Productivity Verification for Functional Programs by Reduction to Termination Verification

Ren Fukaishi
The University of Tokyo
Tokyo, Japan

Naoki Kobayashi
The University of Tokyo
Tokyo, Japan

Ryosuke Sato
The University of Tokyo
Tokyo, Japan

Abstract

A program generating a co-inductive data structure is called *productive* if the program eventually generates all the elements of the data structure. We propose a new method for verifying the productivity, which transforms a co-inductive data structure into a function that takes a path as an argument and returns the corresponding element. For example, an infinite binary tree is converted to a function that takes a sequence consisting of 0 (left) and 1 (right), and returns the element in the specified position, and a stream is converted into a function that takes a sequence of the form 0^n (or, simply a natural number n) and returns the n-th element of the stream. A stream-generating program is then productive just if the function terminates for every n. The transformation allows us to reduce the productivity verification problem to the termination problem for call-by-name higher-order functional programs without co-inductive data structures. We formalize the transformation and prove its correctness. We have implemented an automated productivity checker based on the proposed method, by extending an automated validity checker for a higher-order fixpoint logic called HFL(Z), which can be used as a termination checker.

CCS Concepts: • **Theory of computation → Program verification**.

Keywords: Co-inductive types, Automated Program Verification, Higher-Order Functional Programs

1 Introduction

Functional programs often manipulate infinite data structures such as streams and infinite trees with co-inductive types. For example, functional reactive programming (FRP) [9] essentially requires streams so that one can define time-dependent values. A main concern about co-inductive data structures is productivity, which states that each element can be retrieved in finite time.

We propose an automated method for proving the productivity of co-inductive data structures. There are many size-based verification methods such as guarded recursion [4, 5, 8, 13, 22, 23] and sized types [15]. These methods are practically insufficient for proving the productivity of branching data structures such as trees. Let us consider a recursive definition of a binary tree, called "RightTree", as follows:

$$t = \mathrm{Node}(\mathrm{Right}(t), 1, t).$$

Here, "Node" is the constructor for (infinite) binary trees, which takes as an argument a triple consisting of the left child, the label of the node, and the right child. "Right" is the destructor that retrieves the right child node. RightTree is productive in the sense that every element of the tree is eventually calculated by unfolding the recursive definition. For example, the label of the left child is generated by the following reduction sequence.

$$\begin{aligned}&\mathrm{Node}(\mathrm{Right}(t), 1, t)\\ &\longrightarrow \mathrm{Node}(\mathrm{Right}(\mathrm{Node}(\mathrm{Right}(t), 1, t)), 1, t)\\ &\longrightarrow \mathrm{Node}(t, 1, t)\\ &\longrightarrow \mathrm{Node}(\mathrm{Node}(\mathrm{Right}(t), 1, t), 1, t)\end{aligned}$$

To our knowledge, however, existing size-based methods mentioned above do not allow such a definition. These methods require that a fixpoint definition of a co-inductive data structure be guarded by at least one constructor, or that the number of constructors should exceed the number of destructors. Such requirement is not satisfied by the definition of RightTree: the first occurrence of t is guarded by the destructor "Right" and the constructor "Node" just once.

We propose a new method for productivity verification, by a reduction to termination verification. To this end, we apply a program transformation to convert a co-inductive data structure into a function that takes a path and returns the corresponding element, so that a source program generating a co-inductive data structure is productive just if the target program is terminating for all the possible paths. For example, let us consider the following (productive) definition.

$$t = \mathrm{Node}(t, 1, t).$$

It generates an infinite binary tree consisting of 1's shown in Figure 1 (where [ ], [0], and [1] show the paths of the nodes). The definition can be converted to a function definition like the one shown in Figure 2. Here, we have used OCaml-like syntax and applied some simplification for the sake of simplicity; the actual code obtained by our transformation is slightly more complex, and should be evaluated in the call-by-name semantics. The function `node` corresponds to the constructor Node, and the definition of `tree` corresponds to the definition $t = \mathrm{Node}(t, 1, t)$. Note that the function `tree` is obviously terminating for all the inputs `ls`. Figure 3 shows the function definition obtained from RightTree. Here, the function `node` is as given before, `right` corresponds to

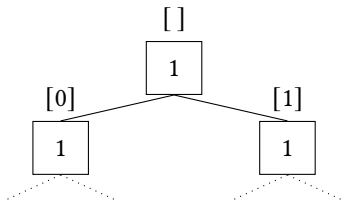


Figure 1. The tree defined by $t = \mathrm{Node}(t, 1, t)$.

```
let node (l, x, r) ls =
  match ls with
  | [] -> x
  | 0 :: ls' -> l ls'
  | 1 :: ls' -> r ls'
  | _ -> 0
let rec tree ls = node (tree, 1, tree) ls
```

Figure 2. The function definition obtained from the tree definition in Figure 1.

```
let right t ls = t (1::ls)
let rec righttree ls =
  node(right(righttree),1,righttree) ls
```

Figure 3. The function definition obtained from RightTree

the destructor Right, and the definition of `righttree` corresponds to the definition $t = \mathrm{Node}(\mathrm{Right}(t), 0, t)$ of RightTree. In this case, the termination of `righttree` is less obvious, but it can be automatically checked by an extention of the tool called MuHFL [18], an automated validity checker for a higher-order fixpoint logic.

We next discuss the case where a term is non-productive. Let us consider LeftTree defined as below, where "Left" is the destructor which retrieves the left child.

$$t = \mathrm{Node}(\mathrm{Left}(t), 1, t).$$

It is similar to RightTree, but it is not productive. Indeed, the reduction of t falls into the following infinite loop, never generating the label of the left child.

$$\begin{aligned} \underline{t} &\longrightarrow \mathrm{Node}(\mathrm{Left}(\underline{t}), 1, t) \\ &\longrightarrow \mathrm{Node}(\underline{\mathrm{Left}(\mathrm{Node}(\mathrm{Left}(t), 1, t))}, 1, t) \\ &\longrightarrow \mathrm{Node}(\underline{\mathrm{Left}(t)}, 1, t) \\ &\longrightarrow \dots \end{aligned}$$

The corresponding function `lefttree` is shown in Figure 4, where `left` corresponds to the destructor Left. The function `lefttree` is non-terminating for the input [0]. This non-termination can also be automatically checked by (the extension of) MuHFL.

```
let left t ls = t (0::ls)
let rec lefttree ls =
  node(left(lefttree),1,lefttree) ls
```

Figure 4. The function definition obtained from LefttTree

We formalize the reduction sketched above from productivity verification to termination verification for general co-inductive data structures (subsuming streams and binary trees), and prove the soundness and completeness of the reduction. The resulting termination verification problem is undecidable in general, but the proposed method is expected to work well in practice, thanks to the recent advance of automated termination verification techniques for higher-order functional programs [18, 21]. To show the effectiveness of our approach, we have implemented a fully-automated productivity verification tool. Since the existing higher-order program termination verification tools [18, 21] deal with only integers and functions, we have extended MuHFL [18] to support integer lists (which are needed to express paths) and used it as the backend termination verification tool.

The rest of this paper is structured as follows. Section 2 introduces the source and target languages of the transformation. Section 3 formalizes our transformation, and proves its correctness. Section 4 reports an implementation and experimental results. Section 5 discusses related work, and Section 6 concludes the paper. A preliminary summary of this paper appeared in Proceedings of PEPM 2024.

2 Source and Target Language

This section introduces the source and target languages for our reduction from productivity verification to termination verification.

2.1 Source Language

This section introduces the source language, which is a call-by-name lambda calculus equipped with co-inductive data types, and defines the notion of productivity. We adopt the call-by-name strategy, as co-inductive data can more naturally be expressed than in a call-by-value language.

2.1.1 Syntax and Operational Semantics. The syntax of types and terms of the source language is given in Figure 5. The types denoted by the meta-variable σ (which we call base types) describe non-functional data, including integers, pairs, sums, and co-inductive data structures. The type $\nu X.\sigma$ denotes the (equi-)recursive type X such that $X = \sigma$. For example, $\nu X.\mathtt{int} \times X$ describes an infinite stream consisting of integers, like $(1, (2, (3, \dots)))$. Streams and infinite binary trees can be represented as follows.

$$\begin{aligned} \mathit{Stream}\,\sigma &:= \nu X.\sigma \times X \\ \mathit{Tree}\,\sigma &:= \nu X.X \times \sigma \times X. \end{aligned}$$

$$
\begin{aligned}
\sigma &::= X \mid \mathtt{int} \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid \nu X.\sigma \ (\sigma \neq X) \\
\tau &::= \sigma \mid \tau_1 \to \tau_2 \\
M &::= n \mid x \mid \lambda x.M \mid M_0 M_1 \mid (M_0, M_1) \mid \pi_i(M) \mid in_i(M) \\
&\quad \mid \mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1) \\
&\quad \mid \mathtt{fix}\ x.M \mid \mathrm{Z} \\
V &::= n \mid (M_0, M_1) \mid in_i(M) \mid \mathrm{Z} \mid \lambda x.M
\end{aligned}
$$

Figure 5. Syntax of the source language

We can also handle nested recursive types, like the type of streams of streams:

$$Stream\,(Stream\,\sigma) = \nu X.(\nu Y.\sigma \times Y) \times X.$$

The meta-variable τ ranges over types including function types. Note that due to the classification between σ and τ, we do not allow function types to occur in co-inductive types. For example, the type $(\nu X.\mathtt{int} \times X) \to (\nu X.\mathtt{int} \times X)$ is allowed but not $\nu X.(\mathtt{int} \to \mathtt{int}) \times X$. Treating co-inductive data containing functions as elements is left for future work.

The syntax of terms, denoted by M, is fairly standard, except for the special term Z explained below. The meta-variables n and x respectively range over the sets of integers and variables. We have also λ-abstractions ($\lambda x.M$) and applications (M_0M_1), pair constructors (M_0, M_1) and destructors $\pi_i(M)$ ($i \in \{0, 1\}$), sum constructors $in_i(M)$ ($i \in \{0, 1\}$) and pattern matching $\mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1)$. The term $\mathtt{fix}\ x.M$ is a primitive for recursion, which evaluates to $[\mathtt{fix}\ x.M/x]M$ (where $[M_1/x]M_0$ denotes the term obtained by substituting M_1 for all the free occurrences of x in M_0). This fixpoint operator can be used for constructing co-inductive data structures, as given in Example 2.1 below. The term Z is a special value of any type σ, which is used only for defining the productivity. It is not supposed to occur in user programs.

Example 2.1. A stream consisting of infinite 1's is defined by:

$$
\begin{aligned}
&ones : \nu X.\mathtt{int} \times X \\
&ones := \mathtt{fix}\ x.(1, x)
\end{aligned}
$$

Since we adopt equi-recursive types, streams are represented by infinite tuples $(a_0, (a_1, (\ldots)))$.

The typing rules of the source language are given in Figure 6, which are fairly standard. As the last two rules show, types are treated equi-recursively. For example, $\nu X.\mathtt{int} \times X$ and $\mathtt{int} \times \nu X.\mathtt{int} \times X$ are equivalent to each other.

