ x))\ v_b\ (\lambda i.i)) \\
\mathit{!}\ & (\lambda v_b.\ (\lambda x.\ (\lambda k_3.\ k_3\ x))\ v_b\ (\lambda i.i))\ 0 \\
\mathit{!}\ & (\lambda x.\ (\lambda k_3.\ k_3\ x))\ 0\ (\lambda i.i) \\
\mathit{!}\ & (\lambda k_3.\ k_3\ 0)\ (\lambda i.i) \\
\mathit{!}\ & (\lambda i.i)\ 0 \\
\mathit{!}\ & 0
\end{aligned}
$$

Reducing the CPS translated term applied to the identity function yields 0 and therefore the same result as if we had reduced the untranslated term $(\lambda x.x)\ 0$.

Having translated the terms of miniFP$^+$ into terms of miniFP using the continuation-passing style translation, we also need to translate the types of miniFP$^+$ to fit the types of the translated terms. Therefore, we will extend the type translation defined by Harper and Lillibridge in [6] [1] to the types of miniFP$^+$.

[1] The translation is itself an extension of the type translation defined by Meyer and Wand in [11].

Definition 4.2.2 (Type translation)**.** We define two mutually recursive functions $\jmath\sigma\jmath$ and $\overline{\sigma}$ that take a type σ of miniFP$^+$ and yield a type for a translated term. The functions are defined as follows.

$$\jmath\sigma\jmath = (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha$$

$$\jmath\forall\beta.\sigma\jmath = \forall\beta.(\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha$$

$$\overline{\sigma} = \sigma \text{ if } \sigma \in \{\mathsf{nat}, \mathsf{bool}, \mathsf{unit}, \mathsf{list}\}$$

$$\overline{\mathsf{Maybe}\ \sigma} \ ! \ \mathsf{Maybe}\ \overline{\sigma}$$

$$\overline{\sigma\ \mathsf{cont}} = \overline{\sigma} \ ! \ \alpha$$

$$\overline{\sigma_1 \ ! \ \sigma_2} = \overline{\sigma_1} \ ! \ \jmath\sigma_2\jmath$$

$$\overline{\forall\beta.\sigma} = \forall\beta.\overline{\sigma}$$

The introduced type α represents the unknown answer type of the continuation introduced by the CPS translation.

Now that we have also defined how to translate the types we can prove that the translated terms are well-typed.

Proposition 4.2.1. *For all values v and terms M in* miniFP$^+$ *the following holds.*

1. *If $\Gamma \vdash v : \sigma$ then $\overline{\Gamma} \vdash \jmath v\jmath : \overline{\sigma}$*
2. *If $\Gamma \vdash M : \sigma$ then $\overline{\Gamma} \vdash \llbracket M \rrbracket : \jmath\sigma\jmath$*

Where $\overline{\Gamma} = x : \overline{\sigma}, \ldots$.

Proof. The proof for both parts is by induction on the derivation of $\Gamma \vdash v : \sigma$ and $\Gamma \vdash M : \sigma$ respectively and a case analysis of the terms.
As an example we will give the proofs for the cases $v = \lambda x.a$ and $M = \mathsf{callcc}\ a$. For the complete proof see Appendix A.6.

Case $v = \lambda x.a$**:** Assuming $\Gamma \vdash \lambda x.a : \sigma \ ! \ \tau$ we have the induction hypothesis IH: $\overline{\Gamma}, x : \overline{\sigma} \vdash \llbracket a \rrbracket : (\overline{\tau} \ ! \ \alpha) \ ! \ \alpha$. Therefore by the abstraction rule we have $\overline{\Gamma} \vdash \lambda x.\llbracket a \rrbracket : \overline{\sigma} \ ! \ ((\overline{\tau} \ ! \ \alpha) \ ! \ \alpha)$ which by the type and value translation is $\overline{\Gamma} \vdash \jmath\lambda x.a\jmath : \overline{\sigma \ ! \ \tau}$.

Case $M = \mathsf{callcc}\ a$**:** Assuming $\Gamma \vdash \mathsf{callcc}\ a : \sigma$ we have the induction hypothesis IH: $\overline{\Gamma} \vdash \llbracket a \rrbracket : \jmath\sigma\ \mathsf{cont} \ ! \ \sigma\jmath = \overline{\Gamma} \vdash \llbracket a \rrbracket : ((\overline{\sigma} \ ! \ \alpha \ ! \ (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha) \ ! \ \alpha) \ ! \ \alpha$. We need to show $\overline{\Gamma} \vdash \llbracket \mathsf{callcc}\ a \rrbracket : (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha$.

$$\dfrac{\dfrac{\overline{\Gamma}_1 \vdash \llbracket a \rrbracket : \gamma \ ! \ \alpha = \text{IH} \qquad \dfrac{\dfrac{\dfrac{f : \delta \in \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash f : \delta}:_{var} \qquad \dfrac{k : \overline{\sigma} \ ! \ \alpha \in \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash k : \overline{\sigma} \ ! \ \alpha}:_{var}}{\overline{\Gamma}_2 \vdash f\ k : (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha}:_{app} \qquad \dfrac{k : \overline{\sigma} \ ! \ \alpha \in \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash k : \overline{\sigma} \ ! \ \alpha}:_{var}}{\overline{\Gamma}_1, f : \delta \vdash f\ k\ k : \alpha}:_{app}}{\overline{\Gamma}_1 \vdash \lambda f.\ f\ k\ k : \gamma}:_{abs}}{\overline{\Gamma}, k : \overline{\sigma} \ ! \ \alpha \vdash \llbracket a \rrbracket (\lambda f.\ f\ k\ k) : \alpha}:_{app}}{\overline{\Gamma} \vdash \lambda k.\ \llbracket a \rrbracket (\lambda f.\ f\ k\ k) : (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha}:_{abs}$$

$\gamma = (\overline{\sigma} \ ! \ \alpha \ ! \ (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha) \ ! \ \alpha$
$\delta = \overline{\sigma} \ ! \ \alpha \ ! \ (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha$
$\overline{\Gamma}_1 = \overline{\Gamma}, k : \overline{\sigma} \ ! \ \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma}_1, f : \delta$

□

4.2.2 Reducing administrative redexes

The content of this section is based on Danvy and Nielsen [3] and Plotkin [12]. From Example 4.2.2 we can see that Plotkin's CPS translation yields very large terms. The terms become so large due to the introduced abstractions at every step of the translation. These abstractions are called administrative redexes. Because of them a reduction in the source term does not directly correspond to a reduction in the translated term. This makes it difficult to prove that a term translated into continuation-passing style yields the same result as the original term.

Plotkin therefore defined a colon translation that reduces some of the administrative redexes. The colon translation translates a term from the original language into a term in continuation-passing style. However, the initial abstraction of the translated term is reduced by applying the continuation. Plotkin defined an infix operator : to write the colon translation of the term a and the continuation K as $a : K$.

Definition 4.2.3 (Plotkin's colon translation)**.** The colon translation $a : K$ translates the term a and its continuation K into continuation-passing style. The function Φ translates values as follows: $\Phi(x) = x$ and $\Phi(\lambda x.a) = \lambda x.\llbracket a \rrbracket$.

$$
\begin{aligned}
v : K &= K \ \Phi(v) \\
v_1 \ v_2 : K &= \Phi(v_1) \ \Phi(v_2) \ K \\
v \ b : K &= b : (\lambda x. \ \Phi(v) \ x \ K) \\
a \ b : K &= a : (\lambda x_1. \ \llbracket b \rrbracket \ (\lambda x_2. \ x_1 \ x_2 \ K))
\end{aligned}
$$

In [12] Plotkin proved the following two lemmas for the colon translation.

Lemma 1. *If K is a closed value then $\llbracket a \rrbracket \ K \ ! \ a : K$.*

Lemma 2. *If $a \ ! \ b$ then $a : K \ ! \ b : K$ (if K is a closed value and a and b are terms).*

Using the two lemmas for the colon translation, a substitution lemma and a lemma regarding stuck terms, Plotkin showed the correctness of the CPS translation by proving the simulation theorem. The theorem states that evaluating a CPS translated program yields the CPS translation of the result of the original program.

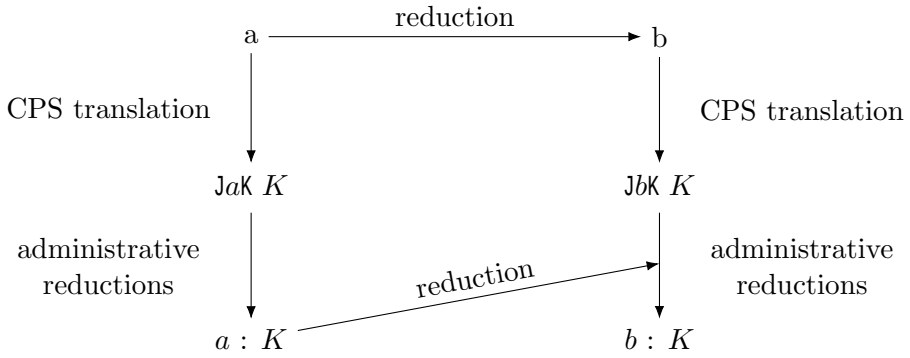


Figure 1: Plotkin's simulation diagram

The simulation theorem can be depicted by the diagram in Figure 1. It shows an indirect correspondence between reductions in the original program and reductions in the translated program. The correspondence stems from the fact that the colon translation reduces the initial administrative redexes. This makes it possible to concentrate on the reduction of the abstractions that were present in the original program. The bottom arrow points to the middle of the arrow from JbK K to $b : K$ to indicate that the reduction of $a : K$ in general results in a term that can be obtained by performing administrative reductions on JbK K.

Example 4.2.3. In this example we will first demonstrate the colon translation and then look at Plotkin's simulation diagram making use of the earlier defined colon translation.

Consider the term $(\lambda x.\ x)\ ((\lambda y.\ y)\ 0)$. We will first translate this term into continuation-passing style using the colon translation.

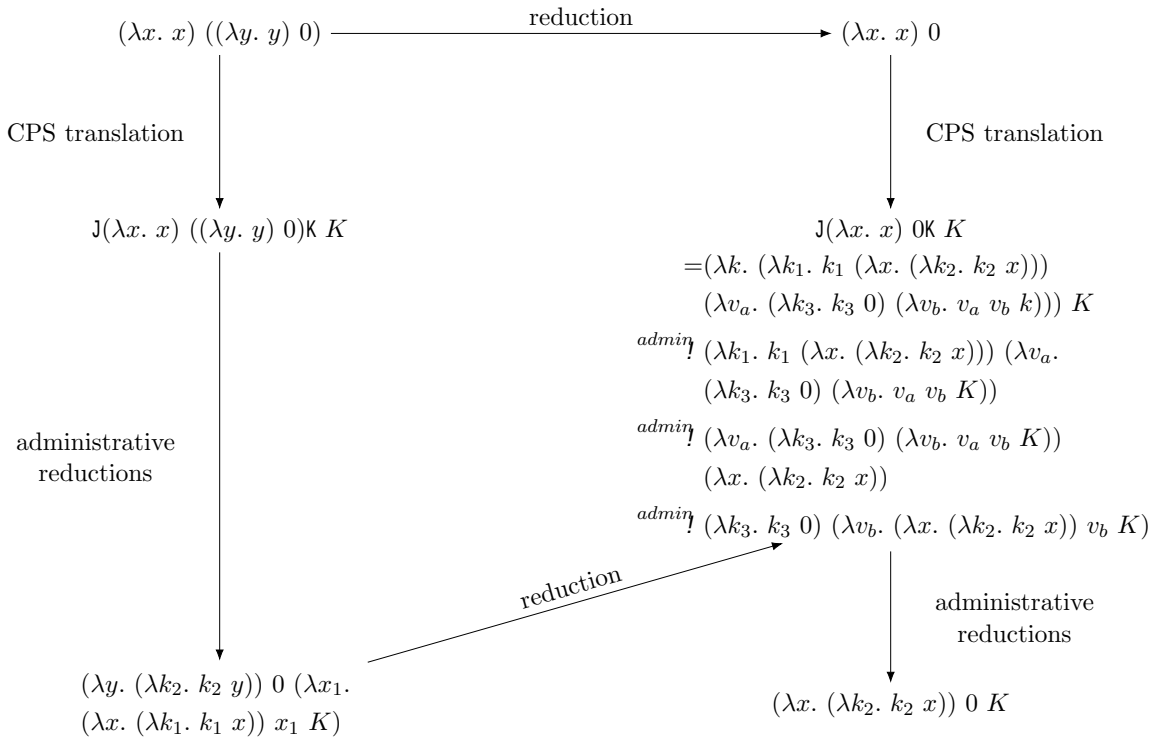
$$\begin{aligned}
&(\lambda x.\ x)\ ((\lambda y.\ y)\ 0) : K \\
=&((\lambda y.\ y)\ 0) : (\lambda x_1.\ \Phi(\lambda x.\ x)\ x_1\ K) \\
=&\Phi(\lambda y.\ y)\ \Phi(0)\ (\lambda x_1.\ (\lambda x.\ \mathtt{J}x\mathtt{K})\ x_1\ K) \\
=&(\lambda y.\ \mathtt{J}y\mathtt{K})\ 0\ (\lambda x_1.\ (\lambda x.\ (\lambda k_1.\ k_1\ x))\ x_1\ K) \\
=&(\lambda y.\ (\lambda k_2.\ k_2\ y))\ 0\ (\lambda x_1.\ (\lambda x.\ (\lambda k_1.\ k_1\ x))\ x_1\ K)
\end{aligned}$$

The term $(\lambda x.\ x)\ ((\lambda y.\ y)\ 0)$ reduces in one step to $(\lambda x.\ x)\ 0$ so to construct Plotkin's simulation diagram we also need the CPS translation and the colon translation of the term $(\lambda x.\ x)\ 0$. These are given by:

$$\begin{aligned}
\mathtt{J}(\lambda x.\ x)\ 0\mathtt{K} &= \lambda k.\ (\lambda k_1.\ k_1\ (\lambda x.\ (\lambda k_2.\ k_2\ x)))\ (\lambda v_a.\ (\lambda k_3.\ k_3\ 0)\ (\lambda v_b.v_a\ v_b\ k)) \\
(\lambda x.\ x)\ 0 : K &= (\lambda x.\ (\lambda k_2.\ k_2\ x))\ 0\ K
\end{aligned}$$

We will omit the CPS translation of $(\lambda x.\ x)\ ((\lambda y.\ y)\ 0)$ because it is evident that there is no correspondence between a reduction of the translated term and the reduction in the original term due to the many administrative redexes.

We will now construct Plotkin's simulation diagram to relate the reduction steps.



As we can see the reduction of the colon translation of $(\lambda x.\ x)\ ((\lambda y.\ y)\ 0)$ reduces the λy-abstraction. This reduction corresponds to the reduction happening in the original term. The reduction of the colon translation does not yield the colon translation of $(\lambda x.\ x)\ 0$. Instead it yields a term that can be obtained by performing administrative reductions on $\mathtt{J}(\lambda x.\ x)\ 0\mathtt{K}\ K$. This term again needs to be further reduced to obtain the colon translation of $(\lambda x.\ x)\ 0$.

We can see from Example 4.2.3 that reasoning about the translated terms is difficult. It is only possible via the colon translation. Even so the reductions corresponding to original reductions are mixed with administrative reductions. Using Plotkin's original colon translation we will give an extension of it to the terms of miniFP^+.

Definition 4.2.4 (Extended colon translation)**.** Instead of Plotkin's function Φ we will make use of the value translation function $\jmath\ \jmath$ defined in Definition 4.2.1.

$$
\begin{aligned}
v : K &= K\ \jmath v \jmath \\
x : K &= K\ x \\
v_1\ v_2 : K &= \jmath v_1 \jmath\ \jmath v_2 \jmath\ K \\
v\ b : K &= b : (\lambda x.\ \jmath v \jmath\ x\ K) \\
a\ b : K &= a : (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \\
\mathsf{Some}\ a : K &= a : (\lambda x.\ K\ \mathsf{Some}\ x) \\
\mathsf{zero?}\ a : K &= a : (\lambda x.\ K\ \mathsf{zero?}\ x) \\
\mathsf{Succ}(a) : K &= a : (\lambda x.\ K\ \mathsf{Succ}(x)) \\
\mathsf{Cons}(a, b) : K &= a : (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ K\ \mathsf{Cons}(x_1, x_2))) \\
a + b : K &= a : (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ K\ (x_1 + x_2)))
\end{aligned}
$$

$$
\begin{aligned}
a\ ;\ b : K &= a : (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ K\ x_2)) \\
\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b : K &= c : (\lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathtt{J}a\mathtt{K}\ K\ \mathsf{else}\ \mathtt{J}b\mathtt{K}\ K) \\
\mathsf{let}\ x = v\ \mathsf{in}\ a : K &= \mathsf{let}\ x = \mathit{jvj}\ \mathsf{in}\ \mathtt{J}a\mathtt{K}\ K \\
\mathsf{match}\ l\ \mathsf{with}\ \mathsf{Nil}\ \mathit{!}\ e\ \mathit{j}\ \mathsf{Cons}(a,b)\ \mathit{!}\ f : K &= l : (\lambda x.\mathsf{match}\ x\ \mathsf{with}\ \mathsf{Nil}\ \mathit{!}\ \mathtt{J}e\mathtt{K}\ K \\
&\qquad \mathit{j}\ \mathsf{Cons}(a,b)\ \mathit{!}\ \mathtt{J}f\mathtt{K}\ K) \\
\mathsf{callcc}\ a : K &= a : (\lambda f.\ f\ K\ K) \\
\mathsf{throw}\ a\ b : K &= a : (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ x_1\ x_2))
\end{aligned}
$$

The colon translation eliminates callcc- and throw-terms. The translated terms are therefore also terms of miniFP.

For the extended colon translation we show that Plotkin's Lemma 1 holds, i.e. we prove that reducing administrative redexes in the CPS translated term yields the colon translation of the same term.

Lemma 3 (Extended colon translation)**.** *If K is a closed value then* $\mathtt{J}a\mathtt{K}\ K\ \mathit{!}\ a : K$.

Proof. The proof is by induction on a and a case analysis of a. As the proof is straightforward we will give as an example the proof of the case that $a = a_1\ a_2$. For the complete proof see Appendix A.7.