The call-by-name reduction relation $M \longrightarrow M'$ for terms is defined in Figure 7, which is also fairly standard. Note that the special term Z has no reduction rule. Thus, the evaluation of a term $\pi_0(\mathrm{Z})$, for example, gets stuck. This does not cause a problem since Z is used only in the definition of productivity,

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \frac{}{\Gamma \vdash n : \mathtt{int}} \quad \frac{}{\Gamma \vdash Z : \sigma}$$

$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash M_1 : \tau_2 \to \tau \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash M_1 M_2 : \tau}$$

$$\frac{\Gamma \vdash M_0 : \sigma_0 \quad \Gamma \vdash M_1 : \sigma_0}{\Gamma \vdash (M_0, M_1) : \sigma_0 \times \sigma_1} \quad \frac{\Gamma \vdash M : \sigma_0 \times \sigma_1}{\Gamma \vdash \pi_i(M) : \sigma_i}$$

$$\frac{\Gamma \vdash M : \sigma_0 + \sigma_1 \quad \Gamma, x : \sigma_0 \vdash M_0 : \tau \quad \Gamma, x : \sigma_1 \vdash M_1 : \tau}{\Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1) : \tau}$$

$$\frac{\Gamma \vdash M : \sigma_i}{\Gamma \vdash in_i(M) : \sigma_0 + \sigma_1} \quad \frac{\Gamma, x : \tau \vdash M : \tau}{\Gamma \vdash \mathtt{fix}\ x.M : \tau}$$

$$\frac{\Gamma \vdash M : \nu X.\tau}{\Gamma \vdash M : [\nu X.\tau/X]\tau} \quad \frac{\Gamma \vdash M : [\nu X.\tau/X]\tau}{\Gamma \vdash M : \nu X.\tau}$$

Figure 6. Typing rules of the source language.

$$\frac{}{(\lambda x.M_1)M_2 \longrightarrow [M_2/x]M_1} \quad \text{(R-Beta)}$$

$$\frac{}{\pi_i(M_0, M_1) \longrightarrow M_i} \quad \text{(R-Proj)}$$

$$\frac{}{\mathtt{case}\ in_i(M)\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1) \longrightarrow [M/x]M_i} \quad \text{(R-Case)}$$

$$\frac{}{\mathtt{fix}\ x.M \longrightarrow [\mathtt{fix}\ x.M/x]M} \quad \text{(R-Fix)}$$

$$\frac{M_0 \longrightarrow M_0'}{M_0 M_1 \longrightarrow M_0' M_1} \quad \text{(R-CApp)}$$

$$\frac{M \longrightarrow M'}{\pi_i(M) \longrightarrow \pi_i(M')} \quad \text{(R-CProj)}$$

$$\frac{M \longrightarrow M'}{\mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1) \longrightarrow \mathtt{case}\ M'\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1)} \quad \text{(R-CCase)}$$

Figure 7. Operational semantics of the source language.

and such a term never occurs during the actual evaluation of a term (on the assumption that Z never occurs in a term provided by a user).

2.1.2 Productivity. There are some variations in the definition of productivity. For example, Endrullis [10] calls a stream M productive if M converges to a unique infinite term. We do not adopt such a definition because infinite terms and convergence are hard to treat in our approach. Instead, we define a term M of type σ is productive if γM is terminating for any projection γ that extracts an element of the data structure produced by M. We first define the set $\mathbf{Path}(\sigma)$ consisting of paths to elements of data of type σ. The set $\mathbf{Path}(\sigma) \subseteq \{\pi_0, \pi_1, in_0^{-1}, in_1^{-1}\}^*$ is defined as the least set that satisfies:

$$\mathbf{Path}(\sigma) \supseteq \{\epsilon\}$$
$$\mathbf{Path}(\sigma_0 \times \sigma_1) \supseteq \{\pi_i \cdot \gamma \mid i \in \{0,1\}, \gamma \in \mathbf{Path}(\sigma_i)\}$$
$$\mathbf{Path}(\sigma_0 + \sigma_1) \supseteq \{in_i^{-1} \cdot \gamma \mid i \in \{0,1\}, \gamma \in \mathbf{Path}(\sigma_i)\}$$
$$\mathbf{Path}(\nu X.\sigma) \supseteq \mathbf{Path}([\nu X.\sigma/X]\sigma).$$

For $\gamma \in \{\pi_0, \pi_1, in_0^{-1}, in_1^{-1}\}^*$ we define $\gamma\, M$, which extracts from M the element at the path γ, by:

$$\epsilon\, M = M$$
$$(\pi_i \cdot \gamma)\, M = \gamma\, (\pi_i(M))$$
$$(in_0^{-1} \cdot \gamma)\, M = \mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow \gamma\, x \mid in_1(x) \Rightarrow \mathrm{Z})$$
$$(in_1^{-1} \cdot \gamma)\, M = \mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow \mathrm{Z} \mid in_1(x) \Rightarrow \gamma\, x).$$

We can now define the productivity of a term. We call a term Z-*free* if the term does not contain any occurrence of Z.

Definition 2.2 (Productivity)**.** A Z-free term M of type σ is *productive* if $\gamma\, M$ is terminating (i.e., has no infinite reduction sequence) for every $\gamma \in \mathbf{Path}(\sigma)$.

Note that, in our definition of the productivity, a term M of a sum type is productive if both terms below are terminating.

$$\mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow x \mid in_1(x) \Rightarrow \mathrm{Z})$$
$$\mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow \mathrm{Z} \mid in_1(x) \Rightarrow x)$$

For example, if $M \longrightarrow^* in_0(M')$ holds, we get

$$\mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow x \mid in_1(x) \Rightarrow \mathrm{Z}) \longrightarrow^* M'$$
$$\mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow \mathrm{Z} \mid in_1(x) \Rightarrow x) \longrightarrow^* \mathrm{Z}.$$

The termination of the first one ensures that the element M' can be evaluated in finite time, and that of the second one trivially holds since Z is a value.

Example 2.3. For integer streams $\sigma = \nu X.\mathtt{int} \times X$,

$$\mathbf{Path}(\sigma) = \{\epsilon, \pi_0, \pi_1, \pi_1\pi_0, \dots\}$$

The productivity of a stream M asserts that i-th element

$$\pi_0(\underbrace{\pi_1(\cdots \pi_1}_{i \text{ times}}(M) \cdots))$$

can be evaluated in finite time for every $i \in \mathbb{N}$. Thus, our definition of productivity coincides with the standard definition of the productivity of streams.

$$\begin{aligned}
\sigma &::= \mathtt{int} \mid \mathtt{intlist} \\
\tau &::= \sigma \mid \tau_1 \to \tau_2 \\
N &::= x \mid n \mid [\,] \mid n :: N \mid \lambda x.N \mid \Lambda x.N \mid N_1 N_2 \\
&\quad \mid \mathtt{if0}\ N\ \mathtt{then}\ N_0\ \mathtt{else}\ N_1 \\
&\quad \mid \mathtt{case}\ N\ \mathtt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \\
&\quad \mid \mathtt{fix}\ x.N \\
V &::= n \mid [\,] \mid n :: V \mid \lambda x.N \mid \Lambda x.N \\
pat &::= [\,] \mid n :: x \mid _ \\
L &::= [\,] \mid n :: N
\end{aligned}$$

Figure 8. Syntax of the target language.

The following lemma guarantees the well-typedness of $\gamma\, M$.

Lemma 2.4. *If* $\vdash M : \sigma$ *and* $\gamma \in \mathbf{Path}(\sigma)$*, then* $\vdash \gamma\, M : \sigma'$ *for some* σ'*.*

Remark 2.1. *Note that although our definition of productivity uses "termination", the termination of a source program means only weak head normalization; for example, the term* $in_0(\mathtt{fix}\ x.x)$ *is terminating. Also, the definition involves the quantification over all the paths. Thus the productivity as defined above cannot be directly checked by ordinary automated termination checkers.*

2.2 Target Language

The target language is a call-by-name lambda calculus without co-inductive data types but equipped with integer lists. The syntax of types and terms of the target language is given in Figure 8. The target language only has `int`, the type `intlist` of integer lists, and function types as types, and does not have product or sum types. Integer lists are used for representing paths in the previous section, and product and sum terms are translated into a function that takes a path and returns the corresponding element.

The syntax of terms, denoted by N, is standard, except for the extra lambda abstraction $\Lambda x.N$ explained below. The terms $[\,]$ and $N_1 :: N_2$ are the list constructors. The term $\mathtt{if0}\ N\ \mathtt{then}\ N_0\ \mathtt{else}\ N_1$ is reduced to N_0 if $N = 0$ and reduced to N_1 otherwise. The if-expression is used to emulate pattern matching in the source language. The term $\mathtt{case}\ N\ \mathtt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)$ represents pattern matching on a list N and pat is a meta-variable for patterns. Pattern matching can be used only for paths represented by lists. For example, a pair (M_0, M_1) is translated into a function that takes a list and returns the elements in M_0 if the list starts from 0 and the elements in M_1 if it starts from 1. This conditional branch is realized by pattern matching. The target language has the fixpoint operator $\mathtt{fix}\ x.N$ as in the source language. Co-inductive data structures constructed by fixpoints are translated into recursive functions in the

$$\frac{}{\mathcal{E}[(\lambda x.N_1)N_2] \longrightarrow \mathcal{E}[[N_2/x]N_1]}$$

$$\frac{}{\mathcal{E}[(\Lambda x.N_1)N_2] \longrightarrow \mathcal{E}[[N_2/x]N_1]}$$

$$\frac{}{\mathcal{E}[\texttt{if0 0 then } N_0 \texttt{ else } N_1] \longrightarrow \mathcal{E}[N_0]}$$

$$\frac{n \neq 0}{\mathcal{E}[\texttt{if0 } n \texttt{ then } N_0 \texttt{ else } N_1] \longrightarrow \mathcal{E}[N_1]}$$

$$\frac{}{\mathcal{E}[\texttt{fix } x.N] \longrightarrow \mathcal{E}[[\texttt{fix } x.N/x]N]}$$

$$\frac{\neg\exists\rho_j.L \vdash pat_j \Rightarrow \rho_j \text{ for each } j < i \qquad L \vdash pat_i \Rightarrow \rho}{\mathcal{E}[\texttt{case } L \texttt{ of } (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)] \longrightarrow \mathcal{E}[\rho N_i]}$$

$$\frac{}{[\,] \vdash [\,] \Rightarrow [\cdot]}$$

$$\frac{}{L \vdash \_ \Rightarrow [\cdot]}$$

$$\frac{}{n :: L' \vdash n :: x \Rightarrow [L'/x]}$$

$$\begin{aligned}\mathcal{E} ::= {} & \langle\,\rangle \mid \mathcal{E}\ N \mid \texttt{if0 } \mathcal{E} \texttt{ then } N_0 \texttt{ else } N_1 \\ & \mid \texttt{case } \mathcal{E} \texttt{ of } (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)\end{aligned}$$

Figure 9. Operational semantics of the target language.

$$\frac{}{\Delta, x : \tau \vdash x : \tau}$$

$$\frac{}{\Delta \vdash n : \texttt{int}}$$

$$\frac{\Delta, x : \tau_1 \vdash N : \tau_2}{\Delta \vdash \lambda x.N : \tau_1 \to \tau_2}$$

$$\frac{\Delta \vdash N_1 : \tau_2 \to \tau \qquad \Delta \vdash N_2 : \tau_2}{\Delta \vdash N_1 N_2 : \tau}$$

$$\frac{\Delta \vdash N : \texttt{int} \qquad \Delta \vdash N_0 : \tau \qquad \Delta \vdash N_1 : \tau}{\Delta \vdash \texttt{if0 } N \texttt{ then } N_0 \texttt{ else } N_1 : \tau}$$

$$\frac{\Delta \vdash N : \texttt{intlist} \qquad \Delta, \Delta_i \vdash N_i : \tau\ (i \in \{1, \ldots, k\}) \qquad \Delta_i = \begin{cases} x : \texttt{intlist} & (pat_i = n :: x) \\ \cdot & (\text{otherwise}) \end{cases}}{\Delta \vdash \texttt{case } N \texttt{ of } (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) : \tau}$$

$$\frac{}{\Delta \vdash [\,] : \texttt{intlist}}$$

$$\frac{\Delta \vdash N_1 : \texttt{int} \qquad \Delta \vdash N_2 : \texttt{intlist}}{\Delta \vdash N_1 :: N_2 : \texttt{intlist}}$$

$$\frac{\Delta, x : \tau \vdash N : \tau}{\Delta \vdash \texttt{fix } x.N : \tau}$$

$$\frac{\Delta, x : \texttt{intlist} \vdash N : \tau}{\Delta \vdash \Lambda x.N : \texttt{intlist} \to \tau}$$

Figure 10. Typing rules of the Target Language.

target language. In the target language, we have another lambda abstraction $\Lambda x.N$, which can only be used for path abstractions, i.e., the argument x is a path introduced by the translation. The semantics and the typing rules of the target language are given in Figures 9 and 10 respectively. In the definition of the semantics, $L \vdash pat \Rightarrow \rho$ means that the list value L matches the pattern pat, and the resulting substitution is ρ. Here, $[\cdot]$ denotes the identity substitution.

3 Transformation

In this section, we define the transformation and prove the correctness of the transformation.

3.1 Definition of the Transformation

As mentioned in the introduction, we transform each co-inductive data into a function that maps paths to nodes in the corresponding tree structure to elements of the nodes. By the transformation, the productivity of co-inductive data is reduced to the termination of the corresponding function.

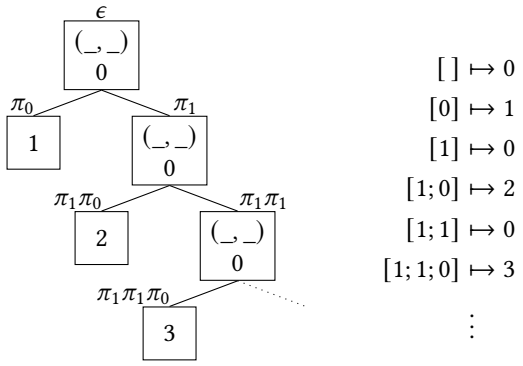
We first explain the idea of the transformation through the example of the stream $(1, (2, (3, \ldots)))$ of type $\nu X.\texttt{int} \times X$. As Example 2.3, the set $\mathbf{Path}(\nu X.\texttt{int} \times X)$ of the paths is $\{\epsilon, \pi_0, \pi_1, \pi_1\pi_0, \ldots\}$. Each path corresponds to the element in the stream as in Figure 11a, and a path γ in the source language is transformed into a integer list $\gamma^\dagger$ as follows:

$$\epsilon^\dagger = [\,]$$
$$(\pi_i \cdot \gamma)^\dagger = i :: \gamma^\dagger$$
$$(in_i^{-1} \cdot \gamma)^\dagger = 0 :: \gamma^\dagger$$

As a consequence, the stream $(1, (2, (3, \ldots)))$ is transformed into the function as shown in Figure 11b. Here, the labels of nodes corresponding to the constructors of product and sum types are transformed into some integer. For example, for the top constructor $(\_, \_)$ at the path ϵ, the transformed function returns 0.

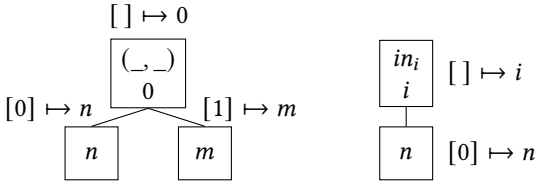
The transformation of types and type environments is as follows:

$$\sigma^\dagger = \texttt{intlist} \to \texttt{int} \qquad (\tau_1 \to \tau_2)^\dagger = \tau_1^\dagger \to \tau_2^\dagger$$
$$(x_1 : \tau_1, \ldots, x_k : \tau_k)^\dagger = x_1 : \tau_1^\dagger, \ldots, x_k : \tau_k^\dagger$$



(a) Original stream. **(b)** Transformed function.

Figure 11. A stream (left) and the corresponding function (right).



(a) Transformation of pair. **(b)** Transformation of injection.

Figure 12. Transformation of constructors.

Base types are transformed into function types from lists to integers. Note that all the base types including product and sum types are transformed into the type $\texttt{intlist} \to \texttt{int}$ because the nodes corresponding to product and sum types are transformed into some integer as in the example above.

We now define our transformation in a type-directed style. Figure 13 shows the definition of the transformation. The transformation judgment is of the form $\Gamma \vdash M : \tau \rightsquigarrow N$, which means that, under the type environment Γ, the term M of type τ in the source language is translated to N. For example, an integer n has type $\texttt{int}$ under the empty type environment, and is translated into a function as given below:

$$\vdash n : \texttt{int} \rightsquigarrow \Lambda u.\texttt{case}\ u\ \texttt{of}\ [\,] \Rightarrow n \mid \_ \Rightarrow 0.$$

We explain some of the key rules below. In the rule Tr-Num, an integer n is transformed into the function that returns n if the argument is $[\,]$ and returns 0 otherwise. In the rule Tr-Pair, the function obtained by the translation returns the dummy value 0 for the empty path $[\,]$. Figure 12a illustrates the transformation of pairs. A pair (m, n) is transformed into the function that returns 0 if the argument is $[\,]$, m if the argument is $[0]$, n if the argument is $[1]$, and 0 otherwise. In the rule Tr-Inj, the function obtained by the translation of $in_i(M)$ returns i. This value is used to emulate pattern matching in the target language. Figure 12b illustrates the transformation of injections. An injection $in_i(n)$ is transformed into a function that returns i if the argument is $[\,]$, n if it is $[0]$, and 0 otherwise. The rule Tr-Proj transforms a projection $\pi_i(M)$ into the function that calls the function corresponds to M and pass an argument $i :: u$ if the argument is u. The rule Tr-Case transforms a pattern matching into an conditional branch in the target language. The term $N\,[\,]$ is supposed to reduce to an integer that represents whether M is in_0 or in_1. We can thus emulate pattern matching by the conditional branch. For a closed term M of type τ, we often write f_M for the output of the transformation, i.e., for the term N such that $\vdash M : \tau \rightsquigarrow N$.

Below we give some examples of the transformation and the reductions of the transformed terms.

Example 3.1. Let $M = \texttt{case}\ in_0(n)\ \texttt{of}\ (in_0(x) \Rightarrow x \mid in_1(x) \Rightarrow 0)$. The term M and its subterms are transformed as follows:

$$\begin{aligned}
f_M &= \texttt{if0}\ f_{in_0(n)}\ [\,]\ \texttt{then}\ \Lambda u.f_{in_0(n)}\ (0 :: u)\ \texttt{else}\ f_0 \\
f_{in_0(n)} &= \Lambda u.\texttt{case}\ u\ \texttt{of}\ ([\,] \Rightarrow 0 \mid 0 :: u' \Rightarrow f_n\,u' \mid _ \Rightarrow 0) \\
f_n &= \Lambda u.\texttt{case}\ u\ \texttt{of}\ [\,] \Rightarrow n \mid _ \Rightarrow 0 \\
f_0 &= \Lambda u.\texttt{case}\ u\ \texttt{of}\ [\,] \Rightarrow 0 \mid _ \Rightarrow 0.
\end{aligned}$$

The root element $f_M\,[\,]$ can be reduced as below:

$$\begin{aligned}
&f_M\,[\,] = (\texttt{if0}\ f_{in_0(n)}\ [\,]\ \texttt{then}\ \Lambda u.f_{in_0(n)}\ (0 :: u)\ \texttt{else}\ f_0)\,[\,] \\
&\longrightarrow \texttt{if0}\ 0\ \texttt{then}\ \Lambda u.f_{in_0(n)}\ (0 :: u)\ \texttt{else}\ f_0\,[\,] \\
&\longrightarrow (\Lambda u.f_{in_0(n)}\ (0 :: u))\,[\,] \\
&\longrightarrow f_{in_0(n)}\ (0 :: [\,]) \\
&= (\Lambda u.\texttt{case}\ u\ \texttt{of}\ ([\,] \Rightarrow i \mid 0 :: u' \Rightarrow f_n\,u' \mid _ \Rightarrow 0))\ (0 :: [\,]) \\
&\longrightarrow \texttt{case}\ 0 :: [\,]\ \texttt{of}\ ([\,] \Rightarrow i \mid 0 :: u' \Rightarrow f_n\,u' \mid _ \Rightarrow 0) \\
&\longrightarrow f_n\,[\,] \\
&\longrightarrow^* n.
\end{aligned}$$

This reduction sequence corresponds to the following reduction of the original term

$$\texttt{case}\ in_0(n)\ \texttt{of}\ (in_0(x) \Rightarrow x \mid in_1(x) \Rightarrow 0) \longrightarrow n.$$

Example 3.2. Let $M = \pi_0(m, n)$. The term M and its subterms are transformed as follows:

$$\begin{aligned}
f_M &= \Lambda u.f_{(m,n)}(0 :: u) \\
f_{(m,n)} &= \Lambda u.\texttt{case}\ u\ \texttt{of}\ (0 :: u' \Rightarrow f_m\,u' \\
&\qquad\qquad \mid 1 :: u' \Rightarrow f_n\,u' \mid _ \Rightarrow 0) \\
f_m &= \Lambda u.\texttt{case}\ u\ \texttt{of}\ ([\,] \Rightarrow m \mid _ \Rightarrow 0) \\
f_n &= \Lambda u.\texttt{case}\ u\ \texttt{of}\ ([\,] \Rightarrow n \mid _ \Rightarrow 0)
\end{aligned}$$