Case $a = a_1\ a_2$:

$$
\begin{aligned}
\mathtt{J}a_1\ a_2\mathtt{K}\ K &= (\lambda k.\ \mathtt{J}a_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ k)))\ K \\
&\quad \mathit{!}\ \mathtt{J}a_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \\
&\quad \mathit{!}\ a_1 : (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \quad \text{by induction hypothesis of } a_1 \\
&= a_1\ a_2 : K
\end{aligned}
$$

□

Furthermore, the terms produced by the colon translation are also well-typed. From Proposition 4.2.1 we know that a CPS translated term $\mathtt{J}a\mathtt{K}$ is well-typed and in general has type $(\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$. We also know that a continuation K has the type $\overline{\sigma}\ \mathit{!}\ \alpha$. Therefore the application $\mathtt{J}a\mathtt{K}\ K$ is well-typed. From Lemma 3 it follows that a CPS translated term $\mathtt{J}a\mathtt{K}$ applied to a continuation K reduces to the term produced by the colon translation. It therefore holds that the terms produced by the colon translation are also well-typed.

Using the extended colon translation we will translate the earlier defined *find_one* program into continuation-passing style. If we compare the evaluation of the translated program in Appendix A.8 and the evaluation of the untranslated program in Appendix A.4 we see that the first reduction of the translated program corresponds to the first reduction in the original program. In both cases the let-expression is reduced and the list_iter program is filled in for the variable *list_iter* in find. Due to the administrative redexes that the colon translation did not remove, the following reductions do not correspond. We can also see that both the untranslated and the translated find program evaluate to Some Succ(0). The *find_one* example shows that it is possible to use a translation into continuation-passing style to eliminate the operators callcc and throw in a program but at the same time preserve their semantics.

4.3 Discussion

During the evaluation of the CPS translated *find_one* program we notice that the redex zero? 0 is positioned at the end of the term.

$$
\begin{array}{lll}
 & E[(\lambda v_3.\ (\lambda v_{20}.\ \textsf{if}\ v_{20}\ \textsf{then...else...})\ \textsf{zero?}\ v_3)\ 0] & \text{with } E = [\,] \\
! & E[(\lambda v_{20}.\ \textsf{if}\ v_{20}\ \textsf{then...else...})\ \textsf{zero?}\ 0] & \text{with } E = [\,] \\
= & E_2[\textsf{zero?}\ 0] & \text{with } E_2 = (\lambda v_{20}.\ \textsf{if}\ v_{20}\ \textsf{then...else...})\ [\,] \\
! & E_2[\textsf{true}] & \text{with } E_2 = (\lambda v_{20}.\ \textsf{if}\ v_{20}\ \textsf{then...else...})\ [\,] \\
= & E[(\lambda v_{20}.\ \textsf{if}\ v_{20}\ \textsf{then...else...})\ \textsf{true}] & \text{with } E = [\,]
\end{array}
$$

As miniFP uses the call-by-value evaluation contexts from Definition 3.2.1, zero? 0 is reduced to true first, before it gets passed to the if-statement.

However, Plotkin's indifference theorem [12] states that $\texttt{J}a\texttt{K}\ (\lambda i.i)$ evaluates in the same manner in call-by-value and call-by-name. Since the semantics for miniFP are fixed to evaluate call-by-value, zero? 0 is evaluated first. But if we change the semantics to evaluate call-by-name, zero? 0 will be substituted into the if-statement immediately and reduced later. So the indifference theorem does not hold for our translation, as the evaluation for both reduction strategies is not the same.

The problem can be traced to the CPS and colon translation of zero? a.

$$
\begin{aligned}
\texttt{J}\textsf{zero?}\ a\texttt{K} &= \lambda k.\ \texttt{J}a\texttt{K}\ (\lambda x.\ k\ \textsf{zero?}\ x) \\
\textsf{zero?}\ a : K &= a : (\lambda x.\ K\ \textsf{zero?}\ x)
\end{aligned}
$$

In both translations the term zero? x gets passed to the continuation. It therefore always ends up at the end of the term and there will be two possible reductions, either to reduce zero? x first or to substitute it immediately.

To solve this problem the translations of zero? a would have to be slightly changed.

$$
\begin{aligned}
\texttt{J}\textsf{zero?}\ a\texttt{K} &= \lambda k.\ \texttt{J}a\texttt{K}\ (\lambda x.\ \textsf{zero?}\ x\ k) \\
\textsf{zero?}\ a : K &= a : (\lambda x.\ \textsf{zero?}\ x\ K)
\end{aligned}
$$

The only difference between the translations is that now the continuation is positioned at the end of the term. However, this also means that zero? x must reduce to a function expecting a continuation. So the semantics of miniFP regarding zero? x would also have to be changed to the following:

$$
\begin{aligned}
&\textsf{zero?}\ 0 \ \ ! \ \ \lambda k.\ k\ \textsf{true} \\
&\textsf{zero?}\ v \ \ ! \ \ \lambda k.\ k\ \textsf{false} \quad \text{if } v \neq 0
\end{aligned}
$$

zero? x now reduces to a function that expects a continuation. Either true or false will then be passed to this continuation.

If we would translate the *find_one* program using the new translation the term from the

beginning and the following reductions will be different.

	$E[(\lambda v_3.\ \mathsf{zero?}\ v_3\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then...else...}))\ 0]$	with $E = [\,]$
$!$	$E[\mathsf{zero?}\ 0\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then...else...})]$	with $E = [\,]$
$=$	$E_2[\mathsf{zero?}\ 0]$	with $E_2 = [\,]\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then...else...})$
$!$	$E_2[\lambda k.\ k\ \mathsf{true}]$	with $E_2 = [\,]\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then...else...})$
$=$	$E[(\lambda k.\ k\ \mathsf{true})\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then...else...})]$	with $E = [\,]$
$!$	$E[(\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then...else...})\ \mathsf{true}]$	with $E = [\,]$

Because the continuation of $\mathsf{zero?}\ x$ is at the end of the term, after we substitute 0 for v_3, the only redex is $\mathsf{zero?}\ 0$. Which reduces by the new reduction rule to $\lambda k.\ k\ \mathsf{true}$. The "if-continuation" then gets substituted for the k. We obtain the same term as in the beginning only now the term evaluates the same way in call-by-value and in call-by-name.

Chapter 5

Related Work

In [14] Reynolds gives a detailed description of the history of continuations. The concept of continuations was discovered multiple times by different individuals. Reynolds believes this was due to the many settings in which continuations appear.

Without explicitly mentioning them, continuations had been present in the implementation of Algol 60 making it possible to jump out of blocks. Furthermore, in retrospect also Landin's SECD machine [7] made use of continuations. Landin had shown a correspondence between some of the semantics for Algol 60 and the λ-calculus [8]. For his newly defined language he then gave a call-by-value interpreter which was the SECD machine. The machine consists of a *stack, environment, control* and *dump* component. The dump stores the remaining computation to be done and can therefore be seen as a continuation. When Landin showed the correspondence between Algol 60 and λ-calculus he introduced the J-operator to model the jumps possible in Algol 60 [8]. The J-operator was able to capture its continuation and can therefore be seen as a predecessor of the *call-with-current-continuation* operator of the Scheme language.

The first actual mention of continuations was 1964 by van Wijngaarden during a conference. He also described a translation into continuation-passing style. However, continuations and continuation-passing style did not become widely known until the 1970's when Plotkin [12], Reynolds [13] and Fischer [5] defined them.

Today many existing CPS translations are based on Plotkin's call-by-value translation [12]. However, Plotkin's CPS translation introduces many administrative redexes that make it difficult to prove properties of the translation. Plotkin therefore defined a colon translation that reduces the administrative redexes. Using this new translation he established a relation between reductions on source terms and terms translated to continuation-passing style.

Many different CPS translations have been defined that try to minimize the number of administrative redexes produced by the translation. In [3] Danvy and Nielsen defined a CPS translation as well as a modified colon translation which removes more administrative redexes than Plotkin's original colon translation. Using their CPS translation they give a direct proof for Plotkin's simulation theorem, relating a reduction in the source term to one or more reductions in the translated term.

Chapter 6

Conclusions

In this thesis we have looked at continuations and their uses in programs. We have defined the language miniFP$^+$ that contains an operator callcc and throw. We described the semantics of miniFP$^+$ using evaluation contexts and typed the terms.

miniFP$^+$ was used to write programs that make use of continuations and callcc. The evaluation steps of these programs were described in detail to highlight the usage and functionality of callcc. The program *find_one* showed that exceptions can be achieved by saving the current context with callcc. If later in the program the specified exception occurs, throw restores the context saved earlier by callcc. The current context will be thrown away. The result value of the exception will become the result of the callcc. The evaluation of the *print_all* program showed that backtracking can be achieved by using two callcc functions. One captured the printing context and the other one captured the current position in the list. The alternating invocation of the two captured continuations made it possible to print an element and then resume iteration over the list.

In Chapter 4 we introduced continuations and gave a translation into continuation-passing style for the terms of miniFP$^+$. The translation is based on Plotkin's CPS translation. It was proved that the translated terms are well-typed. We saw that the terms generated by Plotkin's CPS translation contain many administrative redexes that make it difficult to relate a reduction in the original term to a reduction in the translated term. Using the colon translation, however, Plotkin showed that it is possible to proof the correctness of the translation. We have extended Plotkin's colon translation to the terms of miniFP$^+$ and used it to translate the *find_one* program into continuation-passing style. Comparing the evaluation of the translated and untranslated program we could see a correspondence in one reduction due to the colon translation. Furthermore, we saw that the translated program evaluates to the same as the untranslated program. However, during the evaluation of the translated *find_one* program we noticed that Plotkin's indifference theorem does not hold for our translations. To solve this problem the translation of the term zero? a as well as the semantics of miniFP regarding zero? a would have to be changed.

Bibliography

[1] Henk Barendregt. Lambda calculi with types. In *Handbook of logic in computer science*, pages 117–309. Oxford University Press, 1992.

[2] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[3] Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. In *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2002.

[4] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[5] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3-4):259–288, 1993.

[6] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *Lisp and Symbolic Computation*, 6(3-4):361–380, 1993.

[7] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[8] Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 1965.

[9] Xavier Leroy. MPRI 2-4 Functional Programming languages Part I: interpreters and operational semantics [pdf slideshow], 2016-2017. Retrieved from `https://xavierleroy.org/mpri/2-4/semantics.pdf`.

[10] Xavier Leroy. MPRI 2-4 Functional Programming languages Part III: program transformations [pdf slideshow], 2016-2017. Retrieved from `https://xavierleroy.org/mpri/2-4/transformations.pdf`.

[11] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer, 1985.

[12] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[13] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Proceedings of 25th ACM National Conference*, 2:717–740, 1972.

[14] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

Appendix A

Appendix

A.1 Example Harper and Lillibridge

In this section we will discuss the example Harper and Lillibridge present in [6]. In the example the term e_0 demonstrates why the let-bound terms have to be restricted to values.

$$e_0 = \mathsf{let}\ f = \mathsf{callcc}\ (\lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x))\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true})$$

If we type e_0 we obtain the type bool. The function f has the type $\forall\alpha.\alpha\ \ !\ \ \alpha$ and can subsequently be used as a function of type nat $!$ nat and bool $!$ bool. However, evaluating e_0 results in 0. Because the term that f is assigned to is not a value, we start by evaluating the callcc. This way the current continuation is saved. Later $f\ 0$ is evaluated which invokes the continuation passing $(\lambda y.\ 0)$ as argument. Thus f becomes $(\lambda y.\ 0)$. Then $(\lambda y.\ 0)$ true reduces to 0.

To prevent this the let-bound terms are restricted to values.

Type of e_0:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{k : (\mathsf{bool}\ \ !\ \ \mathsf{bool})\ \mathsf{cont}\ \mathit{2}\ \Gamma_2}{\Gamma_2 \vdash k : (\mathsf{bool}\ \ !\ \ \mathsf{bool})\ \mathsf{cont}}:_{var} \quad \dfrac{\dfrac{x : \mathsf{bool}\ \mathit{2}\ \Gamma_3}{\Gamma_2, y : \mathsf{bool} \vdash x : \mathsf{bool}}:_{var}}{\Gamma_2 \vdash \lambda y.\ x : \mathsf{bool}\ \ !\ \ \mathsf{bool}}:_{abs}}{\Gamma_1, x : \mathsf{bool} \vdash \mathsf{throw}\ k\ (\lambda y.\ x) : \mathsf{bool}}:_{throw}}{\Gamma, k : (\mathsf{bool}\ \ !\ \ \mathsf{bool})\ \mathsf{cont} \vdash \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x) : \mathsf{bool}\ \ !\ \ \mathsf{bool}}:_{abs}}{\Gamma \vdash \lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x) : (\mathsf{bool}\ \ !\ \ \mathsf{bool})\ \mathsf{cont}\ \ !\ \ \mathsf{bool}\ \ !\ \ \mathsf{bool}}:_{callcc}}{\Gamma \vdash \mathsf{callcc}\ (\lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x)) : \mathsf{bool}\ \ !\ \ \mathsf{bool}}$$

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{callcc}\ (\lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x)) : \mathsf{bool}\ \ !\ \ \mathsf{bool}}{\Gamma \vdash \mathsf{callcc}\ (\lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x)) : \forall\alpha.\alpha\ \ !\ \ \alpha}:_{intro},\ \mathsf{bool}\ \mathit{@}\ \Gamma \qquad \textcircled{1}}{\Gamma, f : \forall\alpha.\alpha\ \ !\ \ \alpha \vdash (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true}) : \mathsf{bool}}:_{app}}{\Gamma \vdash \mathsf{let}\ f = \mathsf{callcc}\ (\lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x))\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true}) : \mathsf{bool}}:_{let\_poly}$$

$$\dfrac{\dfrac{f : \forall\alpha.\alpha\ \ !\ \ \alpha\ \mathit{2}\ \Gamma_4}{\Gamma_4 \vdash f : \forall\alpha.\alpha\ \ !\ \ \alpha}:_{var}}{\Gamma_4 \vdash f : \mathsf{bool}\ \ !\ \ \mathsf{bool}}:_{elim} \quad \dfrac{}{\Gamma_4 \vdash \mathsf{true} : \mathsf{bool}}:_{true}}{\Gamma_4 \vdash f\ \mathsf{true} : \mathsf{bool}}:_{app}$$

$\textcircled{1}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{y : \mathsf{bool}\ \mathit{2}\ \Gamma_5, y : \mathsf{bool}}{\Gamma_5, y : \mathsf{bool} \vdash y : \mathsf{bool}}:_{var}}{\Gamma_4, x : \mathsf{nat} \vdash \lambda y.\ y : \mathsf{bool}\ \ !\ \ \mathsf{bool}}:_{abs}}{\Gamma_4 \vdash \lambda x.\ \lambda y.\ y : \mathsf{nat}\ \ !\ \ \mathsf{bool}\ \ !\ \ \mathsf{bool}}:_{abs} \quad \dfrac{\dfrac{\dfrac{f : \forall\alpha.\alpha\ \ !\ \ \alpha\ \mathit{2}\ \Gamma_4}{\Gamma_4 \vdash f : \forall\alpha.\alpha\ \ !\ \ \alpha}:_{var}}{\Gamma_4 \vdash f : \mathsf{nat}\ \ !\ \ \mathsf{nat}}:_{elim} \quad \dfrac{}{\Gamma_4 \vdash 0 : \mathsf{nat}}:_{nat}}{\Gamma_4 \vdash f\ 0 : \mathsf{nat}}:_{app}}{\Gamma_4 \vdash (\lambda x.\ \lambda y.\ y)\ (f\ 0) : \mathsf{bool}\ \ !\ \ \mathsf{bool}}:_{app}$$

$\Gamma_1 = \Gamma, k : (\mathsf{bool}\ \ !\ \ \mathsf{bool})\ \mathsf{cont}$
$\Gamma_2 = \Gamma_1, x : \mathsf{bool}$
$\Gamma_3 = \Gamma_2, y : \mathsf{bool}$
$\Gamma_4 = \Gamma, f : \forall\alpha.\alpha\ \ !\ \ \alpha$
$\Gamma_5 = \Gamma_4, x : \mathsf{nat}$

Evaluation of e_0:

For this example we have to extend our contexts because the term assigned to f is not a value. The reduction rules remain unchanged.

$$E ::= \ldots \mid \mathsf{let}\ x = E\ \mathsf{in}\ a$$

$E[\mathsf{let}\ f = \mathsf{callcc}\ (\lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x))\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true})]$	with $E = [\,]$
$= E_1[\mathsf{callcc}\ (\lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x))]$	with $E_1 = \mathsf{let}\ f = [\,]\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true})$
$!\ E_1[(\lambda k.\ \lambda x.\ \mathsf{throw}\ k\ (\lambda y.\ x))\ (\lambda x.\ E_1[x])]$	with $E_1 = \mathsf{let}\ f = [\,]\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true})$
$!\ E_1[\lambda x.\ \mathsf{throw}\ (\lambda x.\ E_1[x])\ (\lambda y.\ x)]$	with $E_1 = \mathsf{let}\ f = [\,]\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true})$
$= E_2[\mathsf{let}\ f = \lambda x.\ \mathsf{throw}\ (\lambda x.\ E_1[x])\ (\lambda y.\ x)\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true})]$	with $E_2 = [\,]$
$!\ E_2[(\lambda x.\ \lambda y.\ y)\ ((\lambda x.\ \mathsf{throw}\ (\lambda x.\ E_1[x])\ (\lambda y.\ x))\ 0)\ ((\lambda x.\ \mathsf{throw}\ (\lambda x.\ E_1[x])\ (\lambda y.\ x))\ \mathsf{true})]$	with $E_2 = [\,]$
$= E_3[(\lambda x.\ \mathsf{throw}\ (\lambda x.\ E_1[x])\ (\lambda y.\ x))\ 0]$	with $E_3 = (\lambda x.\ \lambda y.\ y)\ [\,]\ ((\lambda x.\ \mathsf{throw}\ (\lambda x.\ E_1[x])\ (\lambda y.\ x))\ \mathsf{true})$
$!\ E_3[\mathsf{throw}\ (\lambda x.\ E_1[x])\ (\lambda y.\ 0)]$	with $E_3 = (\lambda x.\ \lambda y.\ y)\ [\,]\ ((\lambda x.\ \mathsf{throw}\ (\lambda x.\ E_1[x])\ (\lambda y.\ x))\ \mathsf{true})$
$!\ (\lambda x.\ E_1[x])\ (\lambda y.\ 0)$	
$!\ E_1[(\lambda y.\ 0)]$	with $E_1 = \mathsf{let}\ f = [\,]\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true})$
$= E_4[\mathsf{let}\ f = (\lambda y.\ 0)\ \mathsf{in}\ (\lambda x.\ \lambda y.\ y)\ (f\ 0)\ (f\ \mathsf{true})]$	with $E_4 = [\,]$
$!\ E_4[(\lambda x.\ \lambda y.\ y)\ ((\lambda y.\ 0)\ 0)\ ((\lambda y.\ 0)\ \mathsf{true})]$	with $E_4 = [\,]$
$!\ E_4[(\lambda y.\ 0)\ \mathsf{true}]$	with $E_4 = [\,]$
$!\ E_4[0]$	with $E_4 = [\,]$

A.2 Proof of Proposition 3.3.1

Proposition 3.3.1. *For all terms t 2 miniFP$^+$, if ` $t : \sigma$ and t @ Values then there exists a unique E with either*

1. $t = E[e]$ *and* e *reduces by head reduction*
2. $t = E[\mathit{callcc}\ u]$ *and* $E[\mathit{callcc}\ u]$ *reduces by context reduction*
3. $t = E[\mathit{throw}\ u\ v]$ *and* $E[\mathit{throw}\ u\ v]$ *reduces by context reduction.*

Proof. The proof is by induction on t and a case analysis of t.