We next apply nil to f_M. The term $f_M\,[\,]$ is reduced as below:

$$\begin{aligned}
f\,[\,] &= (\Lambda u.f_{(m,n)}(0 :: u))[\,] \\
&\longrightarrow f_{(m,n)}\ (0 :: [\,]) \\
&\longrightarrow f_m\,[\,]
\end{aligned}$$

$$\frac{}{\Gamma, x:\tau \vdash x:\tau \rightsquigarrow x}\ (\textsc{Tr-Var}) \qquad \frac{}{\Gamma \vdash n:\mathtt{int} \rightsquigarrow \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ [\,] \Rightarrow n \mid \_ \Rightarrow 0}\ (\textsc{Tr-Num}) \qquad \frac{}{\Gamma \vdash \mathrm{Z}:\sigma \rightsquigarrow \Lambda u.0}\ (\textsc{Tr-Z})$$

$$\frac{\Gamma, x:\tau_1 \vdash M:\tau_2 \rightsquigarrow N}{\Gamma \vdash \lambda x.M:\tau_1 \to \tau_2 \rightsquigarrow \lambda x.N}\ (\textsc{Tr-Abs}) \qquad \frac{\Gamma \vdash M_1:\tau_2 \to \tau \rightsquigarrow N_1 \qquad \Gamma \vdash M_2:\tau_2 \rightsquigarrow N_2}{\Gamma \vdash M_1 M_2:\tau \rightsquigarrow N_1 N_2}\ (\textsc{Tr-App})$$

$$\frac{\Gamma \vdash M_0:\sigma_0 \rightsquigarrow N_0 \qquad \Gamma \vdash M_1:\sigma_1 \rightsquigarrow N_1}{\Gamma \vdash (M_0, M_1):\sigma_0 \times \sigma_1 \rightsquigarrow \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ 0::u' \Rightarrow N_0\,u' \mid 1::u' \Rightarrow N_1\,u' \mid \_ \Rightarrow 0}\ (\textsc{Tr-Pair})$$

$$\frac{\Gamma, x:\tau \vdash M:\tau \rightsquigarrow N}{\Gamma \vdash \mathtt{fix}\ x.M:\tau \rightsquigarrow \mathtt{fix}\ x.N}\ (\textsc{Tr-Fix}) \qquad \frac{\Gamma \vdash M:\sigma_0 \times \sigma_1 \rightsquigarrow N}{\Gamma \vdash \pi_i(M):\sigma_i \rightsquigarrow \Lambda u.N(i::u)}\ (\textsc{Tr-Proj})$$

$$\frac{\Gamma \vdash M:\sigma_i \rightsquigarrow N}{\Gamma \vdash in_i(M):\sigma_0+\sigma_1 \rightsquigarrow \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ [\,] \Rightarrow i \mid 0::u' \Rightarrow N\,u' \mid \_ \Rightarrow 0}\ (\textsc{Tr-Inj})$$

$$\frac{\Gamma \vdash M:\sigma_0+\sigma_1 \rightsquigarrow N \qquad \Gamma, x:\sigma_0 \vdash M_0:\tau \rightsquigarrow N_0 \qquad \Gamma, x:\sigma_1 \vdash M_1:\tau \rightsquigarrow N_1}{\begin{array}{l}\Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1):\tau \\ \rightsquigarrow \mathtt{if0}\ N\ [\,]\ \mathtt{then}\ [\Lambda u.N\,(0::u)/x]N_0\ \mathtt{else}\ [\Lambda u.N\,(0::u)/x]N_1\end{array}}\ (\textsc{Tr-Case})$$

$$\frac{\Gamma \vdash M:\nu X.\tau \rightsquigarrow N}{\Gamma \vdash M:[\nu X.\tau/X]\tau \rightsquigarrow N}\ (\textsc{Tr-Unfold}) \qquad \frac{\Gamma \vdash M:[\nu X.\tau/X]\tau \rightsquigarrow N}{\Gamma \vdash M:\nu X.\tau \rightsquigarrow N}\ (\textsc{Tr-Fold})$$

Figure 13. Transformation Rules.

This reduction sequence corresponds to a reduction

$$\pi_0(m,n) \longrightarrow m$$

in the source language.

Example 3.3. The stream of 1's in Example 2.1 is translated to the following function.

$$\begin{aligned} f_{ones} =& \mathtt{fix}\ x.\Lambda u.\mathtt{case}\ u\ \mathtt{of} \\ & (0::u \Rightarrow f_1\,u \mid 1::u \Rightarrow x\,u \mid \_ \Rightarrow 0) \end{aligned}$$

where

$$f_1 = \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ [\,] \Rightarrow 1 \mid \_ \Rightarrow 0.$$

As expected, the function f_{ones} returns 1 if the input path is of the form $[1;\cdots;1;0]$.

Example 3.4. The RightTree discussed in the introduction is represented by:

$$M = \mathtt{fix}\ x.(\pi_1(\pi_1(x)),(0,x)).$$

It is translated to:

$$\begin{aligned} f_M &= \mathtt{fix}\ x.f_{(\pi_1(\pi_1(x)),(0,x))} \\ f_{(\pi_1(\pi_1(x)),(0,x))} &= \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ (0::u' \Rightarrow f_{\pi_1(\pi_1(x))}\,u' \\ &\qquad \mid 1::u' \Rightarrow f_{(0,x)}\,u' \mid \_ \Rightarrow 0) \\ f_{\pi_1(\pi_1(x))} &= \Lambda u.f_{\pi_1(x)}\,(1::u) \\ f_{\pi_1(x)} &= \Lambda u.x\,(1::u) \\ f_{(0,x)} &= \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ (0::u' \Rightarrow f_0\,u' \\ &\qquad \mid 1::u' \Rightarrow x\,u' \mid \_ \Rightarrow 0) \\ f_0 &= \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ [\,] \Rightarrow 0 \mid \_ \Rightarrow 0 \end{aligned}$$

The element of the left child of the root node corresponds to $f_M\,[0;1;0]$, which can be calculated as:

$$\begin{aligned} & f_M\,[0;1;0] \\ \longrightarrow\ & [f_M/x]f_{(\pi_1(\pi_1(x)),(0,x))}\,[0;1;0] \\ \longrightarrow\ & f_{\pi_1(\pi_1(M))}\,[1;0] \\ \longrightarrow\ & f_{\pi_1(M)}\,[1;1;0] \\ \longrightarrow\ & f_M\,[1;1;1;0] \\ \longrightarrow\ & [f_M/x]f_{(\pi_1(\pi_1(x)),(0,x))}\,[1;1;1;0] \\ \longrightarrow\ & f_{(0,M)}\,[1;1;0] \\ \longrightarrow\ & f_M\,[1;0] \\ \longrightarrow\ & [f_M/x]f_{(\pi_1(\pi_1(x)),(0,x))}\,[1;0] \\ \longrightarrow\ & f_{(0,M)}\,[0] \\ \longrightarrow\ & f_0\,[\,] \end{aligned}$$

$\longrightarrow 0$.

Here, we used the fact that the translation and substitution commute, as stated in the lemma given below. Similarly, the elements under the left child can be calculated. Thus, the function f_M is terminating for any list argument p. □

Lemma 3.5 (Substitution). *If* $\Gamma, x : \tau_1 \vdash M_0 : \tau \rightsquigarrow N_0$ *and* $\Gamma \vdash M_1 : \tau_1 \rightsquigarrow N_1$, *then* $\Gamma \vdash [M_1/x]M_0 : \tau \rightsquigarrow [N_1/x]N_0$.

The following lemma guarantees the well-typedness of the transformed term.

Lemma 3.6. *If* $\Gamma \vdash M : \sigma \rightsquigarrow N$, *then* $\Gamma^\dagger \vdash N : \sigma^\dagger$

The lemma follows by straightforward induction on the derivation of $\Gamma \vdash M : \sigma \rightsquigarrow N$.

3.2 Correctness of the Transformation

In this section, we show the correctness of the translation, i.e., soundness and completeness. Detailed proofs are found in Appendix A.

The soundness theorem below states that a source program is productive if the translated program is terminating for any list argument.

Theorem 3.7 (Soundness). *Suppose* $\vdash M : \sigma \rightsquigarrow N$. *If* $N\,p$ *is terminating for every value* p *of* `intlist`, *then* M *is productive.*

The completeness theorem below states the converse: a source program is non-productive if the translated program is non-terminating for some list argument.

Theorem 3.8 (Completeness). *Suppose* $\vdash M : \sigma \rightsquigarrow N$. *If* M *is productive, then* $N\,p$ *is terminating for every value* p *of* `intlist`.

As a consequence of the soundness and the completeness, we can use a termination/non-termination checker to prove the productivity/non-productivity.

In order to show the theorems above, we first introduce another semantics of the target language. Since our translation introduces new abstractions for paths, a redex in the original program does not directly correspond to one in the translated program. Thus, we introduce the *extended* reduction $\longrightarrow_e$ to emulate reductions, in which reductions may occur under path abstractions Λu. We define $\longrightarrow_e$ as the same reduction as $\longrightarrow$ except that the following extended evaluation context $\mathcal{E}_e$ is used instead of $\mathcal{E}$.

$$\mathcal{E}_e ::= \langle\,\rangle \mid \mathcal{E}_e\,N \mid \texttt{if0}\ \mathcal{E}_e\ \texttt{then}\ N_0\ \texttt{else}\ N_1 \mid \texttt{case}\ \mathcal{E}_e\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \mid \Lambda x.\mathcal{E}_e.$$

The following lemma guarantees that the new semantics is equivalent to the original semantics with respect to the termination property of terms of base types.

Lemma 3.9. *Let* N *be a term of a base type. Then the followings are equivalent.*

1. N *is terminating with respect to* $\longrightarrow$, *i.e., there exists no infinite reduction sequence* $N \longrightarrow N_1 \longrightarrow N_2 \longrightarrow \cdots$.
2. N *is terminating with respect to* $\longrightarrow_e$, *i.e., there exists no infinite reduction sequence* $N \longrightarrow_e N_1 \longrightarrow_e N_2 \longrightarrow_e \cdots$.
3. N *is may-terminating with respect to* $\longrightarrow_e$, *i.e., there exists a reduction sequence* $N \longrightarrow_e^* V \not\longrightarrow_e$.

Hereafter, we use mainly $\longrightarrow_e$ instead of $\longrightarrow$ for translated programs, and do not distinguish between termination and may-termination for terms of base types.

3.2.1 Soundness.

The soundness theorem is a corollary of the following two lemmas:

Lemma 3.10. *If* $f_M\,[\,]$ *is terminating, then* M *is terminating.*

Lemma 3.11. *Let* γ *be a sequence in* $\{\pi_0, \pi_1, in_0^{-1}, in_1^{-1}\}^*$ *and* M *be a closed term of type* σ. *If* $\gamma \in \mathbf{Path}(\sigma)$ *and* $f_M\,p$ *is terminating for every list value* p, *then* $f_{\gamma M}\,q$ *is terminating for every list value* q.

Using the lemmas above, Theorem 3.7 is obtained as follows.

Proof of Theorem 3.7. Suppose $\vdash M : \sigma \rightsquigarrow N$ and $N\,p$ is terminating for every value p of type `intlist`. We need to show that γM is terminating for every $\gamma \in \mathbf{Path}(\sigma)$. By the assumption and Lemma 3.11, it follows that $f_{\gamma M}[\,]$ is terminating. By Lemma 3.10, γM is also terminating. □

It remains to show the above lemmas. Lemma 3.11 follows by a straightforward induction on the length of γ: see Appendix A.

In the rest of this subsection, we give a proof sketch of the first lemma, which is shown by contraposition. Suppose M is non-terminating, i.e., there exists an infinite reduction sequence:

$$M \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \cdots.$$

We show that $f_M\,[\,]$ also has an infinite reduction sequence. To that end, we first define a strong reduction relation $\geq$ in Figure 14. Here, η-reduction is allowed by C-Eta.

By using the strong reduction, we can state that f_M simulates M in the following sense.

Lemma 3.12. *If* $\vdash M : \tau \rightsquigarrow N$ *and* $M \longrightarrow M'$, *then there exists* N' *such that* $\vdash M' : \tau \rightsquigarrow N'$ *and* $N \longrightarrow_e^+ \geq^* N'$.

Proof. The proof proceeds by induction on the derivation of $M \longrightarrow M'$, with case analysis on the last rule used. □

Additionally, we can exchange $\longrightarrow_e$ and $\geq$ as the following lemma.

Lemma 3.13. *If* $N \geq \longrightarrow_e N'$, *then* $N \longrightarrow_e^+ \geq^* N'$.

Proof. The proof proceeds by case analysis on the evaluation context C of the strong reduction. □

We can now prove Lemma 3.10.

$$\frac{x \text{ is not free in } N}{C[\Lambda x.N\,x] \geq C[N]}\ (\text{C-Eta}) \qquad \frac{N \longrightarrow_e N'}{C[N] \geq C[N']}\ (\text{C-Beta})$$

$$\begin{aligned} C ::= {} & \langle\,\rangle \mid C\,N \mid N\,C \mid n :: C \mid \lambda x.C \mid \Lambda x.C \\ & \mid \texttt{if0}\ C\ \texttt{then}\ N_0\ \texttt{else}\ N_1 \\ & \mid \texttt{if0}\ N\ \texttt{then}\ C\ \texttt{else}\ N_1 \mid \texttt{if0}\ N\ \texttt{then}\ N_0\ \texttt{else}\ C \\ & \mid \texttt{case}\ C\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \\ & \mid \texttt{case}\ N\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_{i-1} \Rightarrow N_{i-1} \mid \cdots \\ & \qquad \mid pat_i \Rightarrow C \mid pat_{i+1} \Rightarrow N_{i+1} \mid \cdots \mid pat_k \Rightarrow N_k) \end{aligned}$$

Figure 14. Strong reduction relation $\geq$

Proof of Lemma 3.10. Suppose there is an infinite reduction sequence:

$$M \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \cdots$$

By Lemma 3.12, we get

$$f_M \longrightarrow_e^+ \geq^* f_{M_1} \longrightarrow_e^+ \geq^* f_{M_2} \longrightarrow_e^+ \geq^* \cdots$$

By using Lemma 3.13 repeatedly, we can construct an infinite reduction sequence starting from f_M, so we obtain an infinite reduction sequence starting from $f_M\,[\,]$. □

3.2.2 Completeness. Below we assume that M is a closed term of a base type. To prove the completeness theorem, we have to show $f_M\,p$ is terminating for every list value p assuming M is productive. We first show some auxiliary lemmas.

Lemma 3.14. *If $M \longrightarrow M'$, then $\gamma M \longrightarrow \gamma M'$.*

Proof. By straightforward induction on γ. □

Lemma 3.15. *Let N_0, N_1 be closed terms of type* `int` *and V be a value. If $N_0 \geq N_1$ and $N_1 \longrightarrow_e^* V$, then $N_0 \longrightarrow_e^* V$ holds.*

Lemma 3.16. *Let M be a closed term of base type and p be an integer list. If $M \longrightarrow M'$, then $f_M\,p$ is terminating if and only if $f_{M'}\,p$ is terminating.*

Proof of Lemma 3.16. By Lemmas 3.12 and 3.13, we have

$$f_M\,p \longrightarrow_e^+ \geq^* f_{M'}\,p.$$

To show the "if" direction, suppose $f_{M'}\,p \longrightarrow_e^* V$. Since $f_M\,p$ is closed term of type `int`, we get $f_M\,p \longrightarrow_e^* V$ by using Lemma 3.15 repeatedly.

To show the "only-if" direction, we prove the contraposition. Suppose $f_{M'}\,p$ is non-terminating. By Lemmas 3.12 and 3.13, we can construct an infinite reduction sequence from $f_M\,p$. □

Lemma 3.17. *Let M be a closed term of base type. If M is terminating, then $f_M\,[\,]$ is terminating.*

Proof. Suppose $M \longrightarrow^* V$. By Lemma 3.16, it is sufficient to show $f_V\,[\,]$ is terminating, which follows by straightforward by case analysis on V. □

The following is another key lemma for the completeness theorem.

Lemma 3.18. *Let p be an integer list and M be a term of type σ. Suppose M is productive. Then, $f_M\,p$ is terminating if $f_{\gamma M}\,[\,]$ is terminating for any $\gamma \in \mathbf{Path}(\sigma)$.*

Proof. By induction on the structure of p. The case $p = [\,]$ is trivial. Suppose $p = i :: p'$ for some i and p'. We perform case analysis on i and σ. We only show the important cases here. Other cases are straightforward or similar to the cases below.

Case $i = 0$ and $\sigma = \sigma_0 \times \sigma_1$: Since M is productive, $M \longrightarrow^* (M_0, M_1)$ for some M_0 and M_1. By Lemma 3.16, it suffices to show $f_{(M_0,M_1)}\,p$ is terminating. By the assumption, $f_{\gamma(M_0,M_1)}\,[\,]$ is terminating for any $\gamma \in \mathbf{Path}(\sigma)$. Hence, by Lemmas 3.14 and 3.16, $f_{\gamma' M_0}\,[\,]$ is terminating for any $\gamma' \in \mathbf{Path}(\sigma_0)$. Therefore, by I.H., $f_{M_0}\,p'$ is terminating. Here,

$$\begin{aligned} & f_{(M_0,M_1)} \\ & = \Lambda u.\texttt{case}\ u\ \texttt{of}\ 0 :: u' \Rightarrow f_{M_0}\,u' \mid 1 :: u' \Rightarrow f_{M_1}\,u' \mid \_ \Rightarrow 0 \end{aligned}$$

and $f_{(M_0,M_1)}\,p \longrightarrow^* f_{M_0}\,p'$. Since $f_{M_0}\,p'$ is terminating, $f_{(M_0,M_1)}\,p$ is also terminating.

Case $i = 0$ and $\sigma = \sigma_0 + \sigma_1$: Since M is productive, $M \longrightarrow^* in_j(M_0)$ for some j and M_0. We now show $f_{in_j(M_0)}\,p$ is terminating. By the assumption, $f_{(in_j^{-1}\cdot\gamma')M}$ is terminating for any $\gamma' \in \mathbf{Path}(\sigma_j)$. Since $(in_j^{-1} \cdot \gamma')M \longrightarrow^* \gamma' M_0$, by Lemma 3.16, $f_{\gamma' M_0}\,[\,]$ is also terminating. Hence, by I.H., $f_{M_0}\,p'$ is terminating. Here,

$$f_{in_j(M_0)} = \Lambda u.\texttt{case}\ u\ \texttt{of}\ [\,] \Rightarrow j \mid 0 :: u' \Rightarrow f_{M_0}\,u' \mid \_ \Rightarrow 0$$

and $f_{in_j(M_0)}\,p \longrightarrow^* f_{M_0}\,p'$. Since $f_{M_0}\,p'$ is terminating, $f_{in_j(M_0)}\,p$ is also terminating. □

The completeness theorem follows from Lemmas 3.17 and 3.18, as follows.

Proof of Theorem 3.8. Suppose $\vdash M : \sigma \rightsquigarrow f_M$ and M is productive. By the definition of productivity, γM is terminating for any $\gamma \in \mathbf{Path}(\sigma)$. Thus, by Lemma 3.17, $f_{\gamma M}\,[\,]$ is also terminating for any $\gamma \in \mathbf{Path}(\sigma)$. By Lemma 3.18, it follows that $f_M\,p$ is terminating for every integer list p. □

4 Implementation and Experiments

We have implemented a prototype tool for automated verification of productivity of functional programs, and conducted experiments. Below we report on the implementation and experimental results in Sections 4.1 and 4.2 respectively.

4.1 Implementation

The tool takes a program written in the source language defined in Section 2.1 as an input, converts it to the corresponding program of the target language based on the type-based translation defined in Section 3, and then passes the resulting program to a backend termination checker for higher-order functional programs. The backend termination checker should be able to automatically prove the termination (or non-termination) of higher-order functional programs with integers and integer lists. Since we could not find such a tool (that can deal with both higher-order functions and integer lists), we have extended MuHFL [18], which can automatically prove the termination (or non-termination) of higher-order functional programs with only integers as base types, to handle higher-order functions on integer lists. Below we discuss this extension briefly.

Actually, MuHFL [18] is an automated validity checker for formulas of HFL($\mathbb{Z}$), a higher-order fixpoint logic extended with integer arithmetic. It is known [19, 25] that arbitrary regular temporal properties (i.e., properties expressible in the modal μ-calculus, including the termination and non-termination properties) of higher-order functional programs can be reduced to the validity checking problem (i.e., the problem of checking whether a given HFL($\mathbb{Z}$) formula is valid). Since MuHFL supported only integer data, we have extended MuHFL to directly[1] support integer lists, which is crucial since the termination property of our target programs heavily relies on the usage of integer lists.

Below we explain how MuHFL proves or disproves (a formula corresponding to) the termination of functional programs, and how we have extended it to deal with integer lists; the familiarity with the higher-order fixpoint logic HFL($\mathbb{Z}$) is not required. The idea used in MuHFL for proving the termination goes back to that of Fedyukovich et al. [12]. To show the termination of a function f that takes integer arguments $x_1, \ldots, x_k$, the tool picks some constants $c, d > 0$ and tries to prove a sufficient condition: "f terminates within $c(|x_1| + \cdots + |x_k|) + d$ nested recursive calls". Since the latter is a safety property, it can be solved by a backend safety property checker (in the case of MuHFL, a validity checker ReTHFL [17] for a subclass of HFL($\mathbb{Z}$)). If the property does not hold, MuHFL increases the values of c and d and retries to prove that f terminates within the given number of recursive calls. As this procedure will diverge when f actually does not terminate, in parallel to the procedure above, MuHFL also tries to disprove the termination of the given program by proving that the negation of the given formula is valid. (Note that the set of formulas treated by MuHFL is closed under negations; thus the same verification method can be used for proving termination and non-termination [18].)

To deal with integer lists, we have replaced the bound $c(|x_1| + \cdots + |x_k|) + d$ on nested recursive calls with $c(|x_1| + \cdots + |x_k| + \mathrm{length}(y_1) + \cdots + \mathrm{length}(y_m)) + d$, where y_i's are integer list arguments, and $\mathrm{length}(y_i)$ denotes the length of the list y_i. We have also extended the backend solver ReTHFL to deal with integer lists. ReTHFL uses a refinement type system to reduce the validity checking problem for the subclass of HFL($\mathbb{Z}$) (corresponding to safety properties) to the problem of CHC (Constrained Horn Clauses) solving. The extended version of ReTHFL now generates CHC on integer lists, which is solved by Eldarica [14].[2]

4.2 Experiments

We prepared a collection of programs which generate co-inductive data structures and ran our tools on them. We measured the total execution time of the translation and the termination checking. The experiments were conducted on a machine with Intel Core CPU i5-12500 with 32 GB of RAM. We set the timeout to 300 seconds.

The code of the test cases is given in Table 1. The column "Code" shows the definitions of co-inductive data structures. The column "Prod?" shows whether the test case is productive or not. Stream Tk (where $k = 1, 2, 4, \ldots$) is an example from [11] and productive if and only if k is even. The other test cases are our original. "Head" and "Tail" are head and tail operations of the streams. "nth" takes the n-th element of a stream. The operator :: is the constructor of streams. The operator (+) is the pointwise addition for streams of integers and trees of integers. "Node" is a constructor of infinite binary trees. "Right" and "Left" are destructors of infinite binary trees and return right and left children respectively. "Node3" is a constructor of infinite ternary trees, and "Mid" and "Left" are destructors of them. The test case "righttree" is the example in the introduction. The test case "lefttree" is similar to "righttree" but a non-productive stream. The test case "tritree" is productive by the same reason as "righttree". The test case "fibstream" is the stream of the Fibonacci numbers.

The experimental results are shown in Table 2. The column "Result" shows the result of our productivity checker. The column "Time" shows the execution time of our tool in seconds. The column "Sized type" shows whether each co-inductive data structure is typable in the sized type system [15] or not. We remark that the typability in sized type system and guarded recursion are equivalent for all the test cases.