Case $\mathsf{Cons}(a, b)$:

a @ **Values** : From induction there exists a unique E^0 with either 1. $a = E^0[e^0]$ and e^0 reduces by head reduction or 2. $a = E^0[\mathsf{callcc}\ c]$ and $E^0[\mathsf{callcc}\ c]$ reduces by context reduction or 3. $a = E^0[\mathsf{throw}\ c\ d]$ and $E^0[\mathsf{throw}\ c\ d]$ reduces by context reduction.

1. Then there exists a unique E and e such that $\mathsf{Cons}(a, b) = E[e]$ and e reduces by head reduction. Because $E = \mathsf{Cons}(E^0, b)$ and $e = e^0$.
2. Then there exists a unique E such that $\mathsf{Cons}(a, b) = E[\mathsf{callcc}\ u]$ and $E[\mathsf{callcc}\ u]$ reduces by context reduction. Because $E = \mathsf{Cons}(E^0, b)$ and $u = c$, we have $E[\mathsf{callcc}\ c] = \mathsf{Cons}(E^0[\mathsf{callcc}\ c], b)$ which reduces by context reduction to $\mathsf{Cons}(E^0[c\ (\lambda x.E[x])], b)$.
3. Then there exists a unique E such that $\mathsf{Cons}(a, b) = E[\mathsf{throw}\ u\ v]$ and $E[\mathsf{throw}\ u\ v]$ reduces by context reduction. Because $E = \mathsf{Cons}(E^0, b)$, $u = c$ and $v = d$, we have $E[\mathsf{throw}\ c\ d] = \mathsf{Cons}(E^0[\mathsf{throw}\ c\ d], b)$ which reduces by context reduction to $c\ d$.

a 2 **Values** : The proof is the same as for the case a @ Values only now the induction is over b.

Case $\mathsf{zero?}\ a$:

a 2 **Values** : Take $E = [\,]$ and $e = \mathsf{zero?}\ a$ and $\mathsf{zero?}\ a$ reduces by head reduction. Now we show that E is unique. Suppose there is another context $\tilde{E}$ and term $\tilde{e}$ for which $\mathsf{zero?}\ a = \tilde{E}[\tilde{e}]$. $\tilde{E}$ can only have one other possible form:

1. $\tilde{E} = \mathsf{zero?}\ [\,]$
   Then $\tilde{e} = a$ but a does not reduce. So $\tilde{E}$ can not be of the form and E.

So E is unique.

a @ **Values** : From induction there exists a unique E^0 with either 1. $a = E^0[e^0]$ and e^0 reduces by head reduction or 2. $a = E^0[\mathsf{callcc}\ c]$ and $E^0[\mathsf{callcc}\ c]$ reduces by context reduction or 3. $a = E^0[\mathsf{throw}\ c\ d]$ and $E^0[\mathsf{throw}\ c\ d]$ reduces by context reduction.

1. Then there exists a unique E and e such that $\mathsf{zero?}\ a = E[e]$ and e reduces by head reduction. Because $E = \mathsf{zero?}\ E^0$ and $e = e^0$.
2. Then there exists a unique E such that $\mathsf{zero?}\ a = E[\mathsf{callcc}\ u]$ and $E[\mathsf{callcc}\ u]$ reduces by context reduction. Because $E = \mathsf{zero?}\ E^0$ and $u = c$, we have $E[\mathsf{callcc}\ c] = \mathsf{zero?}\ E^0[\mathsf{callcc}\ c]$ which reduces by context reduction to $\mathsf{zero?}\ E^0[c\ (\lambda x.E[x])]$.

3. Then there exists a unique E such that $\mathsf{zero?}\ a = E[\mathsf{throw}\ u\ v]$ and $E[\mathsf{throw}\ u\ v]$ reduces by context reduction. Because $E = \mathsf{zero?}\ E^0$, $u = c$ and $v = d$, we have $E[\mathsf{throw}\ c\ d] = \mathsf{zero?}\ E^0[\mathsf{throw}\ c\ d]$ which reduces by context reduction to $c\ d$.

Case $\mathsf{Succ}(a)$: From induction there exists a unique E^0 with either 1. $a = E^0[e^0]$ and e^0 reduces by head reduction or 2. $a = E^0[\mathsf{callcc}\ c]$ and $E^0[\mathsf{callcc}\ c]$ reduces by context reduction or 3. $a = E^0[\mathsf{throw}\ c\ d]$ and $E^0[\mathsf{throw}\ c\ d]$ reduces by context reduction.

1. Then there exists a unique E and e such that $\mathsf{Succ}(a) = E[e]$ and e reduces by head reduction. Because $E = \mathsf{Succ}(E^0)$ and $e = e^0$.
2. Then there exists a unique E such that $\mathsf{Succ}(a) = E[\mathsf{callcc}\ u]$ and $E[\mathsf{callcc}\ u]$ reduces by context reduction. Because $E = \mathsf{Succ}(E^0)$ and $u = c$, we have $E[\mathsf{callcc}\ c] = \mathsf{Succ}(E^0[\mathsf{callcc}\ c])$ which reduces by context reduction to $\mathsf{Succ}(E^0[c\ (\lambda x.E[x])])$.
3. Then there exists a unique E such that $\mathsf{Succ}(a) = E[\mathsf{throw}\ u\ v]$ and $E[\mathsf{throw}\ u\ v]$ reduces by context reduction. Because $E = \mathsf{Succ}(E^0)$, $u = c$ and $v = d$, we have $E[\mathsf{throw}\ c\ d] = \mathsf{Succ}(E^0[\mathsf{throw}\ c\ d])$ which reduces by context reduction to $c\ d$.

The proof for the case $\mathsf{Some}\ a$ is similar.

Case a **;** b : There exists a unique E with $a\ ;\ b = E[e]$ and e reduces by head reduction. Because $E = [\,]$ and $e = a\ ;\ b$.

Case $\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b$:

c **2 Values** : Take $E = [\,]$ and $e = \mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b$ which reduces by head reduction because c is of type bool so either true or false. Now we show that E is unique. Suppose there is another context $\tilde{E}$ and term $\tilde{e}$ for which $\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b = \tilde{E}[\tilde{e}]$. $\tilde{E}$ can only have one other possible form:

1. $\tilde{E} = \mathsf{if}\ [\,]\ \mathsf{then}\ a\ \mathsf{else}\ b$
   Then $\tilde{e} = c$ but c does not reduce. So $\tilde{E}$ cannot be of the form and E.

So E is unique.

c **2 Values** : From induction there exists a unique E^0 with either 1. $c = E^0[e^0]$ and e^0 reduces by head reduction or 2. $c = E^0[\mathsf{callcc}\ d]$ and $E^0[\mathsf{callcc}\ d]$ reduces by context reduction or 3. $a = E^0[\mathsf{throw}\ d\ e]$ and $E^0[\mathsf{throw}\ d\ e]$ reduces by context reduction.

1. Then there exists a unique E and e such that $\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b = E[e]$ and e reduces by head reduction. Because $E = \mathsf{if}\ E^0\ \mathsf{then}\ a\ \mathsf{else}\ b$ and $e = e^0$.
2. Then there exists a unique E such that $\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b = E[\mathsf{callcc}\ u]$ and $E[\mathsf{callcc}\ u]$ reduces by context reduction. Because $E = \mathsf{if}\ E^0\ \mathsf{then}\ a\ \mathsf{else}\ b$ and $u = d$, we have $E[\mathsf{callcc}\ d] = \mathsf{if}\ E^0[\mathsf{callcc}\ d]\ \mathsf{then}\ a\ \mathsf{else}\ b$ which reduces by context reduction to $\mathsf{if}\ E^0[d\ (\lambda x.E[x])]\ \mathsf{then}\ a\ \mathsf{else}\ b$.
3. Then there exists a unique E such that $\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b = E[\mathsf{throw}\ u\ v]$ and $E[\mathsf{throw}\ u\ v]$ reduces by context reduction. Because $E = \mathsf{if}\ E^0\ \mathsf{then}\ a\ \mathsf{else}\ b$, $u = d$ and $v = e$, we have $E[\mathsf{throw}\ d\ e] = \mathsf{if}\ E^0[\mathsf{throw}\ c\ d]\ \mathsf{then}\ a\ \mathsf{else}\ b$ which reduces by context reduction to $d\ e$.

Case $a\ b$:

a **2 Values** : From induction there exists a unique E^0 with either 1. $a = E^0[e^0]$ and e^0 reduces by head reduction or 2. $a = E^0[\mathsf{callcc}\ c]$ and $E^0[\mathsf{callcc}\ c]$ reduces by context reduction or 3. $a = E^0[\mathsf{throw}\ c\ d]$ and $E^0[\mathsf{throw}\ c\ d]$ reduces by context reduction.

1. Then there exists a unique E and e such that $a\ b = E[e]$ and e reduces by head reduction. Because $E = E^0\ b$ and $e = e^0$.
2. Then there exists a unique E such that $a\ b = E[\mathsf{callcc}\ u]$ and $E[\mathsf{callcc}\ u]$ reduces by context reduction. Because $E = E^0\ b$ and $u = c$, we have $E[\mathsf{callcc}\ c] = E^0[\mathsf{callcc}\ c]\ b$ which reduces by context reduction to $E^0[c\ (\lambda x.E[x])]\ b$.
3. Then there exists a unique E such that $a\ b = E[\mathsf{throw}\ u\ v]$ and $E[\mathsf{throw}\ u\ v]$ reduces by context reduction. Because $E = E^0\ b$,$u = c$ and $v = d$, we have $E[\mathsf{throw}\ c\ d] = E^0[\mathsf{throw}\ c\ d]\ b$ which reduces by context reduction to $c\ d$.

a *2* **Values and** b **2 Values** : The proof is the same as for the case a **2** Values only now the induction is over b.

a, b *2* **Values** : Take $E = [\ ]$ and $e = a\ b$ which reduces by head reduction because a is a function type and b of the corresponding argument type. Now we show that E is unique. Suppose there is another context $\tilde{E}$ and term $\tilde{e}$ for which $x = \tilde{E}[\tilde{e}]$. $\tilde{E}$ can have two possible forms:

1. $\tilde{E} = a\ [\ ]$
   Then $\tilde{e} = b$ but b does not reduce. So $\tilde{E}$ cannot be of the form.
2. $\tilde{E} = [\ ]\ b$
   Then $\tilde{e} = a$ but a does not reduce. So $\tilde{E}$ cannot be of the form.

So E is unique.

Case $a + b$: The proof is similar to the proof for the case $a\ b$.

Case $\mathsf{match}\ list\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ a\ \mathsf{j}\ \mathsf{Cons}(hd, tl)\ \ !\ \ b$: The proof is similar to the proof of the case $\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b$.

Case $\mathsf{let}\ x = v\ \mathsf{in}\ b$: Take $E = [\ ]$ and $e = \mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b$ which reduces by head reduction.

Case $\mathsf{callcc}\ a$: There exists a unique E with $\mathsf{callcc}\ a = E[\mathsf{callcc}\ u]$. Because $E = [\ ]$ and $u = a$. Then $E[\mathsf{callcc}\ a]$ reduces by context reduction.

Case $\mathsf{throw}\ a\ b$: There exists a unique E with $\mathsf{throw}\ a\ b = E[\mathsf{throw}\ u\ v]$. Because $E = [\ ]$, $u = a$ and $v = b$. Then $E[\mathsf{throw}\ a\ b]$ reduces by context reduction.

□

A.3 Type derivation of **list_iter** and **find**

In this example we will first give a type derivation for list_iter. Then assuming the type for list_iter we will give the type for the find program. The two programs and their types are defined as follows.

$$
\begin{array}{l}
\mathsf{list\_iter} : (\mathsf{nat}\ \ !\ \ \mathsf{unit})\ \ !\ \ \mathsf{list}\ \ !\ \ \mathsf{unit} \\
\mathsf{list\_iter} := \mu list\_iter.\lambda f.\lambda list.\ \mathsf{match}\ list\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ () \\
\qquad \mid \mathsf{Cons}(\mathrm{hd},\mathrm{tl})\ \ !\ \ f\ \mathrm{hd}\ ; list\_iter\ f\ \mathrm{tl}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{find} : (\mathsf{nat}\ \ !\ \ \mathsf{bool})\ \ !\ \ \mathsf{list}\ \ !\ \ \mathsf{Maybe\ nat} \\
\mathsf{find} := \lambda p.\lambda l.\ \mathsf{callcc}(\lambda k.\ \mathsf{list\_iter}\ (\lambda x.\ \mathsf{if}\ p\ x\ \mathsf{then} \\
\qquad \mathsf{throw}\ k\ (\mathsf{Some}\ \ x)\ \mathsf{else}\ ())\ l;\ \mathsf{None})
\end{array}
$$

Type of list_iter:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash list : \mathsf{list}}\,{:}_{var} \quad \dfrac{}{\Gamma_2 \vdash () : \mathsf{unit}}\,{:}_{unit} \quad \dfrac{\dfrac{\dfrac{f : \mathsf{nat}\ !\ \mathsf{unit}\ \mathit{2}\ \Gamma_3}{\Gamma_3 \vdash f : \mathsf{nat}\ !\ \mathsf{unit}}\,{:}_{var} \quad \dfrac{hd : \mathsf{nat}\ \mathit{2}\ \Gamma_3}{\Gamma_3 \vdash hd : \mathsf{nat}}\,{:}_{var}}{\Gamma_3 \vdash f\ hd : \mathsf{unit}}\,{:}_{app} \quad \textcircled{1}}{\Gamma_2, hd : \mathsf{nat}, tl : \mathsf{list} \vdash f\ hd\ ; list\_iter\ f\ tl : \mathsf{unit}}\,{:}_{comp}}{\Gamma_1, list : \mathsf{list} \vdash \mathsf{match}\ list\ \mathsf{with}\ \mathsf{Nil}\ !\ ()\ \mathit{j}\ \mathsf{Cons}(hd, tl)\ !\ f\ hd\ ; list\_iter\ f\ tl : \mathsf{unit}}\,{:}_{match}}{\Gamma_1 \vdash \lambda list.\ \mathsf{match}\ list\ \mathsf{with}\ \mathsf{Nil}\ !\ ()\ \mathit{j}\ \mathsf{Cons}(hd, tl)\ !\ f\ hd\ ; list\_iter\ f\ tl : \mathsf{list}\ !\ \mathsf{unit}}\,{:}_{abs}}{\Gamma \vdash \mu list\_iter.\lambda f.\lambda list.\ \mathsf{match}\ list\ \mathsf{with}\ \mathsf{Nil}\ !\ ()\ \mathit{j}\ \mathsf{Cons}(hd, tl)\ !\ f\ hd\ ; list\_iter\ f\ tl : (\mathsf{nat}\ !\ \mathsf{unit})\ !\ \mathsf{list}\ !\ \mathsf{unit}}\,{:}_{rec}$$

$\textcircled{1}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_3 \vdash list\_iter : (\mathsf{nat}\ !\ \mathsf{unit})\ !\ \mathsf{list}\ !\ \mathsf{unit}\ \mathit{2}\ \Gamma_3}\,{:}_{var} \quad \dfrac{f : \mathsf{nat}\ !\ \mathsf{unit}\ \mathit{2}\ \Gamma_3}{\Gamma_3 \vdash f : \mathsf{nat}\ !\ \mathsf{unit}}\,{:}_{var}}{\Gamma_3 \vdash list\_iter\ f : \mathsf{list}\ !\ \mathsf{unit}}\,{:}_{app} \quad \dfrac{tl : \mathsf{list}\ \mathit{2}\ \Gamma_3}{\Gamma_3 \vdash tl : \mathsf{list}}\,{:}_{var}}{\Gamma_3 \vdash list\_iter\ f\ tl : \mathsf{unit}}\,{:}_{app}$$

$\Gamma_1 = \Gamma, list\_iter : (\mathsf{nat}\ !\ \mathsf{unit})\ !\ \mathsf{list}\ !\ \mathsf{unit}, f : \mathsf{nat}\ !\ \mathsf{unit}$
$\Gamma_2 = \Gamma_1, list : \mathsf{list}$
$\Gamma_3 = \Gamma_2, hd : \mathsf{nat}, tl : \mathsf{list}$

Type of find:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{list\_iter} : (\text{nat } \mathit{!} \text{ unit}) \ \mathit{!} \ \text{list} \ \mathit{!} \ \text{unit } \mathit{2}\ \Gamma_4}{\Gamma_4 \vdash \text{list\_iter} : (\text{nat } \mathit{!} \text{ unit}) \ \mathit{!} \ \text{list} \ \mathit{!} \ \text{unit}}:_{var} \quad \textcircled{1}}{\Gamma_4 \vdash \text{list\_iter}\ (\lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } ()) : \text{list } \mathit{!} \text{ unit}}:_{app} \quad \dfrac{l : \text{list } \mathit{2}\ \Gamma_4}{\Gamma_4 \vdash l : \text{list}}:_{var}}{\Gamma_4 \vdash \text{list\_iter}\ (\lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } ())\ l : \text{unit}}:_{app} \quad \dfrac{}{\Gamma_4 \vdash \text{None} : \text{Maybe nat}}:_{none}}{\Gamma_3, k : \text{Maybe nat cont} \vdash \text{list\_iter}\ (\lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } ())\ l\ ;\ \text{None} : \text{Maybe nat}}:_{comp}}{\Gamma_3 \vdash \lambda k.\ \text{list\_iter}\ (\lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } ())\ l\ ;\ \text{None} : \text{Maybe nat cont } \mathit{!} \text{ Maybe nat}}:_{abs}}{\Gamma_2, l : \text{list} \vdash \text{callcc}(\lambda k.\ \text{list\_iter}\ (\lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } ())\ l\ ;\ \text{None}) : \text{Maybe nat}}:_{callcc}}{\Gamma_1, p : \text{nat } \mathit{!} \text{ bool} \vdash \lambda l.\ \text{callcc}(\lambda k.\ \text{list\_iter}\ (\lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } ())\ l\ ;\ \text{None}) : \text{list } \mathit{!} \text{ Maybe nat}}:_{abs}}{\Gamma_1 \vdash \lambda p.\lambda l.\ \text{callcc}(\lambda k.\ \text{list\_iter}\ (\lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } ())\ l\ ;\ \text{None}) : (\text{nat } \mathit{!} \text{ bool}) \ \mathit{!} \ \text{list} \ \mathit{!} \ \text{Maybe nat}}:_{abs}$$