Our tool successfully verified all the test cases except for T1 and T38. The timeouts for T1 and T38 are due to the current limitation of the backend solver MuHFL. For T38, the timeout occurs because large constants for c and d explained above are required. For T1, the HFL($\mathbb{Z}$) formula expressing

[1] It would be possible to encode an integer list as a function, but MuHFL would not work well for such encoding.

[2] We have also tried other CHC solvers like Spacer [20] and HoIce [7], but they did not work well for CHCs with integer lists.

Table 1. Code of the test cases

Test case	Code	Prod?
goodstream	s = 1 :: s	Yes
badstream	s = Head(s) :: Tail(s)	No
goodtree	t = Node(t, 1, t)	Yes
lefttree	t = Node(Left(t), 1, t)	No
righttree	t = Node(Right(t), 1, t)	Yes
tritree	t3 = Node3(0, t3, Left(t3), Mid(t3))	Yes
twostream	s = goodstream (+) goodstream	Yes
twotree	t = goodtree (+) goodtree	Yes
fibstream	fib = 0 :: 1 :: (fib (+) (tail fib))	Yes
T1	T1 = 0 :: nth(1, T1) :: T1	No
T2	T2 = 0 :: nth(2, T2) :: T2	Yes
T4	T4 = 0 :: nth(4, T4) :: T4	Yes
T36	T36 = 0 :: nth(36, T36) :: T36	Yes
T38	T38 = 0 :: nth(38, T38) :: T38	Yes

Table 2. Experimental results

Test case	Result	Time(s)	Sized type
goodstream	Productive	0.92	Typable
badstream	Non-productive	9.31	Untypable
goodtree	Productive	0.90	Typable
lefttree	Non-productive	15.23	Unytpable
righttree	Productive	3.48	Untypable
tritree	Productive	9.78	Untypable
twostream	Productive	6.92	Typable
twotree	Productive	11.58	Typable
fibstream	Productive	15.32	Typable
T1	Unknown	timeout	Untypable
T2	Productive	2.42	Untypable
T4	Productive	6.23	Untypable
T36	Productive	256.90	Untypable
T38	Unknown	timeout	Untypable

the non-productivity involves existential quantifiers on lists, for which the current implementation of MuHFL is slow (see also the discussion below).

For any of the test cases, the translation was done within 0.01 seconds. Thus, the total execution time was dominated by that of the termination checker. Our tool could also verify the productivity of trees that cannot be verified by the previous type-based methods such as the sized type system. For example, "righttree" and "tritree" are untypable in the sized type system. As observed in the table, our tool is generally slower for non-productive instances than for valid cases. This is because our extended MuHFL solver looks for a counterexample path in the increasing order of the path length. and repeatedly calls the backend $\nu\text{HFL}(\mathbb{Z})$ solver for each fixed length. This procedure leads to low performance when actual counterexample paths are long. The tool is also slow for programs that handle the operations on co-inductive data structures, such as the addition of streams. This problem may be mitigated by improving the backend CHC solver.

5 Related Work

Verification of productivity has been actively studied in the context of proof assistants, as it is critical for the consistency of the logic. A syntactic method for productivity is implemented in Coq. It is required in Coq that definitions of co-inductive data are syntactically guarded by constructors. Sized types and guarded recursion have been used for type-based verification of productivity. Sized types were introduced by Hughes et al. [15], and implemented in Agda by integrating sized types and dependent types [1]. Guarded recursion was introduced by Nakano [23]. He showed that the stream type can be expressed as $\mu Y.A \times \bullet Y$, and that a constant stream and the merge operator can be expressed in his type system. He also suggested more complicated programs over streams can be expressed by allowing annotations with arithmetic expressions, but automatic verification was out of scope. Atkey and McBride's clock quantifiers [4] allow acausal definitions of co-inductive data structures. Clouston et al. [8] allow productive and acausal definitions by introducing box operators instead of clock quantifiers. Veltri el al. [24] studied the relation between sized types and guarded recursion. To our knowledge, none of those systems can automatically prove the productivity of RightTree.

Endrullis et al. [10, 11] studied the productivity in the context of term rewriting systems (TRS), and showed a reduction from productivity to context-sensitive termination of a TRS. Their reduction is different from ours; Endrullis et al. [10, 11] augments the TRS with rules to retrieve each element in a non-deterministic manner, so that productivity is reduced to context-sensitive termination of the resulting TRS. Also, they do not treat higher-order functions (though, in principle, higher-order functions can be encoded using TRSs). Aguirre et al. [2] studied productivity in a probabilistic setting. Their approach can verify whether a stream definition produces infinite elements with probability 1. However, their language is not Turing complete. Ancona et al. [3] also studied runtime verification of productivity for streams. Their approach can verify the productivity of more stream instances than type-based verification, but it is still incomplete.

Other properties of co-inductive data structure are safety and liveness. Jaber and Riba [16] extended a guarded lambda calculus with refinement types, so that one can represent safety and liveness. Bahr et al. extended guarded recursion to treat liveness in FRP setting [6]. We expect that our approach can also be applied to those problems (of verifying safety/liveness of programs involving co-inductive data structures); we leave it for future work.

6 Conclusion

We have proposed a method for reducing productivity verification to termination verification for functional programs without co-inductive types, and proved its correctness. We have also implemented a prototype productivity checker based on our method. To that end, we have also extended a termination verification tool for higher-order programs to support lists. We have confirmed the effectiveness of our tool for several instances including those that cannot be handled by existing size-based methods.

Future work includes a further optimization of our productivity checker and an application of our method to real programming languages like Haskell. We also plan to extend our verification method to deal with other temporal properties on co-inductive data [16].

Acknowledgments

We would like to thank Ken Sakayori and anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP20H05703.

References

[1] Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. In *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010 (EPiC Series, Vol. 5)*, Ekaterina Komendantskaya, Ana Bove, and Milad Niqui (Eds.). EasyChair, 18–33. https://doi.org/10.29007/322q

[2] Alejandro Aguirre, Gilles Barthe, Justin Hsu, and Alexandra Silva. 2018. Almost Sure Productivity. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic (LIPIcs, Vol. 107)*, Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 113:1–113:15. https://doi.org/10.4230/LIPIcs.ICALP.2018.113

[3] Davide Ancona, Pietro Barbieri, and Elena Zucca. 2021. Enhanced Regular Corecursion for Data Streams. In *Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021 (CEUR Workshop Proceedings, Vol. 3072)*, Claudio Sacerdoti Coen and Ivano Salvo (Eds.). CEUR-WS.org, 266–280. https://ceur-ws.org/Vol-3072/paper22.pdf

[4] Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 197–208. https://doi.org/10.1145/2500365.2500597

[5] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The clocks are ticking: No more delays!. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005097

[6] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds are not forever: liveness in reactive programming with guarded recursion. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434283

[7] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2020. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. *J. Autom. Reason.* 64, 7 (2020), 1393–1418. https://doi.org/10.1007/s10817-020-09571-y

[8] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2016. The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types. *Log. Methods Comput. Sci.* 12, 3 (2016). https://doi.org/10.2168/LMCS-12(3:7)2016

[9] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 263–273. https://doi.org/10.1145/258948.258973

[10] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. 2010. Productivity of stream definitions. *Theor. Comput. Sci.* 411, 4-5 (2010), 765–782. https://doi.org/10.1016/j.tcs.2009.10.014

[11] Jörg Endrullis and Dimitri Hendriks. 2011. Lazy productivity via termination. *Theor. Comput. Sci.* 412, 28 (2011), 3203–3225. https://doi.org/10.1016/j.tcs.2011.03.024

[12] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. 2018. Syntax-Guided Termination Analysis. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (LNCS, Vol. 10981)*. Springer, 124–143. https://doi.org/10.1007/978-3-319-96145-3_7

[13] Adrien Guatto. 2018. A Generalized Modality for Recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 482–491. https://doi.org/10.1145/3209108.3209148

[14] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj S. Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–7. https://doi.org/10.23919/FMCAD.2018.8603013

[15] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 410–423. https://doi.org/10.1145/237721.240882

[16] Guilhem Jaber and Colin Riba. 2021. Temporal Refinements for Guarded Recursive Types. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 548–578. https://doi.org/10.1007/978-3-030-72019-3_20

[17] Hiroyuki Katsura, Naoki Iwayama, Naoki Kobayashi, and Takeshi Tsukada. 2020. A New Refinement Type System for Automated νHFL_Z Validity Checking. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, 86–104. https://doi.org/10.1007/978-3-030-64437-6_5

[18] Naoki Kobayashi, Kento Tanahashi, Ryosuke Sato, and Takeshi Tsukada. 2023. HFL(Z) Validity Checking for Automated Program Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 154–184. https://doi.org/10.1145/3571199

[19] Naoki Kobayashi, Takeshi Tsukada, and Keiichi Watanabe. 2018. Higher-Order Program Verification via HFL Model Checking. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 711–738. https://doi.org/10.1007/978-3-

319-89884-1_25

[20] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods Syst. Des.* 48, 3 (2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4

[21] Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 392–411. https://doi.org/10.1007/978-3-642-54833-8_21

[22] Rasmus Ejlers Møgelberg. 2014. A type theory for productive co-programming via guarded recursion. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 71:1–71:10. https://doi.org/10.1145/2603088.2603132

[23] Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 255–266. https://doi.org/10.1109/LICS.2000.855774

[24] Niccolò Veltri and Niels van der Weide. 2019. Guarded Recursion in Agda via Sized Types. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPIcs.FSCD.2019.32

[25] Keiichi Watanabe, Takeshi Tsukada, Hiroki Oshikawa, and Naoki Kobayashi. 2019. Reduction from branching-time property verification of higher-order programs to HFL validity checking. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, Manuel V. Hermenegildo and Atsushi Igarashi (Eds.). ACM, 22–34. https://doi.org/10.1145/3294032.3294077

A Proofs for Section 3

This section provides detailed proofs omitted in Section 3.

A.1 Basic Properties

The following lemmas hold regarding reduction and substitution.

Lemma A.1. *$N \longrightarrow_e N'$, then $[P/x]N \longrightarrow_e [P/x]N'$.*

Proof. This follows by induction on the structure of the evaluation context of $N \longrightarrow_e N'$. □

Lemma A.2. *$N \geq N'$, then $[P/x]N \geq^* [P/x]N'$.*

Proof. This follows by induction on the structure of the context of $N \geq N'$. □

Lemma A.3. *$P \geq P'$, then $[P/x]N \geq^* [P'/x]N$.*

Proof. By induction on the structure of N. □

Our translation rules holds inversion lemma.