$\textcircled{1}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p : \text{nat } \mathit{!} \text{ bool } \mathit{2}\ \Gamma_5}{\Gamma_5 \vdash p : \text{nat } \mathit{!} \text{ bool}}:_{var} \quad \dfrac{x : \text{nat } \mathit{2}\ \Gamma_5}{\Gamma_5 \vdash x : \text{nat}}:_{var}}{\Gamma_5 \vdash p\ x : \text{bool}}:_{app} \quad \dfrac{k : \text{Maybe nat cont } \mathit{2}\ \Gamma_5}{\Gamma_5 \vdash k : \text{Maybe nat cont}}:_{var} \quad \dfrac{\dfrac{x : \text{nat } \mathit{2}\ \Gamma_5}{\Gamma_5 \vdash x : \text{nat}}:_{var}}{\Gamma_5 \vdash \text{Some } x : \text{Maybe nat}}:_{some}}{\Gamma_5 \vdash \text{throw } k\ (\text{Some } x) : \text{unit}}:_{throw} \quad \dfrac{}{\Gamma_5 \vdash () : \text{unit}}:_{unit}}{\Gamma_4, x : \text{nat} \vdash \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } () : \text{unit}}:_{if}}{\Gamma_4 \vdash \lambda x.\ \text{if } p\ x \text{ then throw } k\ (\text{Some } x) \text{ else } () : \text{nat } \mathit{!} \text{ unit}}:_{abs}$$

$\Gamma_1 = \Gamma, \text{list\_iter} : (\text{nat } \mathit{!} \text{ unit}) \ \mathit{!} \ \text{list} \ \mathit{!} \ \text{unit}$
$\Gamma_2 = \Gamma_1, p : \text{nat } \mathit{!} \text{ bool}$
$\Gamma_3 = \Gamma_2, l : \text{list}$
$\Gamma_4 = \Gamma_3, k : \text{Maybe nat cont}$
$\Gamma_5 = \Gamma_4, x : \text{nat}$

A.4 Evaluation of $find\_one$

In the following we will evaluate the $find\_one$ program using evaluation contexts. The program makes use of the list_iter and find programs from Leroy [10] which are defined as follows.

$$\begin{aligned}\text{list\_iter} :=& \mu list\_iter.\lambda f.\lambda list.\text{match } list \text{ with Nil } ! \ () \\ & \text{ | Cons}(hd, tl) \ ! \ f \ hd \ ; \ list\_iter \ f \ tl\end{aligned}$$

$$\begin{aligned}\text{find} :=& \lambda p.\lambda l.\ \text{callcc } (\lambda k.\ \text{list\_iter } (\lambda x.\ \text{if } p\ x \text{ then} \\ & \text{throw } k \text{ (Some } x) \text{ else } ()) \ l; \text{ None})\end{aligned}$$

The two programs are combined into the program $find\_one$ which given a predicate and a list will iterate over the list. If an element is encountered that fulfills the predicate the element is returned. For better readability we abbreviate the list_iter program with iterPrg.

$$\begin{aligned}\text{iterPrg} :=& \mu list\_iter.\lambda f.\lambda list.\text{match } list \text{ with Nil } ! \ () \\ & \text{ | Cons}(hd, tl) \ ! \ f \ hd \ ; \ list\_iter \ f \ tl\end{aligned}$$

The program $find\_one$ is given by:

$$\begin{aligned}find\_one =& \text{let list\_iter} = \text{ iterPrg} \\ & \text{in (let find} = \lambda p.\lambda l.\ \text{callcc } (\lambda k.\text{list\_iter } (\lambda x.\ \text{if } p\ x \text{ then} \\ & \qquad \text{throw } k \text{ (Some } x) \text{ else } ()) \ l; \text{ None}) \\ & \text{in find } (\lambda x.\text{zero? Pred } x) \text{ Cons(Succ(Succ(0)), Cons(Succ(0), Nil)))}\end{aligned}$$

As predicate we have chosen the function ($\lambda x.$ zero? Pred x) where Pred is the function that returns the predecessor of its argument.

The evaluation shows that callcc saves the context E_2 which is simply a hole. The first element of the list is not Succ(0) therefore the result of the if-statement in the find program is (). Because of the composition () gets thrown away and iteration continues over the rest of the list. The second element of the list is Succ(0). The predicate function evaluates to true and evaluation continues with the "true"-branch of the if-statement. throw throws away the current context and applies the continuation ($\lambda x.E_2[x]$) captured by the callcc to Some Succ(0). Some Succ(0) gets passed to the hole of E_2 and since E_2 was only a hole, Some Succ(0) becomes the result value of the $find\_one$ program.

E[let list_iter = iterPrg in (let find =$\lambda p.\lambda l.$ callcc ($\lambda k.$ $list\_iter$ ($\lambda x.$ if p x then throw k (Some x) else ()) l ; None)
in find (λx.zero? Pred x) Cons(Succ(Succ(0)), Cons(Succ(0), Nil)))]
with $E = [\,]$

! E[let find =$\lambda p.\lambda l.$ callcc ($\lambda k.$ iterPrg ($\lambda x.$ if p x then throw k (Some x) else ()) l ; None)
in find (λx.zero? Pred x) Cons(Succ(Succ(0)), Cons(Succ(0), Nil))]
with $E = [\,]$

! E[($\lambda p.\lambda l.$ callcc ($\lambda k.$ iterPrg ($\lambda x.$ if p x then throw k (Some x) else ()) l ; None))(λx.zero? Pred x) Cons(Succ(Succ(0)), Cons(Succ(0), Nil))]
with $E = [\,]$

=E^1[($\lambda p.\lambda l.$ callcc ($\lambda k.$ iterPrg ($\lambda x.$ if p x then throw k (Some x) else ()) l ; None))(λx.zero? Pred x)]
with $E^1 = [\,]$ Cons(Succ(Succ(0)), Cons(Succ(0), Nil))

! E^1[$\lambda l.$ callcc ($\lambda k.$ iterPrg ($\lambda x.$ if ($\lambda x.$ zero? Pred x) x then throw k (Some x) else ()) l ; None)]
with $E^1 = [\,]$ Cons(Succ(Succ(0)), Cons(Succ(0), Nil))

=E^2[($\lambda l.$ callcc ($\lambda k.$ iterPrg ($\lambda x.$ if ($\lambda x.$ zero? Pred x) x then throw k (Some x) else ()) l ; None)) Cons(Succ(Succ(0)), Cons(Succ(0), Nil))]
with $E^2 = [\,]$

! E^2[callcc ($\lambda k.$ iterPrg ($\lambda x.$ if ($\lambda x.$ zero? Pred x) x then throw k (Some x) else ()) Cons(Succ(Succ(0)), Cons(Succ(0), Nil)) ; None)]
with $E^2 = [\,]$

! E^2[($\lambda k.$ iterPrg ($\lambda x.$ if ($\lambda x.$ zero? Pred x) x then throw k (Some x) else ()) Cons(Succ(Succ(0)), Cons(Succ(0), Nil)) ; None) ($\lambda x.E^2[x]$)]
with $E^2 = [\,]$

Notice how callcc gets hold of its continuation ($\lambda x.E^2[x]$) thereby saving the current context E^2.
The continuation is then passed as argument to callcc's argument.

! E^2[iterPrg ($\lambda x.$ if ($\lambda x.$ zero? Pred x) x then throw ($\lambda x.E^2[x]$) (Some x) else ()) Cons(Succ(Succ(0)), Cons(Succ(0), Nil)) ; None]
with $E^2 = [\,]$

! E^2[($\lambda x.\lambda y.y$) (iterPrg ($\lambda x.$ if ($\lambda x.$ zero? Pred x) x then throw ($\lambda x.E^2[x]$) (Some x) else ()) Cons(Succ(Succ(0)), Cons(Succ(0), Nil))) None]

with $E^2 = [\,]$

$=E^3[(\mu list\_iter.\lambda f.\lambda list.\text{match } list \text{ with Nil } !\ ()\ j\ \text{Cons}(hd, tl)\ !\ f\ hd\ ;\ list\_iter\ f\ tl)\ (\lambda x.\ \text{if } (\lambda x.\ \text{zero? Pred x})\ x \text{ then throw } (\lambda x.E^2[x])\ (\text{Some } x) \text{ else } ())]$

with $E^3 = (\lambda x.\lambda y.y)\ ([\,]\ \text{Cons}(\text{Succ}(\text{Succ}(0)), \text{Cons}(\text{Succ}(0), \text{Nil})))\ \text{None}$

For reasons of clarity let check $= (\lambda x.\ \text{if } (\lambda x.\ \text{zero? Pred x})\ x \text{ then throw } (\lambda x.E^2[x])\ (\text{Some } x) \text{ else } ())$
Furthermore iterPrg is used again as representation for the list_iter code generated by the recursive call.

$!\ E^3[\lambda list.\text{match } list \text{ with Nil } !\ ()\ j\ \text{Cons}(hd, tl)\ !\ \text{check } hd\ ;\ \text{iterPrg check } tl]$

with $E^3 = (\lambda x.\lambda y.y)\ ([\,]\ \text{Cons}(\text{Succ}(\text{Succ}(0)), \text{Cons}(\text{Succ}(0), \text{Nil})))\ \text{None}$

$=E^4[(\lambda list.\text{match } list \text{ with Nil } !\ ()\ j\ \text{Cons}(hd, tl)\ !\ \text{check } hd\ ;\ \text{iterPrg check } tl)\ \text{Cons}(\text{Succ}(\text{Succ}(0)), \text{Cons}(\text{Succ}(0), \text{Nil}))]$

with $E^4 = (\lambda x.\lambda y.y)\ [\,]\ \text{None}$

$!\ E^4[\text{match Cons}(\text{Succ}(\text{Succ}(0)), \text{Cons}(\text{Succ}(0), \text{Nil})) \text{ with Nil } !\ ()\ j\ \text{Cons}(hd, tl)\ !\ \text{check } hd\ ;\ \text{iterPrg check } tl]$

with $E^4 = (\lambda x.\lambda y.y)\ [\,]\ \text{None}$

$!\ E^4[\text{check Succ}(\text{Succ}(0))\ ;\ \text{iterPrg check Cons}(\text{Succ}(0), \text{Nil})]$

with $E^4 = (\lambda x.\lambda y.y)\ [\,]\ \text{None}$

$!\ E^4[(\lambda x.\lambda y.y)\ (\text{check Succ}(\text{Succ}(0)))\ (\text{iterPrg check Cons}(\text{Succ}(0), \text{Nil}))]$

with $E^4 = (\lambda x.\lambda y.y)\ [\,]\ \text{None}$

Evaluation continues with the first part of the composition. Therefore we have to fill in the code for check and apply it to Succ(Succ(0)).

$=E^5[(\lambda x.\ \text{if } (\lambda x.\ \text{zero? Pred x})\ x \text{ then throw } (\lambda x.E^2[x])\ (\text{Some } x) \text{ else } ())\ \text{Succ}(\text{Succ}(0))]$

with $E^5 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)\ [\,]\ (\text{iterPrg check Cons}(\text{Succ}(0), \text{Nil})))\ \text{None}$

$!\ E^5[\text{if } (\lambda x.\ \text{zero? Pred x})\ \text{Succ}(\text{Succ}(0)) \text{ then throw } (\lambda x.E^2[x])\ (\text{Some Succ}(\text{Succ}(0))) \text{ else } ()]$

with $E^5 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)\ [\,]\ (\text{iterPrg check Cons}(\text{Succ}(0), \text{Nil})))\ \text{None}$

$=E^6[(\lambda x.\text{zero? Pred } x)\ \text{Succ}(\text{Succ}(0))]$

with $E^6 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if [ ] then throw $(\lambda x.E^2[x])$ (Some Succ(Succ(0))) else ()) (iterPrg check Cons(Succ(0), Nil))) None

! E^6[zero? Pred Succ(Succ(0))]

with $E^6 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if [ ] then throw $(\lambda x.E^2[x])$ (Some Succ(Succ(0))) else ()) (iterPrg check Cons(Succ(0), Nil))) None

=E^7[Pred Succ(Succ(0))]

with $E^7 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if zero?[ ] then throw $(\lambda x.E^2[x])$ (Some Succ(Succ(0))) else ()) (iterPrg check Cons(Succ(0), Nil))) None

! E^7[Succ(0)]

with $E^7 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if zero?[ ] then throw $(\lambda x.E^2[x])$ (Some Succ(Succ(0))) else ()) (iterPrg check Cons(Succ(0), Nil))) None

=E^8[zero? Succ(0)]

with $E^8 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if [ ] then throw $(\lambda x.E^2[x])$ (Some Succ(Succ(0))) else ()) (iterPrg check Cons(Succ(0), Nil))) None

! E^8[false]

with $E^8 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if [ ] then throw $(\lambda x.E^2[x])$ (Some Succ(Succ(0))) else ()) (iterPrg check Cons(Succ(0), Nil))) None

=E^9[if false then throw $(\lambda x.E^2[x])$ (Some Succ(Succ(0))) else ()]

with $E^9 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)$ [ ] (iterPrg check Cons(Succ(0), Nil))) None

! E^9[()]

with $E^9 = (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)$ [ ] (iterPrg check Cons(Succ(0), Nil))) None

! E^{10}[$(\lambda x.\lambda y.y)$ ()]

with $E^{10} = (\lambda x.\lambda y.y)$ ([ ] (iterPrg check Cons(Succ(0), Nil))) None

! E^{10}[$(\lambda y.y)$]

with $E^{10} = (\lambda x.\lambda y.y)$ ([ ] (iterPrg check Cons(Succ(0), Nil))) None

We continue with the evaluation of the second element of the composition. Therefore iterPrg is applied to its first argument check. To do this we fill in the code for both iterPrg and check.

=E^{11}[($\mu list\_iter.\lambda f.\lambda list$.match $list$ with Nil ! () *j* Cons(hd, tl) ! $f\ hd$; $list\_iter\ f\ tl$) (λx. if (λx. zero? Pred x) x then throw $(\lambda x.E^2[x])$ (Some x) else ())]

with $E^{11} = (\lambda x.\lambda y.y)\ ((\lambda y.y)$ ([ ] Cons(Succ(0), Nil))) None

check is used again to represent $(\lambda x.$ if $(\lambda x.$ zero? Pred x$)$ x then throw $(\lambda x.E^2[x])$ (Some x) else $())$ and iterPrg is used as representation for the list_iter code generated by the recursive call.

$!\ E^{11}[\lambda list.$match $list$ with Nil $!$ $()$ j Cons(hd, tl) $!$ check hd ; iterPrg check $tl]$
with $E^{11} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ([\,]$ Cons(Succ(0), Nil$)))$ None
$=E^{12}[(\lambda list.$match $list$ with Nil $!$ $()$ j Cons(hd, tl) $!$ check hd ; iterPrg check $tl)$ Cons(Succ(0), Nil$)]$
with $E^{12} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ [\,])$ None
$!\ E^{12}[$match Cons(Succ(0), Nil) with Nil $!$ $()$ j Cons(hd, tl) $!$ check hd ; iterPrg check $tl]$
with $E^{12} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ [\,])$ None
$!\ E^{12}[$check Succ(0) ; iterPrg check Nil$]$
with $E^{12} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ [\,])$ None
$!\ E^{12}[(\lambda x.\lambda y.y)$ (check Succ(0)) (iterPrg check Nil$)]$
with $E^{12} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ [\,])$ None
$=E^{13}[(\lambda x.$ if $(\lambda x.$ zero? Pred x$)$ x then throw $(\lambda x.E^2[x])$ (Some x) else $())$ Succ$(0)]$
with $E^{13} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)\ [\,]$ (iterPrg check Nil$)))$ None
$!\ E^{13}[$if $(\lambda x.$ zero? Pred x$)$ Succ(0) then throw $(\lambda x.E^2[x])$ (Some Succ(0)) else $()]$
with $E^{13} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)\ [\,]$ (iterPrg check Nil$)))$ None
$=E^{14}[(\lambda x.$zero? Pred $x)$ Succ$(0)]$
with $E^{14} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if $[\,]$ then throw $(\lambda x.E^2[x])$ (Some Succ(0)) else $())$ (iterPrg check Nil$)))$ None
$!\ E^{14}[$zero? Pred Succ$(0)]$
with $E^{14} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if $[\,]$ then throw $(\lambda x.E^2[x])$ (Some Succ(0)) else $())$ (iterPrg check Nil$)))$ None
$=E^{15}[$Pred Succ$(0)]$
with $E^{15} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if zero? $[\,]$ then throw $(\lambda x.E^2[x])$ (Some Succ(0)) else $())$ (iterPrg check Nil$)))$ None
$!\ E^{15}[0]$

with $E^{15} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if zero? [ ] then throw $(\lambda x.E^2[x])$ (Some Succ(0)) else ()) (iterPrg check Nil))) None

$=E^{16}$[zero? 0]

with $E^{16} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if [ ] then throw $(\lambda x.E^2[x])$ (Some Succ(0)) else ()) (iterPrg check Nil))) None

! E^{16}[true]

with $E^{16} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)$ (if [ ] then throw $(\lambda x.E^2[x])$ (Some Succ(0)) else ()) (iterPrg check Nil))) None

$=E^{17}$[if true then throw $(\lambda x.E^2[x])$ (Some Succ(0)) else ()]

with $E^{17} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)$ [ ] (iterPrg check Nil))) None

! E^{17}[throw $(\lambda x.E^2[x])$ (Some Succ(0))]

with $E^{17} = (\lambda x.\lambda y.y)\ ((\lambda y.y)\ ((\lambda x.\lambda y.y)$ [ ] (iterPrg check Nil))) None

! $(\lambda x.E^2[x])$ (Some Succ(0))

Notice how throw jumps out of the depth of E^{17} and restores the evaluation context E^2 that was saved earlier by callcc.

! E^2[(Some Succ(0))]

with $E^2 = [\ ]$

A.5 Evaluation of *print_all*

In [10] Leroy defines a program printall which prints all elements in a list that fulfill a certain predicate. The usage of callcc in this program makes backtracking possible so that after printing an element the search for further elements that fulfill the predicate continues starting from the last found element.

The program printall makes use of the earlier defined programs list_iter and find. Inside the find program, however, a second callcc is added.

$$\begin{aligned}\mathsf{find} :=& \lambda p.\lambda l.\ \mathsf{callcc}\ (\lambda k.\ \mathsf{list\_iter}\ (\lambda x.\ \mathsf{if}\ p\ x\ \mathsf{then}\\ &\quad \mathsf{callcc}\ (\lambda k^{0}\ \mathsf{throw}\ k\ (\mathsf{Some}\ (x,\ k^{0}))\ \mathsf{else}\ ())\ l;\ \mathsf{None})\end{aligned}$$

In miniFP^{+} Leroy's program printall is defined by combining the programs printall, the modified find and list_iter into a program *print_all*. For better readability the code for list_iter and find is replaced by the placeholders iterPrg and findPrg respectively. The program printall is executed with the predicate function (λx.zero? Pred x) and list Cons(Succ(0), Cons(Succ(0), Nil)).

$$\begin{aligned}\mathit{print\_all} = \ & \mathsf{let\ list\_iter} = \ \mathsf{iterPrg}\\ & \mathsf{in\ (let\ find} = \ \mathsf{findPrg}\\ & \mathsf{in\ (let\ printall} = \lambda p.\lambda l.\mathsf{match\ find}\ p\ l\ \mathsf{with\ None}\ \ !\ \ ()\\ & \qquad\qquad\qquad\qquad j\ \mathsf{Some}\ (x,k)\ \ !\ \ \mathsf{print\_string}\ x\ ;\ \mathsf{throw}\ k\ ()\\ & \mathsf{in\ printall}\ (\lambda x.\mathsf{zero?\ Pred}\ x)\ \mathsf{Cons(Succ(0), Cons(Succ(0), Nil))))}\end{aligned}$$

As before Pred is the function that returns the predecessor of its argument. Furthermore assume that the reduction rules for match are changed to accommodate for None and Some(x, k) and that print_string is a function that prints its argument.

The evaluation shows that the use of the two callcc's makes it possible to jump between printing and iterating. If an element fulfilling the predicate is found, throw invokes the continuation of the outer callcc with the element passed as argument. The context of the outer callcc is restored and the element is printed. However, at the moment an element was found, the inner callcc captured another continuation which saved the context and thereby the position in the list. After the element is printed this continuation is invoked and iteration over the list resumes from the saved position.

We will only show the important steps during the evaluation that show how backtracking is achieved. Multiple reduction steps are denoted by $!$.

E[let list_iter = iterPrgin (let find = findPrgin (let printall = $\lambda p.\lambda l.$match find p l with None $!$ () j Some (x, k) $!$ print_string x ; throw k ()
in printall ($\lambda x.$zero? Pred x) Cons(Succ(0), Cons(Succ(0), Nil)))))]
with $E = [\,]$

$!$ E[match findPrg ($\lambda x.$zero? Pred x) Cons(Succ(0), Cons(Succ(0), Nil)) with None $!$ () j Some (x, k) $!$ print_string x ; throw k ()]
with $E = [\,]$

Evaluation continues with the findPrg. Therefore we fill in the two arguments supplied to it.

$!$ E_1[callcc ($\lambda k.$ iterPrg ($\lambda x.$ if ($\lambda x.$zero? Pred x) x then callcc ($\lambda k^0.$ throw k (Some (x, k^0))) else ()) Cons(Succ(0), Cons(Succ(0), Nil)) ; None)]
with E_1 = match [ ] with None $!$ () j Some (x, k) $!$ print_string x ; throw k ()

$!$ E_1[($\lambda k.$ iterPrg ($\lambda x.$ if ($\lambda x.$zero? Pred x) x then callcc ($\lambda k^0.$ throw k (Some (x, k^0))) else ()) Cons(Succ(0), Cons(Succ(0), Nil)) ; None) ($\lambda x.E_1[x]$)]
with E_1 = match [ ] with None $!$ () j Some (x, k) $!$ print_string x ; throw k ()

By evaluating the outer callcc the current continuation ($\lambda x.E_1[x]$) is passed as argument to the function of callcc. The continuation is substituted for every occurrence of k, so the k inside the function passed to the iterPrg becomes the continuation ($\lambda x.E_1[x]$). Because the function remains the same for every call to iterPrg, ($\lambda x.E_1[x]$) stays inside the function throughout the whole evaluation. This makes it possible for every evaluation of iterPrg to jump to the same context. The continuation ($\lambda x.E_1[x]$) will pass its argument to the context E_1 which is match [ ] with None $!$ () j Some (x, k) $!$ print_string x ; throw k ().
If the argument passed is of the form Some (x, k) then the x is printed. Because ($\lambda x.E_1[x]$) restores the context of the outer callcc where something can be printed we will call ($\lambda x.E_1[x]$) the print continuation k_{print}.

$!$ E_1[iterPrg ($\lambda x.$ if ($\lambda x.$zero? Pred x) x then callcc ($\lambda k^0.$ throw k_{print} (Some (x, k^0))) else ()) Cons(Succ(0), Cons(Succ(0), Nil)) ; None]
with E_1 = match [ ] with None $!$ () j Some (x, k) $!$ print_string x ; throw k ()

Evaluation continues with the first part of the composition. The code for iterPrg is filled in and applied to its argument.
For readability let check $= (\lambda x.$ if $(\lambda x.$zero? Pred $x)$ x then callcc $(\lambda k^0.$ throw k_{print} (Some (x, k^0))) else ()) . We use iterPrg again as representation for the code generated by the recursive call.
The list supplied to iterPrg matches with the second case Cons(hd, tl) and so evaluation continues with the first part of the composition check Succ(0) ; iterPrg check Cons(Succ(0), Nil)

! $E_2[(\lambda x.$ if $(\lambda x.$zero? Pred $x)$ x then callcc $(\lambda k^0.$ throw k_{print} (Some (x, k^0))) else ()) Succ(0)]
with $E_2 =$ match $(\lambda x.\lambda y.y)$ $((\lambda x.\lambda y.y)$ [ ] (iterPrg checkCons(Succ(0), Nil))) None with None ! () *j* Some (x, k) ! print_string x ; throw k ()
! E_2[callcc $(\lambda k^0.$ throw k_{print} (Some (Succ(0), k^0)))]
with $E_2 =$ match $(\lambda x.\lambda y.y)$ $((\lambda x.\lambda y.y)$ [ ] (iterPrg checkCons(Succ(0), Nil))) None with None ! () *j* Some (x, k) ! print_string x ; throw k ()
! $E_2[(\lambda k^0.$ throw k_{print} (Some (Succ(0), k^0))) $(\lambda x.E_2[x])]$
with $E_2 =$ match $(\lambda x.\lambda y.y)$ $((\lambda x.\lambda y.y)$ [ ] (iterPrg check Cons(Succ(0), Nil))) None with None ! () *j* Some (x, k) ! print_string x ; throw k ()

The continuation $(\lambda x.E_2[x])$ captured by the inner callcc saves the current context E_2.
This way if a value gets passed to it, evaluation resumes within the context E_2. To be more precise evaluation resumes with iterPrg check Cons(Succ(0), Nil) which iterates over the rest of the list starting after the current element. Therefore we will be calling the continuation k_{pos} because it saves the position in the list thereby making it possible to *backtrack* to the last checked element in the list.

! E_2[throw k_{print} (Some (Succ(0), k_{pos}))]
with $E_2 =$ match $(\lambda x.\lambda y.y)$ $((\lambda x.\lambda y.y)$ [ ] (iterPrg check Cons(Succ(0), Nil))) None with None ! () *j* Some (x, k) ! print_string x ; throw k ()
! k_{print} (Some (Succ(0), k_{pos}))
! E_1[Some (Succ(0), k_{pos})]
with $E_1 =$ match [ ] with None ! () *j* Some (x, k) ! print_string x ; throw k ()

throw throws away the current context E_2 and restores the context E_1 of the outer callcc. This happens by applying the print-continuation k_{print} to the argument Some(Succ(0), k_{pos}) which is then passed into the hole of the context.

$!\ E_3[\mathsf{print\_string}\ \mathsf{Succ}(0)\ ;\ \mathsf{throw}\ k_{\mathrm{pos}}\ ()]$
with $E_3 = [\,]$

Evaluation of the first part of the composition prints $\mathsf{Succ}(0)$, the value is thrown away and evaluation continues with $\mathsf{throw}\ k_{\mathrm{pos}}\ ()$.

$!\ E_3[\mathsf{throw}\ k_{\mathrm{pos}}\ ()]$
with $E_3 = [\,]$
$!\ k_{\mathrm{pos}}\ ()$
$!\ E_2[()]$
with $E_2 = \mathsf{match}\ (\lambda x.\lambda y.y)\ ((\lambda x.\lambda y.y)\ [\,]\ (\mathsf{iterPrg\ check\ Cons}(\mathsf{Succ}(0), \mathsf{Nil})))\ \mathsf{None\ with\ None}\ !\ ()\ j\ \mathsf{Some}\ (x,k)\ !\ \mathsf{print\_string}\ x\ ;\ \mathsf{throw}\ k\ ()$
$!\ E_4[\mathsf{iterPrg\ check\ Cons}(\mathsf{Succ}(0), \mathsf{Nil})]$
with $E_4 = \mathsf{match}\ (\lambda x.\lambda y.y)\ ((\lambda y.y)\ [\,])\ \mathsf{None\ with\ None}\ !\ ()\ j\ \mathsf{Some}\ (x,k)\ !\ \mathsf{print\_string}\ x\ ;\ \mathsf{throw}\ k\ ()$

By evaluating $\mathsf{throw}\ k_{\mathrm{pos}}\ ()$ the list-position continuation is invoked with $()$. k_{pos} restores the context E_2 in which the position in the list was saved. Evaluation continues therefore with the iteration over the rest of the list $\mathsf{Cons}(\mathsf{Succ}(0), \mathsf{Nil})$ that had not been checked yet.

If the list $\mathsf{Cons}(\mathsf{Succ}(0), \mathsf{Nil})$ contains another element that fulfills the predicate, the element will get passed to k_{print} together with the continuation of the inner callcc. This way the current context and therefore the current position in list will be saved. k_{print} will restore the context of the outer callcc and the element will be printed. Afterwards the continuation of the inner callcc is invoked and its context restored. Evaluation will continue with the rest of the list.
The usage of an outer and inner callcc make it possible to jump to the context of the outer callcc to print something and then jump back to the context of the inner callcc to resume iterating over the list.

A.6 Proof of Proposition 4.2.1

Proposition 4.2.1. *For all values v and terms M in* miniFP$^+$ *the following holds.*

1. *If* $\Gamma \vdash v : \sigma$ *then* $\overline{\Gamma} \vdash \jmath v \jmath : \overline{\sigma}$
2. *If* $\Gamma \vdash M : \sigma$ *then* $\overline{\Gamma} \vdash \mathtt{J}M\mathtt{K} : \jmath\sigma\jmath$

Where $\overline{\Gamma} = x : \overline{\sigma}, \ldots$.

Proof. The proof for both parts is by induction on the derivation of $\Gamma \vdash v : \sigma$ and $\Gamma \vdash M : \sigma$ respectively and a case analysis of the terms.

Case $v = 0$**:** Assuming $\Gamma \vdash 0 : \mathsf{nat}$, we need to show $\overline{\Gamma} \vdash \jmath 0 \jmath : \mathsf{nat}$. Which is the same as $\overline{\Gamma} \vdash 0 : \mathsf{nat}$ and holds trivially.

The proofs for the cases true, false, (), None and Nil are similar.

Case $v = \mathsf{Some}\ v^0$ Assuming $\Gamma \vdash \mathsf{Some}\ v^0 : \mathsf{Maybe}\ \tau$ we have the induction hypothesis IH: $\overline{\Gamma} \vdash \left|v^0\right| : \overline{\tau}$. Therefore by the *some* rule we have $\overline{\Gamma} \vdash \mathsf{Some}\ \left|v^0\right| : \mathsf{Maybe}\ \overline{\tau}$ which by the type and value translation is $\overline{\Gamma} \vdash \left|\mathsf{Some}\ v^0\right| : \overline{\mathsf{Maybe}\ \tau}$.

The proof for the case $\mathsf{Succ}(v)$ is similar.

Case $v = \mathsf{Cons}(v_1, v_2)$ Assuming $\Gamma \vdash \mathsf{Cons}(v_1, v_2) : \mathsf{list}$ we have the induction hypotheses IH_1: $\overline{\Gamma} \vdash \jmath v_1 \jmath : \mathsf{nat}$ and IH_2: $\overline{\Gamma} \vdash \jmath v_2 \jmath : \mathsf{list}$. Therefore by the *cons* rule we have $\overline{\Gamma} \vdash \mathsf{Cons}(\jmath v_1 \jmath, \jmath v_2 \jmath) : \mathsf{list}$ which by the type and value translation is $\overline{\Gamma} \vdash \left|\mathsf{Cons}(v_1, v_2)\right| : \overline{\mathsf{list}}$.

Case $v = \lambda x.a$**:** Assuming $\Gamma \vdash \lambda x.a : \sigma \ ! \ \tau$ we have the induction hypothesis IH: $\overline{\Gamma}, x : \overline{\sigma} \vdash \mathtt{J}a\mathtt{K} : (\overline{\tau} \ ! \ \alpha) \ ! \ \alpha$. Therefore by the *abs* rule we have $\overline{\Gamma} \vdash \lambda x.\mathtt{J}a\mathtt{K} : \overline{\sigma} \ ! \ ((\overline{\tau} \ ! \ \alpha) \ ! \ \alpha)$ which by the type and value translation is $\overline{\Gamma} \vdash \jmath \lambda x.a \jmath : \overline{\sigma \ ! \ \tau}$.

Case $v = \mu f.\lambda x.a$ Assuming $\Gamma \vdash \mu f.\lambda x.a : \sigma \ ! \ \tau$ we have the induction hypothesis IH: $\overline{\Gamma}, f : \overline{\sigma \ ! \ \tau}, x : \overline{\sigma} \vdash \mathtt{J}a\mathtt{K} : \jmath\tau\jmath = \overline{\Gamma}, f : \overline{\sigma} \ ! \ ((\overline{\tau} \ ! \ \alpha) \ ! \ \alpha), x : \overline{\sigma} \vdash \mathtt{J}a\mathtt{K} : (\overline{\tau} \ ! \ \alpha) \ ! \ \alpha$. Therefore by the application of the *rec* rule we have $\overline{\Gamma} \vdash \mu f.\lambda x.\ \mathtt{J}a\mathtt{K} : \overline{\sigma} \ ! \ ((\overline{\tau} \ ! \ \alpha) \ ! \ \alpha)$. which by the type and value translation is $\overline{\Gamma} \vdash \jmath \mu f.\lambda x.\ a \jmath : \overline{\sigma \ ! \ \tau}$.

Case $M = x$**:** Assuming $\Gamma \vdash x : \sigma$ we have the induction hypothesis IH: $x : \overline{\sigma}\ \mathit{2}\ \overline{\Gamma}$. We need to show $\overline{\Gamma} \vdash \mathtt{J}x\mathtt{K} : (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha$

$$\dfrac{\dfrac{\dfrac{k : \overline{\sigma} \ ! \ \alpha\ \mathit{2}\ \overline{\Gamma}_1}{\overline{\Gamma}_1 \vdash k : \overline{\sigma} \ ! \ \alpha} :_{var} \quad \dfrac{x : \overline{\sigma}\ \mathit{2}\ \overline{\Gamma}_1 = x : \overline{\sigma}\ \mathit{2}\ \overline{\Gamma}, k : \overline{\sigma} \ ! \ \alpha = \mathrm{IH}}{\overline{\Gamma}_1 \vdash x : \overline{\sigma}} :_{var}}{\overline{\Gamma}, k : \overline{\sigma} \ ! \ \alpha \vdash k\ x : \alpha} :_{app}}{\overline{\Gamma} \vdash \lambda k.\ k\ x : (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha} :_{abs}$$

$\overline{\Gamma}_1 = \overline{\Gamma}, k : \overline{\sigma} \ ! \ \alpha$

Case $M = \mathsf{Some}\ a$**:** Assuming $\Gamma \vdash \mathsf{Some}\ a : \mathsf{Maybe}\ \sigma$ we have the induction hypothesis IH: $\overline{\Gamma} \vdash \mathtt{J}a\mathtt{K} : (\overline{\sigma} \ ! \ \alpha) \ ! \ \alpha$. We need to show $\overline{\Gamma} \vdash \mathtt{JSome}\ a\mathtt{K} : (\mathsf{Maybe}\ \overline{\sigma} \ ! \ \alpha) \ ! \ \alpha$.

$$\dfrac{\overline{\Gamma}_1 \text{ ` } \mathsf{J}a\mathsf{K} : (\overline{\sigma} \;!\; \alpha) \;!\; \alpha = \mathrm{IH} \qquad \dfrac{\dfrac{k : \mathsf{Maybe}\ \overline{\sigma} \;!\; \alpha\ \mathit{2}\ \overline{\Gamma}_2}{\overline{\Gamma}_2 \text{ ` } k : \mathsf{Maybe}\ \overline{\sigma} \;!\; \alpha}:_{var} \quad \dfrac{\dfrac{v_a : \overline{\sigma}\ \mathit{2}\ \overline{\Gamma}_2}{\overline{\Gamma}_2 \text{ ` } v_a : \overline{\sigma}}:_{var}}{\overline{\Gamma}_2 \text{ ` } \mathsf{Some}\ v_a : \mathsf{Maybe}\ \overline{\sigma}}:_{some}}{\overline{\Gamma}_2 \text{ ` } k\ \mathsf{Some}\ v_a : \alpha}:_{app}}{\overline{\Gamma}_1 \text{ ` } \lambda v_a.\ k\ \mathsf{Some}\ v_a : \overline{\sigma} \;!\; \alpha}:_{abs}}{\overline{\Gamma}, k : \mathsf{Maybe}\ \overline{\sigma} \;!\; \alpha \text{ ` } \mathsf{J}a\mathsf{K}\ (\lambda v_a.\ k\ \mathsf{Some}\ v_a) : \alpha}:_{app}}{\overline{\Gamma} \text{ ` } \lambda k.\ \mathsf{J}a\mathsf{K}\ (\lambda v_a.\ k\ \mathsf{Some}\ v_a) : (\mathsf{Maybe}\ \overline{\sigma} \;!\; \alpha) \;!\; \alpha}:_{abs}$$

$\overline{\Gamma}_1 = \overline{\Gamma}, k : \mathsf{Maybe}\ \overline{\sigma} \;!\; \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma}_1, v_a : \overline{\sigma}$

The proofs for the cases $\mathsf{Succ}(a)$ and $\mathsf{zero?}\ a$ are similar.