Lemma A.4 (Inversion)**.** *Suppose $\Gamma \vdash M : \tau \rightsquigarrow N$ the following statements hold.*

1. *If $M = \lambda x.M_0$, then $\Gamma, x{:} = \tau_1 \vdash M : \tau_2 \rightsquigarrow N_0$ with $N = \lambda x.N_0$.*
2. *If $M = M_0\, M_1$, then $\Gamma \vdash M_0 : \tau_1 \to \tau_2 \rightsquigarrow N_0$ and $\Gamma \vdash M_1 : \tau_1 \rightsquigarrow N_1$ with $N = N_0\, N_1$ for some τ_1, τ_2.*
3. *If $M = (M_0, M_1)$, then $\Gamma \vdash M_0 : \sigma_0 \rightsquigarrow N_0$ and $\Gamma \vdash M_1\sigma_1 \rightsquigarrow N_1$ with $N = \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ 0 :: u' \Rightarrow N_0\, u' \mid 1 :: u' \Rightarrow N_1\, u' \mid \_ \Rightarrow 0$ for some σ_1, σ_2.*
4. *If $M = \pi_i(M')$, then $\Gamma \vdash M' : \sigma \rightsquigarrow N'$ with $N = \Lambda u.N'\ (i :: u)$ for some σ.*
5. *If $M = in_i(M')$, then $\Gamma \vdash M' : \sigma \rightsquigarrow N'$ with $N = \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ [\,] \Rightarrow i \mid 0 :: u' \Rightarrow N''\, u' \mid \_ \Rightarrow 0$ for some σ.*
6. *If $M = \mathtt{case}\ M'\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1)$, then $\Gamma \vdash M' : \sigma_0 + \sigma_1 \rightsquigarrow N'$ $\Gamma, x : {:}\sigma_0 \vdash M_0 : \tau' \rightsquigarrow N_0$, and $\Gamma, x : \sigma_1 \vdash M_1 : \tau' \rightsquigarrow N_1$ with $N = \mathtt{if0}\ N'\ [\,]\ \mathtt{then}\ [\lambda u.N'\ (0 :: u)/x]N_0\ \mathtt{else}\ [\lambda u.N'\ (0 :: u)/x]N_1$ for some τ'.*

Proof. The proof proceeds by induction on derivation of $\Gamma \vdash M : \tau \rightsquigarrow N$, with case analysis on the last rule used. If the rule is Tr-Fold or Tr-Unfold, the required condition holds directly from the induction hypothesis. Otherwise, the rule is uniquely determined from the definition of the translation, and the required condition holds by the form of the rule. □

A.2 Proof of Lemma 3.5

Proof of Lemma 3.5. This follows by induction on the derivation of $\Gamma, x : \tau_1 \vdash M_0 : \tau \rightsquigarrow N_0$.

- Case Tr-Var: If $M_0 = x$, we get $[M_1/x]M_0 = M_1$ and the required condition holds by assumption. If $M_0 = y$, we get $[M_1/x]M_0 = M_0$ and the required condition holds by assumption.
- Case Tr-Num, Tr-Z: In this case, we get $[M_1/x]M_0 = M_0$ and the required condition holds by assumption.
- Case Tr-Abs: In this case, we get $M_0 = \lambda y.M_0'$ and $N_0 = \lambda y.N_0'$ where $\Gamma, x : \tau_1 \vdash M_0' : \tau \rightsquigarrow N_0'$. By the induction hypothesis, we get $\Gamma \vdash [M_1/x]M_0' : \tau \rightsquigarrow [N_1/x]N_0'$. By Tr-Abs, we get $\Gamma \vdash [M_1/x]\lambda y.M_0' : \tau \rightsquigarrow [N_1/x]\lambda y.N_0'$.
- Case Tr-App: In this case, we get $M_0 = M_0'\, M_0''$ and $N_0 = N_0'\, N_0''$ where $\Gamma, x : \tau_1 \vdash M_0' : \tau \rightsquigarrow N_0'$ and $\Gamma, x : \tau_1 \vdash M_0'' : \tau \rightsquigarrow N_0''$. By the induction hypothesis, we get $\Gamma \vdash [M_1/x]M_0' : \tau \rightsquigarrow [N_1/x]N_0'$ and $\Gamma \vdash [M_1/x]M_0'' : \tau \rightsquigarrow [N_1/x]N_0''$. By Tr-App we get $\Gamma \vdash [M_1/x]M_0'\, M_0'' : \tau \rightsquigarrow [N_1/x]N_0'\, N_0''$.
- Case Tr-Fix: In this case, we get $M_0 = \mathtt{fix}\ y.M_0'$ and $N_0 = \mathtt{fix}\ y.N_0'$ where $\Gamma, x : \tau_1 \vdash M_0' : \tau \rightsquigarrow N_0'$. By the induction hypothesis, we get $\Gamma \vdash [M_1/x]M_0' : \tau \rightsquigarrow [N_1/x]N_0'$. By Tr-Fix, we get $\Gamma \vdash [M_1/x]\mathtt{fix}\ y.M_0' : \tau \rightsquigarrow [N_1/x]\mathtt{fix}\ y.N_0'$.
- Case Tr-Pair: In this case, we get $M_0 = (M_0', M_0'')$ and $N_0 = \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ 0 :: u' \Rightarrow N_0\, u' \mid 1 :: u' \Rightarrow N_1\, u' \mid \_ \Rightarrow 0$ where $\Gamma, x : \sigma_0 \vdash M_0' : \tau \rightsquigarrow N_0'$ and $\Gamma, x : \sigma_1 \vdash M_0'' : \tau \rightsquigarrow N_0''$. By the induction hypothesis, we get $\Gamma \vdash [M_1/x]M_0' : \sigma_0 \rightsquigarrow [N_1/x]N_0'$ and $\Gamma \vdash [M_1/x]M_0'' : \sigma_1 \rightsquigarrow [N_1/x]N_0''$. By Tr-Pair we get $\Gamma \vdash [M_1/x](M_0', M_0'') : \sigma_0 \times \sigma_1 \rightsquigarrow \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ 0 :: u' \Rightarrow [N_1/x]N_0'\, u' \mid 1 :: u' \Rightarrow [N_1/x]N_0''\, u' \mid \_ \Rightarrow 0$.
- Case Tr-Proj: In this case, we get $M_0 = \pi_i(M_0')$ and $N_0 = \Lambda u.N_0'(i :: u)$ where $\Gamma, x : \tau_1 \vdash M_0' : \sigma_0 \times \sigma_1 \rightsquigarrow N_0'$. By the induction hypothesis, we get $\Gamma \vdash [M_1/x]M_0' : \sigma_0 \times \sigma_1 \rightsquigarrow [N_1/x]N_0'$. By Tr-Proj, we get $\Gamma \vdash [M_1/x]\pi_i(M_0') : \tau \rightsquigarrow \Lambda u.[N_1/x]N_0'(i :: u)$.
- Case Tr-Inj: In this case, we get $M_0 = in_i(M_0')$ and $N_0 = \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ [\,] \Rightarrow i \mid 0 :: u' \Rightarrow N_0'\, u' \mid \_ \Rightarrow 0$ where $\Gamma, x : \tau_1 \vdash M_0' : \sigma_i \rightsquigarrow N_0'$. By the induction hypothesis, we get $\Gamma \vdash [M_1/x]M_0' : \sigma_i \rightsquigarrow [N_1/x]N_0'$. By Tr-Inj, we get $\Gamma \vdash [M_1/x]M_0 : \tau \rightsquigarrow [N_1/x]N_0$.
- Case Tr-Case: In this case, we get $M_0 = \mathtt{case}\ M\ \mathtt{of}\ (in_0(x) \Rightarrow M_0' \mid in_1(x) \Rightarrow M_0'')$ and $N_0 = \mathtt{if0}\ N\,[\,]\ \mathtt{then}\ [\lambda u.N\,(0 :: u)/y]N_0'\ \mathtt{else}\ [\lambda u.N\,(0 :: u)/y]N_0''$ where $\Gamma, x : \tau_1 \vdash M : \sigma_0 + \sigma_1 \rightsquigarrow N$, $\Gamma, x : \tau_1 \vdash M_0' : \sigma_0 \rightsquigarrow N_0'$, and $\Gamma, x : \tau_1 \vdash M_0'' : \sigma_1 \rightsquigarrow N_0''$. By the induction hypothesis, we get $\Gamma \vdash [M_1/x]M : \sigma_0 + \sigma_1 \rightsquigarrow [N_1/x]N$, $\Gamma \vdash [M_1/x]M_0' : \sigma_0 \rightsquigarrow [N_1/x]N_0'$, and $\Gamma \vdash [M_1/x]M_0'' : \sigma_1 \rightsquigarrow [N_1/x]N_0''$. By Tr-Case, we get $\Gamma \vdash [M_1/x]M_0 : \tau \rightsquigarrow [N_1/x]N_0$.
- Case Tr-Fold, Tr-Unfold: In this case, the required condition holds directly from the induction hypothesis.

□

Note that the translation is deterministic even though its derivation is not unique due to the rules of fold and unfold of the fixpoint ν.

A.3 Proof of Lemma 3.9

We first show the equivalence of the conditions 1 and 2 of Lemma 3.9.

Lemma A.5. *Let N be a term of base type. Then N is terminating with respect to $\longrightarrow_e$ if and only if it is terminating with respect to $\longrightarrow$.*

Proof. The "only if" direction is trivial as $\longrightarrow \subseteq \longrightarrow_e$. To show the "if" direction, suppose $N \longrightarrow {}^n V$. We prove that any term N' that satisfies $N \longrightarrow_e N'$ is terminating with respect to $\longrightarrow_e$ by induction on n. If $N \longrightarrow N'$, then N' is terminating by the induction hypothesis, since $\longrightarrow$ is deterministic. Otherwise, the reduction $N \longrightarrow N'$ must occur under Λx, so N and N' must be of the form:

$$N = \mathcal{E}'[\Lambda x.P]$$
$$N' = \mathcal{E}'[\Lambda x.P']$$

where $P \longrightarrow_e P'$ and $\mathcal{E}'$ is a normal evaluation context. Since N is a term of base type and $\Lambda x.P$ has a function type, the hole of $\mathcal{E}'$ must occur in an application $\langle\,\rangle\, Q$. Thus, $\mathcal{E}'$ is of the form $\mathcal{E}[\langle\,\rangle\, Q]$, and we have $N = \mathcal{E}[(\Lambda x.P)Q]$ and $N' = \mathcal{E}[(\Lambda x.P')Q]$. We can also get

$$N \longrightarrow N'' = \mathcal{E}[[Q/x]P] \longrightarrow {}^{n-1} V$$

Hence, N'' is terminating with respect to $\longrightarrow_e$ by the induction hypothesis. If there is an infinite reduction sequence in the form:

$$\begin{aligned} N' &\longrightarrow_e \mathcal{E}[(\Lambda x.P_1)\, Q] \\ &\longrightarrow_e \mathcal{E}[(\Lambda x.P_2)\, Q] \\ &\longrightarrow_e \cdots . \end{aligned}$$

By Lemma A.1 we get follows infinite reduction sequence:

$$\begin{aligned} N'' &\longrightarrow_e \mathcal{E}[[Q/x]P'] \\ &\longrightarrow_e \mathcal{E}[[Q/x]P_1] \\ &\longrightarrow_e \cdots . \end{aligned}$$

But it contradicts the fact that N'' is terminating. Thus, any reduction sequence starts from N' is in the form

$$\begin{aligned} N' &\longrightarrow_e^* \mathcal{E}[(\Lambda x.P_k)\, Q] \\ &\longrightarrow_e \mathcal{E}[[Q/x]P_k] \\ &\longrightarrow_e \cdots . \end{aligned}$$

Thus, it suffices to show $\mathcal{E}[[Q/x]P_k]$ is terminating, which follows immediately from

$$N'' \longrightarrow_e^* \mathcal{E}[[Q/x]P_k]$$

and the fact that N'' is terminating. □

To show the equivalence between the conditions 1 and 3 of Lemma 3.9, we prepare a few lemmas.

Definition A.6. We call N is normal-terminating if N is terminating with respect to $\longrightarrow$.