Case $M = \mathsf{Cons}(a, b)$: Assuming $\Gamma \text{ ` } \mathsf{Cons}(a, b) : \mathsf{list}$ we have the induction hypotheses $\mathrm{IH}_1$: $\overline{\Gamma} \text{ ` } \mathsf{J}a\mathsf{K} : (\mathsf{nat} \;!\; \alpha) \;!\; \alpha$ and IH_2: $\overline{\Gamma} \text{ ` } \mathsf{J}b\mathsf{K} : (\mathsf{list} \;!\; \alpha) \;!\; \alpha$. We need to show $\overline{\Gamma} \text{ ` } \mathsf{JCons}(a, b)\mathsf{K} : (\mathsf{list} \;!\; \alpha) \;!\; \alpha$.

$$\dfrac{\overline{\Gamma}_1 \text{ ` } \mathsf{J}a\mathsf{K} : (\mathsf{nat} \;!\; \alpha) \;!\; \alpha = \mathrm{IH}_1 \qquad \dfrac{\dfrac{\overline{\Gamma}_2 \text{ ` } \mathsf{J}b\mathsf{K} : (\mathsf{list} \;!\; \alpha) \;!\; \alpha = \mathrm{IH}_2 \qquad \textcircled{1}}{\overline{\Gamma}_1, v_a : \mathsf{nat} \text{ ` } \mathsf{J}b\mathsf{K}\ (\lambda v_b.\ k\ \mathsf{Cons}(v_a, v_b)) : \alpha}:_{app}}{\overline{\Gamma}_1 \text{ ` } \lambda v_a.\ \mathsf{J}b\mathsf{K}\ (\lambda v_b.\ k\ \mathsf{Cons}(v_a, v_b)) : \mathsf{nat} \;!\; \alpha}:_{abs}}{\overline{\Gamma}, k : \mathsf{list} \;!\; \alpha \text{ ` } \mathsf{J}a\mathsf{K}\ (\lambda v_a.\ \mathsf{J}b\mathsf{K}\ (\lambda v_b.\ k\ \mathsf{Cons}(v_a, v_b))) : \alpha}:_{app}}{\overline{\Gamma} \text{ ` } \lambda k.\ \mathsf{J}a\mathsf{K}\ (\lambda v_a.\ \mathsf{J}b\mathsf{K}\ (\lambda v_b.\ k\ \mathsf{Cons}(v_a, v_b))) : (\mathsf{list} \;!\; \alpha) \;!\; \alpha}:_{abs}$$

$$\textcircled{1}: \dfrac{\dfrac{k : \mathsf{list} \;!\; \alpha\ \mathit{2}\ \overline{\Gamma}_3}{\overline{\Gamma}_3 \text{ ` } k : \mathsf{list} \;!\; \alpha}:_{var} \quad \dfrac{\dfrac{v_a : \mathsf{nat}\ \mathit{2}\ \overline{\Gamma}_3}{\overline{\Gamma}_3 \text{ ` } v_a : \mathsf{nat}}:_{var} \quad \dfrac{v_b : \mathsf{list}\ \mathit{2}\ \overline{\Gamma}_3}{\overline{\Gamma}_3 \text{ ` } v_b : \mathsf{list}}:_{var}}{\overline{\Gamma}_3 \text{ ` } \mathsf{Cons}(v_a, v_b) : \mathsf{list}}:_{cons}}{\overline{\Gamma}_2, v_b : \mathsf{list} \text{ ` } k\ \mathsf{Cons}(v_a, v_b) : \alpha}:_{app}}{\overline{\Gamma}_2 \text{ ` } \lambda v_b.\ k\ \mathsf{Cons}(v_a, v_b) : \mathsf{list} \;!\; \alpha}:_{abs}$$

$\overline{\Gamma}_1 = \overline{\Gamma}, k : \mathsf{list} \;!\; \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma}_1, v_a : \mathsf{nat}$
$\overline{\Gamma}_3 = \overline{\Gamma}_2, v_b : \mathsf{list}$

The proofs for the cases $a\ ;\ b$ and $a + b$ are similar.

Case $M = \mathsf{callcc}\ a$**:** Assuming $\Gamma \text{ ` } \mathsf{callcc}\ a : \sigma$ we have the induction hypothesis IH: $\overline{\Gamma} \text{ ` } \mathsf{J}a\mathsf{K} : j\sigma\ \mathsf{cont} \;!\; \sigma j = \overline{\Gamma} \text{ ` } \mathsf{J}a\mathsf{K} : ((\overline{\sigma} \;!\; \alpha \;!\; (\overline{\sigma} \;!\; \alpha) \;!\; \alpha) \;!\; \alpha) \;!\; \alpha$. We need to show $\overline{\Gamma} \text{ ` } \mathsf{Jcallcc}\ a\mathsf{K} : (\overline{\sigma} \;!\; \alpha) \;!\; \alpha$.

$$\dfrac{\dfrac{\dfrac{\dfrac{f : \delta \mathit{2}\ \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash f : \delta}:_{var} \quad \dfrac{k : \overline{\sigma}\ \mathit{!}\ \alpha \mathit{2}\ \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash k : \overline{\sigma}\ \mathit{!}\ \alpha}:_{var}}{\overline{\Gamma}_2 \vdash f\ k : (\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha}:_{app} \quad \dfrac{k : \overline{\sigma}\ \mathit{!}\ \alpha \mathit{2}\ \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash k : \overline{\sigma}\ \mathit{!}\ \alpha}:_{var}}{\overline{\Gamma}_1, f : \delta \vdash f\ k\ k : \alpha}:_{app}}{\overline{\Gamma}_1 \vdash \lambda f.\ f\ k\ k : \gamma}:_{abs}}{\overline{\Gamma}, k : \overline{\sigma}\ \mathit{!}\ \alpha \vdash \mathtt{J}a\mathtt{K}\ (\lambda f.\ f\ k\ k) : \alpha}:_{app}}{\overline{\Gamma} \vdash \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda f.\ f\ k\ k) : (\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha}:_{abs}$$

$\overline{\Gamma}_1 \vdash \mathtt{J}a\mathtt{K} : \gamma\ \mathit{!}\ \alpha = \mathrm{IH}$

$\gamma = (\overline{\sigma}\ \mathit{!}\ \alpha\ \mathit{!}\ (\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$
$\delta = \overline{\sigma}\ \mathit{!}\ \alpha\ \mathit{!}\ (\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$
$\overline{\Gamma}_1 = \overline{\Gamma}, k : \overline{\sigma}\ \mathit{!}\ \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma}_1, f : \delta$

Case $M = \mathsf{throw}\ a\ b$**:** Assuming $\Gamma \vdash \mathsf{throw}\ a\ b : \sigma$ we have the induction hypotheses IH_1: $\overline{\Gamma} \vdash \mathtt{J}a\mathtt{K} : j\tau\ \mathsf{cont}j = \overline{\Gamma} \vdash \mathtt{J}a\mathtt{K} : ((\overline{\tau}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$ and IH_2: $\overline{\Gamma} \vdash \mathtt{J}b\mathtt{K} : (\overline{\tau}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$. We need to show $\overline{\Gamma} \vdash \mathtt{J}\mathsf{throw}\ a\ b\mathtt{K} : (\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{v_a : \overline{\tau}\ \mathit{!}\ \alpha \mathit{2}\ \overline{\Gamma}_3}{\overline{\Gamma}_3 \vdash v_a : \overline{\tau}\ \mathit{!}\ \alpha}:_{var} \quad \dfrac{v_b : \overline{\tau} \mathit{2}\ \overline{\Gamma}_3}{\overline{\Gamma}_3 \vdash v_b : \overline{\tau}}:_{var}}{\overline{\Gamma}_2, v_b : \overline{\tau} \vdash v_a\ v_b : \alpha}:_{app} \quad \dfrac{}{\overline{\Gamma}_2 \vdash \lambda v_b.\ v_a\ v_b : \overline{\tau}\ \mathit{!}\ \alpha}:_{abs}}{\overline{\Gamma}_1, v_a : \overline{\tau}\ \mathit{!}\ \alpha \vdash \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b) : \alpha}:_{app}}{\overline{\Gamma}_1 \vdash \lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b) : \gamma}:_{abs}}{\overline{\Gamma}, k : \overline{\sigma}\ \mathit{!}\ \alpha \vdash \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b)) : \alpha}:_{app}}{\overline{\Gamma} \vdash \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b)) : (\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha}:_{abs}$$

$\overline{\Gamma}_1 \vdash \mathtt{J}a\mathtt{K} : \gamma\ \mathit{!}\ \alpha = \mathrm{IH}_1$

$\overline{\Gamma}_2 \vdash \mathtt{J}b\mathtt{K} : \gamma = \mathrm{IH}_2$

$\gamma = (\overline{\tau}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$
$\overline{\Gamma}_1 = \overline{\Gamma}, k : \overline{\sigma}\ \mathit{!}\ \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma}_1, v_a : \overline{\tau}\ \mathit{!}\ \alpha$
$\overline{\Gamma}_3 = \overline{\Gamma}_2, v_b : \overline{\tau}$

Case $M = a\ b$**:** Assuming $\Gamma \vdash a\ b : \tau$ we have the induction hypotheses IH_1: $\overline{\Gamma} \vdash \mathtt{J}a\mathtt{K} : j\sigma\ \mathit{!}\ \tau j = \overline{\Gamma} \vdash \mathtt{J}a\mathtt{K} : ((\overline{\sigma}\ \mathit{!}\ (\overline{\tau}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$ and IH_2: $\overline{\Gamma} \vdash \mathtt{J}b\mathtt{K} : (\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$. We need to show $\overline{\Gamma} \vdash \mathtt{J}a\ b\mathtt{K} : (\overline{\tau}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma}_2 \vdash \mathtt{J}b\mathtt{K} : (\overline{\sigma}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha = \mathrm{IH}_2 \quad (1)}{\overline{\Gamma}_1, v_a : \delta \vdash \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b\ k) : \alpha}:_{app}}{\overline{\Gamma}_1 \vdash \lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b\ k) : \gamma}:_{abs}}{\overline{\Gamma}, k : \overline{\tau}\ \mathit{!}\ \alpha \vdash \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b\ k)) : \alpha}:_{app}}{\overline{\Gamma} \vdash \lambda k.\ \mathtt{J}a\mathtt{K}\ (\lambda v_a.\ \mathtt{J}b\mathtt{K}\ (\lambda v_b.\ v_a\ v_b\ k)) : (\overline{\tau}\ \mathit{!}\ \alpha)\ \mathit{!}\ \alpha}:_{abs}$$

$\overline{\Gamma}_1 \vdash \mathtt{J}a\mathtt{K} : \gamma\ \mathit{!}\ \alpha = \mathrm{IH}_1$

$$\textcircled{1}: \dfrac{\dfrac{\dfrac{v_a : \delta \mathit{\ 2\ } \overline{\Gamma}_3}{\overline{\Gamma}_3 \vdash v_a : \delta}:_{var} \quad \dfrac{v_b : \delta \mathit{\ 2\ } \overline{\Gamma}_3}{\overline{\Gamma}_3 \vdash v_b : \overline{\sigma}}:_{var}}{\overline{\Gamma}_3 \vdash v_a\ v_b : (\overline{\tau} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha}:_{app} \quad \dfrac{k : \overline{\tau} \mathit{\ !\ } \alpha \mathit{\ 2\ } \overline{\Gamma}_3}{\overline{\Gamma}_3 \vdash k : \overline{\tau} \mathit{\ !\ } \alpha}:_{var}}{\overline{\Gamma}_2, v_b : \overline{\sigma} \vdash v_a\ v_b\ k : \alpha}:_{app}}{\overline{\Gamma}_2 \vdash \lambda v_b.\ v_a\ v_b\ k : \overline{\sigma} \mathit{\ !\ } \alpha}:_{abs}$$

$\gamma = (\overline{\sigma} \mathit{\ !\ } (\overline{\tau} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha$
$\delta = \overline{\sigma} \mathit{\ !\ } (\overline{\tau} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha$
$\overline{\Gamma}_1 = \overline{\Gamma}, k : \overline{\tau} \mathit{\ !\ } \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma}_1, v_a : \delta$
$\overline{\Gamma}_3 = \overline{\Gamma}_2, v_b : \overline{\sigma}$

Case $M = \mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b$**:** Assuming $\Gamma \vdash \mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b : \sigma$ we have the induction hypotheses IH_1: $\overline{\Gamma} \vdash \llbracket c \rrbracket : (\mathsf{bool} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha$, IH_2: $\overline{\Gamma} \vdash \llbracket a \rrbracket : (\overline{\sigma} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha$ and IH_3: $\overline{\Gamma} \vdash \llbracket b \rrbracket : (\overline{\sigma} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha$. We need to show $\overline{\Gamma} \vdash \llbracket \mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b \rrbracket : (\overline{\sigma} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha$.

$$\dfrac{\overline{\Gamma}_1 \vdash \llbracket c \rrbracket : (\mathsf{bool} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha = \mathrm{IH}_1 \quad \dfrac{\dfrac{\dfrac{v_c : \mathsf{bool} \mathit{\ 2\ } \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash v_c : \mathsf{bool}}:_{var} \quad \textcircled{1} \quad \textcircled{2}}{\overline{\Gamma}_1, v_c : \mathsf{bool} \vdash \mathsf{if}\ v_c\ \mathsf{then}\ \llbracket a \rrbracket\ k\ \mathsf{else}\ \llbracket b \rrbracket\ k : \alpha}:_{if}}{\overline{\Gamma}_1 \vdash \lambda v_c.\ \mathsf{if}\ v_c\ \mathsf{then}\ \llbracket a \rrbracket\ k\ \mathsf{else}\ \llbracket b \rrbracket\ k : \mathsf{bool} \mathit{\ !\ } \alpha}:_{abs}}{\overline{\Gamma}, k : \overline{\sigma} \mathit{\ !\ } \alpha \vdash \llbracket c \rrbracket\ (\lambda v_c.\ \mathsf{if}\ v_c\ \mathsf{then}\ \llbracket a \rrbracket\ k\ \mathsf{else}\ \llbracket b \rrbracket\ k) : \alpha}:_{app}}{\overline{\Gamma} \vdash \lambda k.\ \llbracket c \rrbracket\ (\lambda v_c.\ \mathsf{if}\ v_c\ \mathsf{then}\ \llbracket a \rrbracket\ k\ \mathsf{else}\ \llbracket b \rrbracket\ k) : (\overline{\sigma} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha}:_{abs}$$

$$\textcircled{1}: \dfrac{\overline{\Gamma}_2 \vdash \llbracket a \rrbracket : (\overline{\sigma} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha = \mathrm{IH}_2 \quad \dfrac{k : \overline{\sigma} \mathit{\ !\ } \alpha \mathit{\ 2\ } \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash k : \overline{\sigma} \mathit{\ !\ } \alpha}:_{var}}{\overline{\Gamma}_2 \vdash \llbracket a \rrbracket\ k : \alpha}:_{app}$$

$$\textcircled{2}: \dfrac{\overline{\Gamma}_2 \vdash \llbracket b \rrbracket : (\overline{\sigma} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha = \mathrm{IH}_3 \quad \dfrac{k : \overline{\sigma} \mathit{\ !\ } \alpha \mathit{\ 2\ } \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash k : \overline{\sigma} \mathit{\ !\ } \alpha}:_{var}}{\overline{\Gamma}_2 \vdash \llbracket b \rrbracket\ k : \alpha}:_{app}$$

$\overline{\Gamma}_1 = \overline{\Gamma}, k : \overline{\sigma} \mathit{\ !\ } \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma}_1, v_c : \mathsf{bool}$

Case $M = \mathsf{let}\ x = v\ \mathsf{in}\ a$: Assuming $\Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ a : \tau$ we have the induction hypotheses $\mathrm{IH}_1 : \overline{\Gamma} \vdash \jmath v \jmath : \forall\beta.\overline{\sigma}$, $\mathrm{IH}_2 : \overline{\Gamma}, x : \forall\beta.\overline{\sigma} \vdash \llbracket a \rrbracket : (\overline{\tau} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha$ and $\beta \mathit{\ \#\ } \overline{\Gamma}$. We need to show $\overline{\Gamma} \vdash \llbracket \mathsf{let}\ x = v\ \mathsf{in}\ a \rrbracket : (\overline{\tau} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha$.

$$\dfrac{\overline{\Gamma}_1 \vdash \jmath v \jmath : \forall\beta.\overline{\sigma} = \mathrm{IH}_1 \quad \dfrac{\overline{\Gamma}_2 \vdash \llbracket a \rrbracket : (\overline{\tau} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha = \mathrm{IH}_2 \quad \dfrac{k : \overline{\tau} \mathit{\ !\ } \alpha \mathit{\ 2\ } \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash k : \overline{\tau} \mathit{\ !\ } \alpha}:_{var}}{\overline{\Gamma}_1, x : \forall\beta.\overline{\sigma} \vdash \llbracket a \rrbracket\ k : \alpha}:_{app}}{\overline{\Gamma}, k : \overline{\tau} \mathit{\ !\ } \alpha \vdash \mathsf{let}\ x = \jmath v \jmath\ \mathsf{in}\ \llbracket a \rrbracket\ k : \alpha}:_{let\_poly} \text{ and } \beta \mathit{\ \#\ } \overline{\Gamma}_1$$
$$\overline{\Gamma} \vdash \lambda k.\ \mathsf{let}\ x = \jmath v \jmath\ \mathsf{in}\ \llbracket a \rrbracket\ k : (\overline{\tau} \mathit{\ !\ } \alpha) \mathit{\ !\ } \alpha :_{abs}$$

$\overline{\Gamma}_1 = \overline{\Gamma}, k : \overline{\tau} \mathit{\ !\ } \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma_1}, x : \forall\beta.\overline{\sigma}$

Case $M = \mathsf{match}\ l\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ e\ \mathit{j}\ \mathsf{Cons}(a,b)\ \ !\ \ f$: Assuming $\Gamma \vdash \mathsf{match}\ l\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ e\ \mathit{j}\ \mathsf{Cons}(a,b)\ \ !\ \ f : \sigma$ we have the induction hypotheses IH_1: $\overline{\Gamma} \vdash \mathtt{J}l\mathtt{K} : (\mathsf{list}\ \ !\ \ \alpha)\ \ !\ \ \alpha$, IH_2: $\overline{\Gamma} \vdash \mathtt{J}e\mathtt{K} : (\overline{\sigma}\ \ !\ \ \alpha)\ \ !\ \ \alpha$ and IH_3: $\overline{\Gamma}, a : \mathsf{nat}, b : \mathsf{list} \vdash \mathtt{J}f\mathtt{K} : (\overline{\sigma}\ \ !\ \ \alpha)\ \ !\ \ \alpha$. We need to show $\overline{\Gamma} \vdash \mathsf{Jmatch}\ l\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ e\ \mathit{j}\ \mathsf{Cons}(a,b)\ \ !\ \ f\mathtt{K} : (\overline{\sigma}\ \ !\ \ \alpha)\ \ !\ \ \alpha$.

$$\dfrac{\dfrac{v_l : \mathsf{list}\ \mathit{2}\ \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash v_l : \mathsf{list}}\ :_{var} \qquad \dfrac{\dfrac{\dfrac{v_l : \mathsf{list}\ \mathit{2}\ \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash v_l : \mathsf{list}}\ :_{var} \qquad \textcircled{1} \qquad \textcircled{2}}{\overline{\Gamma}_1, v_l : \mathsf{list} \vdash \mathsf{match}\ v_l\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ \mathtt{J}e\mathtt{K}\ k\ \mathit{j}\ \mathsf{Cons}(a,b)\ \ !\ \ \mathtt{J}f\mathtt{K}\ k : \alpha}\ :_{match}}{\overline{\Gamma}_1 \vdash \lambda v_l.\ \mathsf{match}\ v_l\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ \mathtt{J}e\mathtt{K}\ k\ \mathit{j}\ \mathsf{Cons}(a,b)\ \ !\ \ \mathtt{J}f\mathtt{K}\ k : \mathsf{list}\ \ !\ \ \alpha}\ :_{abs}}{\overline{\Gamma}, k : \overline{\sigma}\ \ !\ \ \alpha \vdash \mathtt{J}l\mathtt{K}\ (\lambda v_l.\ \mathsf{match}\ v_l\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ \mathtt{J}e\mathtt{K}\ k\ \mathit{j}\ \mathsf{Cons}(a,b)\ \ !\ \ \mathtt{J}f\mathtt{K}\ k) : \alpha}\ :_{app}}{\overline{\Gamma} \vdash \lambda k.\ \mathtt{J}l\mathtt{K}\ (\lambda v_l.\ \mathsf{match}\ v_l\ \mathsf{with}\ \mathsf{Nil}\ \ !\ \ \mathtt{J}e\mathtt{K}\ k\ \mathit{j}\ \mathsf{Cons}(a,b)\ \ !\ \ \mathtt{J}f\mathtt{K}\ k) : (\overline{\sigma}\ \ !\ \ \alpha)\ \ !\ \ \alpha}\ :_{abs}$$

$$\textcircled{1}: \quad \dfrac{\overline{\Gamma}_2 \vdash \mathtt{J}e\mathtt{K} : (\overline{\sigma}\ \ !\ \ \alpha)\ \ !\ \ \alpha = \mathrm{IH}_2 \qquad \dfrac{k : \overline{\sigma}\ \ !\ \ \alpha\ \mathit{2}\ \overline{\Gamma}_2}{\overline{\Gamma}_2 \vdash k : \overline{\sigma}\ \ !\ \ \alpha}\ :_{var}}{\overline{\Gamma}_2 \vdash \mathtt{J}e\mathtt{K}\ k : \alpha}\ :_{app}$$

$$\textcircled{2}: \quad \dfrac{\overline{\Gamma}_3 \vdash \mathtt{J}f\mathtt{K} : (\overline{\sigma}\ \ !\ \ \alpha)\ \ !\ \ \alpha = \mathrm{IH}_3 \qquad \dfrac{k : \overline{\sigma}\ \ !\ \ \alpha\ \mathit{2}\ \overline{\Gamma}_3}{\overline{\Gamma}_3 \vdash k : \overline{\sigma}\ \ !\ \ \alpha}\ :_{var}}{\overline{\Gamma}_2, a : \mathsf{nat}, b : \mathsf{list} \vdash \mathtt{J}f\mathtt{K}\ k : \alpha}\ :_{app}$$

$\overline{\Gamma}_1 = \overline{\Gamma}, k : \overline{\sigma}\ \ !\ \ \alpha$
$\overline{\Gamma}_2 = \overline{\Gamma}_1, v_l : \mathsf{list}$
$\overline{\Gamma}_3 = \overline{\Gamma}_2, a : \mathsf{nat}, b : \mathsf{list}$

□

A.7 Proof of Lemma 3

Lemma 3 (Extended colon translation)**.** *If K is a closed value then* $\mathtt{J}a\mathtt{K}\ K\ !\ a : K$.

Proof. The proof is by induction on a and a case analysis of a.

Case $a = v$:

$$\begin{aligned}\mathtt{J}v\mathtt{K}\ K &= (\lambda k.\ k\ \mathit{jvj})\ K \\ &!\ K\ \mathit{jvj} \\ &= v : K\end{aligned}$$

Case $a = x$:

$$\begin{aligned}\mathtt{J}x\mathtt{K}\ K &= (\lambda k.\ k\ x)\ K \\ &!\ K\ x \\ &= x : K\end{aligned}$$

Case $a = v_1\ v_2$:

$$\begin{aligned}\mathtt{J}v_1\ v_2\mathtt{K}\ K &= (\lambda k.\ \mathtt{J}v_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}v_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ k)))\ K \\ &!\ \mathtt{J}v_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}v_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \\ &!\ v_1 : (\lambda x_1.\ \mathtt{J}v_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \quad \text{by IH of } v_1 \\ &= (\lambda x_1.\ \mathtt{J}v_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K))\ \mathit{jv_1j} \\ &!\ \mathtt{J}v_2\mathtt{K}\ (\lambda x_2.\ \mathit{jv_1j}\ x_2\ K) \\ &!\ v_2 : (\lambda x_2.\ \mathit{jv_1j}\ x_2\ K) \quad \text{by IH of } v_2 \\ &= (\lambda x_2.\ \mathit{jv_1j}\ x_2\ K)\ \mathit{jv_2j} \\ &!\ \mathit{jv_1j}\ \mathit{jv_2j}\ K \\ &= \mathit{jv_1j}\ \mathit{jv_2j} : K\end{aligned}$$

Case $a = v\ b$:

$$\begin{aligned}\mathtt{J}v\ b\mathtt{K}\ K &= (\lambda k.\ \mathtt{J}v\mathtt{K}\ (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ k)))\ K \\ &!\ \mathtt{J}v\mathtt{K}\ (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \\ &!\ v : (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \quad \text{by IH of } v \\ &= (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K))\ \mathit{jvj} \\ &!\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ \mathit{jvj}\ x_2\ K) \\ &!\ b : (\lambda x_2.\ \mathit{jvj}\ x_2\ K) \quad \text{by IH } b \\ &= v\ b : K\end{aligned}$$

Case $a = a_1\ a_2$:

$$\begin{aligned}\mathtt{J}a_1\ a_2\mathtt{K}\ K &= (\lambda k.\ \mathtt{J}a_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}b\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ k)))\ K \\ &!\ \mathtt{J}a_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \\ &!\ a_1 : (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2\ K)) \quad \text{by IH of } a_1 \\ &= a_1\ a_2 : K\end{aligned}$$

Case $a = \mathsf{Some}\ a_1$:

$$\begin{aligned}
\mathtt{J}\mathsf{Some}\ a_1\mathtt{K}\ K &= (\lambda k.\ \mathtt{J}a_1\mathtt{K}\ (\lambda v_a.\ k\ \mathsf{Some}\ v_a))\ K \\
&\ !\ \ \mathtt{J}a_1\mathtt{K}\ (\lambda v_a.\ K\ \mathsf{Some}\ v_a) \\
&\ !\ \ a_1 : (\lambda v_a.\ K\ \mathsf{Some}\ v_a) \quad \text{by IH of } a_1 \\
&= \mathsf{Some}\ a_1 : K
\end{aligned}$$

The proofs for the cases $\mathsf{zero?}\ a$ and $\mathsf{Succ}(a)$ are similar.

Case $a = \mathsf{Cons}(a_1, a_2)$:

$$\begin{aligned}
\mathtt{J}\mathsf{Cons}(a_1, a_2)\mathtt{K}\ K &= (\lambda k.\ \mathtt{J}a_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ k\ \mathsf{Cons}(x_1, x_2))))\ K \\
&\ !\ \ \mathtt{J}a_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ K\ \mathsf{Cons}(x_1, x_2))) \\
&\ !\ \ a_1 : (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ K\ \mathsf{Cons}(x_1, x_2))) \quad \text{by IH of } a_1 \\
&= \mathsf{Cons}(a_1, a_2) : K
\end{aligned}$$

The proofs for the cases $a + b$ and $a\ ;\ b$ are similar.

Case $a = \mathsf{let}\ x = v\ \mathsf{in}\ b$:

$$\begin{aligned}
\mathtt{J}\mathsf{let}\ x = v\ \mathsf{in}\ b\mathtt{K}\ K &= (\lambda k.\mathsf{let}\ x = \mathit{jvj}\ \mathsf{in}\mathtt{J}b\mathtt{K}\ k)\ K \\
&\ !\ \ \mathsf{let}\ x = \mathit{jvj}\ \mathsf{in}\ \mathtt{J}b\mathtt{K}\ K \\
&= \mathsf{let}\ x = \mathit{jvj}\ \mathsf{in}\ \mathtt{J}b\mathtt{K} : K
\end{aligned}$$

Case $a = \mathsf{if}\ c\ \mathsf{then}\ a_1\ \mathsf{else}\ a_2$:

$$\begin{aligned}
\mathtt{J}\mathsf{if}\ c\ \mathsf{then}\ a_1\ \mathsf{else}\ a_2\mathtt{K}\ K &= (\lambda k.\ \mathtt{J}c\mathtt{K}\ (\lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathtt{J}a_1\mathtt{K}\ k\ \mathsf{else}\ \mathtt{J}a_2\mathtt{K}\ k))\ K \\
&\ !\ \ \mathtt{J}c\mathtt{K}\ (\lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathtt{J}a_1\mathtt{K}\ K\ \mathsf{else}\ \mathtt{J}a_2\mathtt{K}\ K) \\
&\ !\ \ c : (\lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathtt{J}a_1\mathtt{K}\ K\ \mathsf{else}\ \mathtt{J}a_2\mathtt{K}\ K) \quad \text{by IH of } c \\
&= \mathsf{if}\ c\ \mathsf{then}\ a_1\ \mathsf{else}\ a_2 : K
\end{aligned}$$

The proof for the match case is similar.

Case $a = \mathsf{callcc}\ a_1$:

$$\begin{aligned}
\mathtt{J}\mathsf{callcc}\ a_1\mathtt{K}\ K &= (\lambda k.\ \mathtt{J}a_1\mathtt{K}\ (\lambda f.\ f\ k\ k))\ K \\
&\ !\ \ \mathtt{J}a_1\mathtt{K}\ (\lambda f.\ f\ K\ K) \\
&\ !\ \ a_1 : (\lambda f.\ f\ K\ K) \quad \text{by IH of } a_1 \\
&= \mathsf{callcc}\ a_1 : K
\end{aligned}$$

Case $a = \mathsf{throw}\ a_1\ a_2$:

$$\begin{aligned}
\mathtt{J}\mathsf{throw}\ a_1\ a_2\mathtt{K} &= (\lambda k.\ \mathtt{J}a_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2)))\ K \\
&\ !\ \ \mathtt{J}a_1\mathtt{K}\ (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2)) \\
&\ !\ \ a_1 : (\lambda x_1.\ \mathtt{J}a_2\mathtt{K}\ (\lambda x_2.\ x_1\ x_2)) \quad \text{by IH of } a_1 \\
&= \mathsf{throw}\ a_1\ a_2 : K
\end{aligned}$$

□

A.8 CPS translation of $find\_one$

In this section we will translate the program $find\_one$ into continuation-passing style using the colon translation defined in Definition 4.2.4.

$$
\begin{aligned}
find\_one = \ & \mathsf{let\ list\_iter} = \mu list\_iter.\lambda f.\lambda list.\ \mathsf{match}\ list\ \mathsf{with\ Nil}\ \ !\ \ () \\
& \qquad\qquad \mathsf{j\ Cons}(hd, tl)\ \ !\ \ f\ hd\ ;\ list\_iter\ f\ tl \\
& \mathsf{in\ (let\ find} = \lambda p.\lambda l.\ \mathsf{callcc}\ (\lambda k.\ \mathsf{list\_iter}\ (\lambda x.\ \mathsf{if}\ p\ x\ \mathsf{then} \\
& \qquad\qquad \mathsf{throw}\ k\ (\mathsf{Some}\ x)\ \mathsf{else}\ ())\ l;\ \mathsf{None}) \\
& \mathsf{in\ find}\ (\lambda x.\mathsf{zero?\ Pred}\ x)\ \mathsf{Cons(Succ(Succ(0)),\ Cons(Succ(0),\ Nil)))}
\end{aligned}
$$

First we give the translation into continuation-passing style using the CPS translation from Definition 4.2.1 for the list_iter program and find program. We will then give the complete colon translation of the program $find\_one$.

We can see that the translated program evaluates to Some Succ(0) just as the untranslated program in Appendix A.4. Furthermore we see a correspondence between the first reduction in both the translated and the untranslated program. In both programs the let is reduced first and the list_iter program substituted for the variable $list\_iter$ in find.

The list_iter and find program are the let-bound expressions. They are values and have to be translated using the value translation.

$$
\begin{aligned}
&\textit{j}\mathsf{list\_iter}\textit{j}\\
=&\left|\mu list\_iter.\lambda f.\lambda list.\mathsf{match}\ list\ \mathsf{with}\ \mathsf{Nil}\ \Rightarrow\ ()\ \mathsf{j}\ \mathsf{Cons}(hd,tl)\ \Rightarrow\ f\ hd\ ;\ list\_iter\ f\ tl\right|\\
=&\mu list\_iter.\lambda f.\ (\lambda k_{14}.\ k_{14}\ (\lambda list.\ (\lambda k_{15}.\ (\lambda k_{16}.\ k_{16}\ list)\ (\lambda v_{11}.\ \mathsf{match}\ v_{11}\ \mathsf{with}\ \mathsf{Nil}\ \Rightarrow\ (\lambda k_{17}.\ k_{17}\ ())\ k_{15}\ \mathsf{j}\ \mathsf{Cons}(hd,tl)\ \Rightarrow\ (\lambda k_{18}.\ (\lambda k_{19}.\ (\lambda k_{20}.\\
&k_{20}\ f)\ (\lambda v_{14}.\ (\lambda k_{21}.\ k_{21}\ hd)\ (\lambda v_{15}.\ v_{14}\ v_{15}\ k_{19})))\ (\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ list\_iter)\ (\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ f)\ (\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))\ (\lambda v_{16}.\\
&(\lambda k_{23}.\ k_{23}\ tl)\ (\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))\ (\lambda v_{13}.\ k_{18}\ v_{13})))\ k_{15}))))
\end{aligned}
$$

$$
\begin{aligned}
&\textit{j}\mathsf{find}\textit{j}\\
=&\left|\lambda p.\lambda l.\ \mathsf{callcc}\ (\lambda k\ .\ \mathsf{list\_iter}\ (\lambda x.\ \mathsf{if}\ p\ x\ \mathsf{then\ throw}\ k\ \ (\mathsf{Some}\ x)\ \mathsf{else}\ ())\ l\ ;\ \mathsf{None})\right|\\
=&\lambda p.\ (\lambda k_5.\ k_5\ (\lambda l.\ (\lambda k_6.\ (\lambda k_7.\ k_7\ (\lambda k\ .\ (\lambda k_8.\ (\lambda k_{10}.\ (\lambda k_{12}.\ (\lambda k_{13}.\ k_{13}\ \mathsf{list\_iter})(\lambda v_9.\ (\lambda k_{27}.\ k_{27}\ (\lambda x.\ (\lambda k_{28}.\ (\lambda k_{30}.\ (\lambda k_{31}.\ k_{31}\ p)(\lambda v_{21}.\ (\lambda k_{31a}.\\
&k_{31a}\ x)(\lambda v_{21a}.\ v_{21}\ v_{21a}\ k_{30})))(\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ k\ )(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ x)\ (\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))\ (\lambda v_{23}.\ v_{22}\ v_{23})))\ k_{28}\\
&\mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())\ k_{28}))))\ (\lambda v_{10}.\ v_9\ v_{10}\ k_{12})))\ (\lambda v_7.\ (\lambda k_{11}.\ k_{11}\ l)\ (\lambda v_8.\ v_7\ v_8\ k_{10})))\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})\ (\lambda v_6.\ k_8\ v_6)))))\ (\lambda f.\ f\ k_6\ k_6))))
\end{aligned}
$$

We will now give the colon translation for the complete $find\_one$ program. The initial continuation will be the identity function $(\lambda i.i)$. For readability we will abbreviate the list_iter and find program with iterPrg and findPrg respectively. Therefore *j*list_iter*j* is the same as *j*iterPrg*j* and *j*find*j* is the same as *j*findPrg*j*.

$$
\begin{aligned}
&\mathsf{let\ list\_iter} = \mathsf{iterPrg\ in\ (let\ find} = \mathsf{findPrg\ in\ find}\ (\lambda p.\mathsf{zero?\ Pred}\ p)\ \mathsf{Cons(Succ(Succ(0)),Cons(Succ(0),Nil)))} : (\lambda i.i)\\
=&\mathsf{let\ list\_iter} = \textit{j}\mathsf{iterPrg}\textit{j}\ \mathsf{in\ J(let\ find} = \mathsf{findPrg\ in\ find}\ (\lambda p.\mathsf{zero?\ Pred}\ p)\ \mathsf{Cons(Succ(Succ(0)),Cons(Succ(0),Nil)))K}\ (\lambda i.i)\\
=&\mathsf{let\ list\_iter} = \textit{j}\mathsf{iterPrg}\textit{j}\ \mathsf{in}\ (\lambda k_a.\ \mathsf{let\ find} = \textit{j}\mathsf{findPrg}\textit{j}\ \mathsf{in\ Jfind}\ (\lambda p.\mathsf{zero?\ Pred}\ p)\ \mathsf{Cons(Succ(Succ(0)),Cons(Succ(0),Nil))K}\ k_a)\ (\lambda i.i)\\
=&\mathsf{let\ list\_iter} = \textit{j}\mathsf{iterPrg}\textit{j}\ \mathsf{in}\ (\lambda k_a.\ \mathsf{let\ find} = \textit{j}\mathsf{findPrg}\textit{j}\ \mathsf{in}\ (\lambda k_b.\ (\lambda k_d.\ (\lambda k_e.\ k_4\ \mathsf{find})\ (\lambda v_a.\ (\lambda k_c.\ k_c\ (\lambda p.\ (\lambda k_2.\ (\lambda k_3.\ (\lambda k_4.\ k_4\ p)\ (\lambda v_4.\ k_3\ \mathsf{Pred}\ v_4))\\
&(\lambda v_3.\ k_2\ \mathsf{zero?}\ v_3))))\ (\lambda v_b.\ v_a\ v_b\ k_d)))\ (\lambda v_1.\ (\lambda k_1.\ k_1\ \mathsf{Cons(Succ(Succ(0)),Cons(Succ(0),Nil)))}\ (\lambda v_2.\ v_1\ v_2\ k_b)))\ k_a)\ (\lambda i.i)
\end{aligned}
$$

After having translated the whole program we begin reducing. Because of the large terms we will only show some intermediate terms and denote multiple reductions with $\Rightarrow$. We can see that due to the colon translation the first reduction corresponds to the first reduction in the untranslated program where the let-expression is reduced and the list_iter program is substituted for the variable

in the find program.

E[let list_iter =*j*iterPrg*j* in (λk_a. let find =*j*findPrg*j* in (λk_b. (λk_d. (λk_e. k_4 find) (λv_a. (λk_c. k_c (λp. (λk_2. (λk_3. (λk_4. k_4 p) (λv_4. k_3 Pred v_4)) (λv_3. k_2 zero? v_3)))) (λv_b. v_a v_b k_d))) (λv_1. (λk_1. k_1 Cons(Succ(Succ(0)), Cons(Succ(0), Nil))) (λv_2. v_1 v_2 k_b))) k_a) ($\lambda i.i$)]
with $E = [\,]$

! E[(λk_a. let find = (λp. (λk_5. k_5 (λl. (λk_6. (λk_7. k_7 (λk . (λk_8. (λk_{10}. (λk_{12}. (λk_{13}. k_{13} ($\mu list\_iter.\lambda f$. (λk_{14}. k_{14} ($\lambda list$. (λk_{15}. (λk_{16}. k_{16} $list$) (λv_{11}. match v_{11} with Nil ! (λk_{17}. k_{17} ()) k_{15} *j* Cons(hd, tl) ! (λk_{18}. (λk_{19}. (λk_{20}. k_{20} f)(λv_{14}. (λk_{21}. k_{21} hd)(λv_{15}. v_{14} v_{15} k_{19}))) (λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} $list\_iter$)(λv_{18}. (λk_{26}. k_{26} f)(λv_{19}. v_{18} v_{19} k_{24})))(λv_{16}. (λk_{23}. k_{23} tl)(λv_{17}. v_{16} v_{17} k_{22})))(λv_{13}. k_{18} v_{13}))) k_{15}))))))(λv_9. (λk_{27}. k_{27} (λx. (λk_{28}. (λk_{30}. (λk_{31}. k_{31} p)(λv_{21}. (λk_{31a}. k_{31a} x)(λv_{21a}. v_{21} v_{21a} k_{30})))(λv_{20}. if v_{20} then (λk_{31}. (λk_{32}. k_{32} k) (λv_{22}. (λk_{33}. (λk_{34}. k_{34} x) (λv_{24}. k_{33} (Some v_{24}))) (λv_{23}. v_{22} v_{23}))) k_{28} else (λk_{29}. k_{29} ()) k_{28})))) (λv_{10}. v_9 v_{10} k_{12}))) (λv_7. (λk_{11}. k_{11} l) (λv_8. v_7 v_8 k_{10}))) (λv_5. (λk_9. k_9 None) (λv_6. k_8 v_6))))) (λf. f k_6 k_6))))) in (λk_b. (λk_d. (λk_e. k_4 find) (λv_a. (λk_c. k_c (λp. (λk_2. (λk_3. (λk_4. k_4 p) (λv_4. k_3 Pred v_4)) (λv_3. k_2 zero? v_3)))) (λv_b. v_a v_b k_d))) (λv_1. (λk_1. k_1 Cons(Succ(Succ(0)), Cons(Succ(0), Nil))) (λv_2. v_1 v_2 k_b))) k_a) ($\lambda i.i$)]
with $E = [\,]$

! E[(λp. (λk_5. k_5 (λl. (λk_6. (λk_7. k_7 (λk . (λk_8. (λk_{10}. (λk_{12}. (λk_{13}. k_{13} ($\mu list\_iter.\lambda f$.(λk_{14}. k_{14} ($\lambda list$. (λk_{15}. (λk_{16}. k_{16} $list$) (λv_{11}. match v_{11} with Nil ! (λk_{17}. k_{17} ()) k_{15} *j* Cons(hd, tl) ! (λk_{18}. (λk_{19}. (λk_{20}. k_{20} f) (λv_{14}. (λk_{21}. k_{21} hd) (λv_{15}. v_{14} v_{15} k_{19}))) (λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} $list\_iter$) (λv_{18}. (λk_{26}. k_{26} f) (λv_{19}. v_{18} v_{19} k_{24}))) (λv_{16}. (λk_{23}. k_{23} tl) (λv_{17}. v_{16} v_{17} k_{22}))) (λv_{13}. k_{18} v_{13}))) k_{15}))))))(λv_9. (λk_{27}. k_{27} (λx. (λk_{28}. (λk_{30}. (λk_{31}. k_{31} p) (λv_{21}. (λk_{31a}. k_{31a} x)(λv_{21a}. v_{21} v_{21a} k_{30}))) (λv_{20}. if v_{20} then (λk_{31}. (λk_{32}. k_{32} k) (λv_{22}. (λk_{33}. (λk_{34}. k_{34} x) (λv_{24}. k_{33} (Some v_{24}))) (λv_{23}. v_{22} v_{23}))) k_{28} else (λk_{29}. k_{29} ()) k_{28})))) (λv_{10}. v_9 v_{10} k_{12}))) (λv_7. (λk_{11}. k_{11} l) (λv_8. v_7 v_8 k_{10}))) (λv_5. (λk_9. k_9 None) (λv_6. k_8 v_6))))) (λf. f k_6 k_6))))) (λp. (λk_2. (λk_3. (λk_4. k_4 p) (λv_4. k_3 Pred v_4)) (λv_3. k_2 zero? v_3))) (λv_1. (λk_1. k_1 Cons(Succ(Succ(0)), Cons(Succ(0), Nil))) (λv_2. v_1 v_2 ($\lambda i.i$)))]
with $E = [\,]$

Multiple administrative reduction as well as the reduction of the let find = ... , which corresponds to the reduction in the untranslated program, yielded the term above. The term above is the colon translation of find (λp.zero? Pred p) Cons(Succ(Succ(0)),Cons(Succ(0), Nil))) : ($\lambda i.i$).

! E[(λk_5. k_5 (λl.(λk_6. (λk_7. k_7 (λk .(λk_8. (λk_{10}. (λk_{12}. (λk_{13}. k_{13} ($\mu list\_iter.\lambda f$.(λk_{14}. k_{14} ($\lambda list$.(λk_{15}. (λk_{16}. $k_{16}list$)(λv_{11}. match v_{11} with Nil ! (λk_{17}. k_{17} ()) k_{15} *j* Cons(hd, tl) ! (λk_{18}. (λk_{19}. (λk_{20}. k_{20} f)(λv_{14}. (λk_{21}. k_{21} hd)(λv_{15}. v_{14} v_{15} k_{19})))(λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} $list\_iter$) (λv_{18}. (λk_{26}. k_{26} f)(λv_{19}. v_{18} v_{19} k_{24})))(λv_{16}. (λk_{23}. k_{23} tl)(λv_{17}. v_{16} v_{17} k_{22})))(λv_{13}. k_{18} v_{13}))) k_{15}))))))(λv_9. (λk_{27}. k_{27} (λx.(λk_{28}. (λk_{30}. (λk_{31}. k_{31} (λp. (λk_2. (λk_3. (λk_4. k_4 p)(λv_4. k_3 Pred v_4))(λv_3. k_2 zero? v_3))))(λv_{21}. (λk_{31a}. k_{31a} x)(λv_{21a}. v_{21} v_{21a}

k_{30})))(λv_{20}. if v_{20} then (λk_{31}. (λk_{32}. k_{32} k)(λv_{22}. (λk_{33}. (λk_{34}. k_{34} x)(λv_{24}. k_{33} (Some v_{24})))(λv_{23}. v_{22} v_{23}))) k_{28} else (λk_{29}. k_{29} ()) k_{28}))))(λv_{10}. v_9 v_{10} k_{12})))(λv_7. (λk_{11}. k_{11} l)(λv_8. v_7 v_8 k_{10})))(λv_5. (λk_9. k_9 None)(λv_6. k_8 v_6)))))(λf. f k_6 k_6))))(λv_1. (λk_1. k_1 Cons(Succ(Succ(0)), Cons(Succ(0), Nil)))(λv_2. v_1 v_2 ($\lambda i.i$)))]
with $E = [\,]$

! E[($\mu list\_iter.\lambda f.$(λk_{14}. k_{14} ($\lambda list$. (λk_{15}. (λk_{16}. k_{16} $list$)(λv_{11}. match v_{11} with Nil ! (λk_{17}. k_{17} ()) k_{15} *j* Cons(hd, tl) ! (λk_{18}. (λk_{19}. (λk_{20}. k_{20} f) (λv_{14}. (λk_{21}. k_{21} hd)(λv_{15}. v_{14} v_{15} k_{19})))(λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} $list\_iter$)(λv_{18}. (λk_{26}. k_{26} f)(λv_{19}. v_{18} v_{19} k_{24})))(λv_{16}. (λk_{23}. k_{23} tl) (λv_{17}. v_{16} v_{17} k_{22})))(λv_{13}. k_{18} v_{13}))) k_{15})))))(λx.(λk_{28}. (λk_{30}. (λk_{31}. k_{31} (λp.(λk_2. (λk_3. (λk_4. k_4 p)(λv_4. k_3 Pred v_4))(λv_3. k_2 zero? v_3))))(λv_{21}. (λk_{31a}. k_{31a} x)(λv_{21a}. v_{21} v_{21a} k_{30})))(λv_{20}. if v_{20} then (λk_{31}. (λk_{32}. k_{32} ($\lambda i.i$))(λv_{22}. (λk_{33}. (λk_{34}. k_{34} x)(λv_{24}. k_{33} (Some v_{24}))) (λv_{23}. v_{22} v_{23}))) k_{28} else (λk_{29}. k_{29} ()) k_{28})))(λv_7. (λk_{11}. k_{11} Cons(Succ(Succ(0)), Cons(Succ(0), Nil)))(λv_8. v_7 v_8 (λv_5. (λk_9. k_9 None)(λv_6. ($\lambda i.i$) v_6))))]
with $E = [\,]$

! E[match Cons(Succ(Succ(0)), Cons(Succ(0), Nil)) with Nil ! (λk_{17}. k_{17} ())(λv_5. (λk_9. k_9 None)(λv_6. ($\lambda i.i$) v_6)) *j* Cons(hd, tl) ! (λk_{18}. (λk_{19}. (λk_{20}. k_{20} check) (λv_{14}. (λk_{21}. k_{21} hd)(λv_{15}. v_{14} v_{15} k_{19})))(λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} saved_iter)(λv_{18}. (λk_{26}. k_{26} check)(λv_{19}. v_{18} v_{19} k_{24}))) (λv_{16}. (λk_{23}. k_{23} tl)(λv_{17}. v_{16} v_{17} k_{22})))(λv_{13}. k_{18} v_{13})))(λv_5. (λk_9. k_9 None)(λv_6. ($\lambda i.i$) v_6))]
with $E = [\,]$

! E[check Succ(Succ(0)) (λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} saved_iter)(λv_{18}. (λk_{26}. k_{26} check)(λv_{19}. v_{18} v_{19} k_{24})))(λv_{16}. (λk_{23}. k_{23} Cons(Succ(0), Nil))(λv_{17}. v_{16} v_{17} k_{22})))(λv_{13}. (λv_5. (λk_9. k_9 None)(λv_6. ($\lambda i.i$) v_6)) v_{13}))]
with $E = [\,]$

! E[if false then (λk_{31}. (λk_{32}. k_{32} ($\lambda i.i$))(λv_{22}. (λk_{33}. (λk_{34}. k_{34} x)(λv_{24}. k_{33} (Some v_{24})))(λv_{23}. v_{22} v_{23})))(λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} saved_iter)(λv_{18}. (λk_{26}. k_{26} check)(λv_{19}. v_{18} v_{19} k_{24})))(λv_{16}. (λk_{23}. k_{23} Cons(Succ(0), Nil))(λv_{17}. v_{16} v_{17} k_{22})))(λv_{13}. (λv_5. (λk_9. k_9 None) (λv_6. ($\lambda i.i$) v_6)) v_{13})) else (λk_{29}. k_{29} ())(λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} saved_iter)(λv_{18}. (λk_{26}. k_{26} check) (λv_{19}. v_{18} v_{19} k_{24})))(λv_{16}. (λk_{23}. k_{23} Cons(Succ(0), Nil))(λv_{17}. v_{16} v_{17} k_{22})))(λv_{13}. (λv_5. (λk_9. k_9 None)(λv_6. ($\lambda i.i$) v_6)) v_{13}))]
with $E = [\,]$

! E[saved_iter check (λv_{16}. (λk_{23}. k_{23} Cons(Succ(0), Nil))(λv_{17}. v_{16} v_{17} (λv_{13}. (λv_5. (λk_9. k_9 None)(λv_6. ($\lambda i.i$) v_6)) v_{13})))]
with $E = [\,]$

! E[match Cons(Succ(0), Nil) with Nil ! (λk_{17}. k_{17} ())(λv_{13}. (λv_5. (λk_9. k_9 None)(λv_6. ($\lambda i.i$) v_6)) v_{13}) *j* Cons(hd, tl) ! (λk_{18}. (λk_{19}. (λk_{20}. k_{20} check)(λv_{14}. (λk_{21}. k_{21} hd)(λv_{15}. v_{14} v_{15} k_{19})))(λv_{12}. (λk_{22}. (λk_{24}. (λk_{25}. k_{25} saved_iter)(λv_{18}. (λk_{26}. k_{26} check)(λv_{19}. v_{18} v_{19} k_{24}))) (λv_{16}. (λk_{23}. k_{23} tl)(λv_{17}. v_{16} v_{17} k_{22})))(λv_{13}. k_{18} v_{13})))(λv_{13}. (λv_5. (λk_9. k_9 None)(λv_6. ($\lambda i.i$) v_6)) v_{13})]

with $E = [\,]$

! $E[\mathsf{check}\ \mathsf{Succ}(0)\ (\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))\ (\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))]$
with $E = [\,]$

! $E[(\lambda v_3.\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13})))\mathsf{zero?}\ v_3)\ \mathsf{Pred}\ \mathsf{Succ}(0)]$
with $E = [\,]$

$=E_1[\mathsf{Pred}\ \mathsf{Succ}(0)]$
with $E_1 = (\lambda v_3.\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13})))\mathsf{zero?}\ v_3)\ [\,]$

! $E_1[0]$
with $E_1 = (\lambda v_3.\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13})))\mathsf{zero?}\ v_3)\ [\,]$

$=E[(\lambda v_3.\ (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13})))\ \mathsf{zero?}\ v_3)\ 0]$
with $E = [\,]$

! $E[(\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13})))\ \mathsf{zero?}\ 0]$
with $E = [\,]$

$=E_2[\mathsf{zero?}\ 0]$

with $E_2 = (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ [\,]$

$=E_2[\mathsf{true}]$

with $E_2 = (\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ [\,]$

$=E[(\lambda v_{20}.\ \mathsf{if}\ v_{20}\ \mathsf{then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{true}]$

with $E = [\,]$

$!\ E[\mathsf{if\ true\ then}\ (\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))\ \mathsf{else}\ (\lambda k_{29}.\ k_{29}\ ())(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))]$

with $E = [\,]$

$!\ E[(\lambda k_{31}.\ (\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23})))(\lambda v_{12}.\ (\lambda k_{22}.\ (\lambda k_{24}.\ (\lambda k_{25}.\ k_{25}\ \mathsf{saved\_iter})(\lambda v_{18}.\ (\lambda k_{26}.\ k_{26}\ \mathsf{check})(\lambda v_{19}.\ v_{18}\ v_{19}\ k_{24})))(\lambda v_{16}.\ (\lambda k_{23}.\ k_{23}\ \mathsf{Nil})(\lambda v_{17}.\ v_{16}\ v_{17}\ k_{22})))(\lambda v_{13}.\ (\lambda v_{13}.\ (\lambda v_5.\ (\lambda k_9.\ k_9\ \mathsf{None})(\lambda v_6.\ (\lambda i.i)\ v_6))\ v_{13})\ v_{13}))]$

with $E = [\,]$

$!\ E[(\lambda k_{32}.\ k_{32}\ (\lambda i.i))(\lambda v_{22}.\ (\lambda k_{33}.\ (\lambda k_{34}.\ k_{34}\ \mathsf{Succ}(0))(\lambda v_{24}.\ k_{33}\ (\mathsf{Some}\ v_{24})))(\lambda v_{23}.\ v_{22}\ v_{23}))]$

with $E = [\,]$

$!\ E[(\lambda v_{23}.\ (\lambda i.i)\ v_{23})(\mathsf{Some\ Succ}(0))]\quad$ with $E = [\,]$

$!\ E[(\lambda i.i)\ (\mathsf{Some\ Succ}(0))]\quad$ with $E = [\,]$

$!\ E[\mathsf{Some\ Succ}(0)]\quad$ with $E = [\,]$