Lemma A.7. *Let N be a term of base type. If $N \longrightarrow_e N_1$ and $N \longrightarrow N_2$ and $N_1 \neq N_2$, then N, N_1, and N_2 must be of the form*

$$N = \mathcal{E}[(\Lambda x.P)\, Q]$$
$$N_1 = \mathcal{E}[(\Lambda x.P')\, Q]$$
$$N_2 = \mathcal{E}[[Q/x]P]$$

with $P \longrightarrow_e P'$.

Proof. Since $N \longrightarrow N_1$ does not hold, the context of $N \longrightarrow_e N_1$ is of the form:

$$\mathcal{E}[(\Lambda x.\mathcal{E}_e[])\, Q]$$

Then, N, N_1, N_2 must be of the form

$$N = \mathcal{E}[(\Lambda x.P)\, Q]$$
$$N_1 = \mathcal{E}[(\Lambda x.P')\, Q]$$
$$N_2 = \mathcal{E}[[Q/x]P]$$

with $P \longrightarrow_e P'$. □

Lemma A.8. *Let N be a term of base type. If $N \longrightarrow_e N_1$ and $N \longrightarrow N_2$, then either $N_1 = N_2$ or there exists N_3 such that $N_1 \longrightarrow N_3$ and $N_2 \longrightarrow_e N_3$.*

Proof. If $N_1 \neq N_2$, by Lemma A.7, N, N_1, and N_2 must be of the form.

$$N = \mathcal{E}[(\Lambda x.P)\, Q]$$
$$N_1 = \mathcal{E}[(\Lambda x.P')\, Q]$$
$$N_2 = \mathcal{E}[[Q/x]P]$$

with $P \longrightarrow_e P'$. By Lemma A.1, the required condition holds for $N_3 = [Q/x]P'$. □

We can now show the equivalence between the conditions 1 and 3.

Lemma A.9. *Let N be a term of base type. If N is may-terminating with respect to $\longrightarrow_e$, then N is normal-terminating.*

Proof. Suppose $N \longrightarrow_e^n V$. This follows by induction on n. If $n = 0$, the condition trivially holds. We suppuse $N \longrightarrow_e N' \longrightarrow_e^{n-1} V$ and $N \longrightarrow N_1$. By induction hypothesis, N' is normal-terminating. If $N \longrightarrow N'$, N is normal-terminating by assumption. Otherwise, suppose N is not normal-terminating. Then, we get an inifinite reduction sequence from N. By Lemma A.8, we can construct infinite reduction sequence from N'. However, it contradicts the fact that N' is normal-terminating. Thus, the required condition holds. □

Proof of Lemma 3.9. By Lemmas A.5 and Lemma A.9. □

Below we will call termination with respect to $\longrightarrow_e$ and that of $\longrightarrow$ simply "termination".

A.4 Proof of Lemma 3.11

Proof of Lemma 3.11. This follows by induction on the length of γ. The case $\gamma = \epsilon$ is trivial. If M is non-terminating, f_M is also non-terminating by Lemmas 3.12 and 3.13. It contradicts the assumption. We suppose $M \longrightarrow^* V$ for some value V. We perform case analysis on σ. Since M is a closed term, σ is not type variable. If $\sigma = \mathtt{int}$, $\gamma \in \mathbf{Path}(\sigma)$ if and only if $\gamma = \epsilon$. Thus, the required condition trivially holds. If $\sigma = \nu X.\sigma'$ for some σ', we unfold the fixpoint. Since σ' is not type variable, the unfolded type is not fixpoint type. Thus, it suffices to show the case σ is product or sum type.

- Case $\sigma = \sigma_0 \times \sigma_1$: We first consider the case $\gamma = \pi_0 \cdot \gamma'$. For any p, we get

  $$f_{\pi_0(M)}\, p = (\Lambda u.f_M(0 :: u))\, p \longrightarrow_e f_M\,(0 :: p).$$

  Since $f_M\,(0 :: p)$ is terminating by assumption, $f_{\pi_0(M)}\, p$ is also terminating. By induction hypothesis, we get $f_{\gamma M}\, q = f_{\gamma'(\pi_0(M))}\, q$ is also terminating. The case $\gamma = \pi_1 \cdot \gamma'$ similarly holds. Otherwise, $\gamma \notin \mathbf{Path}(\sigma)$.
- Case $\sigma = \sigma_0 + \sigma_1$: We first consider the case $\gamma = in_0^{-1} \cdot \gamma'$. Since V is value, $V = in_i(M')$ for some i and M'. If $i = 0$, we get $f_V\,[\,] \longrightarrow_e^* 0$. By Lemma 3.16, we get $f_M\,[\,] \longrightarrow_e^* 0$ and

  $$\begin{aligned}
  &f_{\mathsf{case}\ M\ \mathsf{of}\ (in_0(x) \Rightarrow \gamma\, x \mid in_1(x) \Rightarrow \mathrm{Z})}\, p \\
  &= (\mathtt{if0}\ f_M\,[\,]\ \mathtt{then}\ \Lambda u.f_M\,(0 :: u)\ \mathtt{else}\ \Lambda u.0)\, p \\
  &\longrightarrow_e^* (\mathtt{if0}\ 0\ \mathtt{then}\ \Lambda u.f_M\,(0 :: u)\ \mathtt{else}\ \Lambda u.0)\, p \\
  &\longrightarrow_e (\Lambda u.f_M\,(0 :: u))\, p \\
  &\longrightarrow_e f_M\,(0 :: p)
  \end{aligned}$$

  for any p. Since $f_M\,(0 :: p)$ is terminating by assumption, $f_{\mathsf{case}\ M\ \mathsf{of}\ (in_0(x) \Rightarrow \gamma\, x \mid in_1(x) \Rightarrow \mathrm{Z})}\, p$ is also terminating. By induction hypothesis,

  $$f_{\gamma M}\, q = f_{\gamma'(\mathsf{case}\ M\ \mathsf{of}\ (in_0(x) \Rightarrow \gamma\, x \mid in_1(x) \Rightarrow \mathrm{Z}))}\, q$$

  is terminating. If $i = 1$, we get $f_V\,[\,] \longrightarrow_e^* 1$. By Lemma 3.16, we get $f_M\,[\,] \longrightarrow_e^* 1$ and

  $$\begin{aligned}
  &f_{\mathsf{case}\ M\ \mathsf{of}\ (in_0(x) \Rightarrow \gamma\, x \mid in_1(x) \Rightarrow \mathrm{Z})}\, p \\
  &= (\mathtt{if0}\ f_M\,[\,]\ \mathtt{then}\ \Lambda u.f_M\,(0 :: u)\ \mathtt{else}\ \Lambda u.0)\, p \\
  &\longrightarrow_e^* (\mathtt{if0}\ 1\ \mathtt{then}\ \Lambda u.f_M\,(0 :: u)\ \mathtt{else}\ \Lambda u.0)\, p \\
  &\longrightarrow_e (\Lambda u.0)\, p \\
  &\longrightarrow_e 0
  \end{aligned}$$

  for any p. By induction hypothesis,

  $$f_{\gamma M}\, q = f_{\gamma'(\mathsf{case}\ M\ \mathsf{of}\ (in_0(x) \Rightarrow \gamma\, x \mid in_1(x) \Rightarrow \mathrm{Z}))}\, q$$

  is terminating. The case $\gamma = in_1^{-1} \cdot \gamma'$ similarly holds. Otherwise, $\gamma \notin \mathbf{Path}(\sigma)$.

□

A.5 Proof of Lemma 3.12

Proof of Lemma 3.12. The proof proceeds by induction on the derivation of $M \longrightarrow_e M'$, with case analysis on the last rule used.

- Case R-Beta: In this case, $M = (\lambda x.M_0)M_1$ and $M' = [M_1/x]M_0$. By Lemma A.4, we have $x : \tau_1 \vdash M_0 : \tau \rightsquigarrow N_0$ and $\vdash M_1 : \tau_1 \rightsquigarrow N_1$ with $N = (\lambda x.N_0)N_1$. By Lemma 3.5, we have $\vdash [M_1/x]M_0 : \tau \rightsquigarrow [N_1/x]N_0$. Thus, the required condition holds for $N' = [N_1/x]N_0$.
- Case R-Proj: In this case, $M = \pi_i(M_0, M_1)$ and $M' = M_i$. By Lemma A.4, we have $\vdash M_0 : \sigma_0 \rightsquigarrow N_0$ and $\vdash M_1 : \sigma_1 \rightsquigarrow N_1$ with $\tau = \sigma_i$ and $N = \Lambda u.(\Lambda v.\mathtt{case}\ v\ \mathtt{of}\ 0 :: u' \Rightarrow N_0\, u' \mid 1 :: u' \Rightarrow N_1\, u' \mid \_ \Rightarrow 0)(i :: u)$. For $i \in \{0, 1\}$, we have:

  $$\begin{aligned}
  &N \\
  &= \Lambda u.(\Lambda v.\mathtt{case}\ v\ \mathtt{of}\ 0 :: u' \Rightarrow N_0\, u' \\
  &\qquad\qquad \mid 1 :: u' \Rightarrow N_1\, u' \mid _ \Rightarrow 0)(i :: v) \\
  &\longrightarrow_e \Lambda u.\mathtt{case}\ i :: u\ \mathtt{of}\ 0 :: u' \Rightarrow N_0\, u' \mid 1 :: u' \Rightarrow N_1\, u' \mid _ \Rightarrow 0 \\
  &\geq \Lambda u.N_i\, u \\
  &\geq N_i u.
  \end{aligned}$$

  Thus, the required condition holds for $N' = N_i$.
- Case R-Case: In this case, $M = \mathtt{case}\ in_i(M_2)\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1)$ and $M' = [M_2/x]M_i$. By Lemma A.4, we have $\vdash M_2 : \sigma_i \rightsquigarrow N_2$ and $x : \sigma_j \vdash M_j : \tau \rightsquigarrow N_j$ for $j \in \{0, 1\}$, with $N = \mathtt{if0}\ N_3[\,]\ \mathtt{then}\ N_0'\ \mathtt{else}\ N_1'$ where $N_3 = \Lambda u.\mathtt{case}\ u\ \mathtt{of}\ [\,] \Rightarrow i \mid 0 :: u' \Rightarrow N_2\, u' \mid \_ \Rightarrow 0$, $N_0' = [\Lambda u.N_3(0 :: u)/x]N_0$, and $N_1' = [\Lambda u.N_3(0 :: u)/x]N_1$. By Lemma A.3, for $i \in \{0, 1\}$, N can be reduced as follows.

  $$\begin{aligned}
  N &= \mathtt{if0}\ N_3[\,]\ \mathtt{then}\ N_0'\ \mathtt{else}\ N_1' \\
  &\longrightarrow_e \mathtt{if0}\ i\ \mathtt{then}\ N_0'\ \mathtt{else}\ N_1' \\
  &\longrightarrow_e [\Lambda u.N_3(0 :: u)/x]N_i \\
  &\geq^* [\Lambda u.\mathtt{case}\ 0 :: u\ \mathtt{of}\ [\,] \Rightarrow i \mid 0 :: u' \Rightarrow N_2\, u' \mid _ \Rightarrow 0/x]N_i \\
  &\geq^* [\Lambda u.N_2\, u/x]N_i \\
  &\geq^* [N_2/x]N_i.
  \end{aligned}$$

  By Lemma 3.5, we also have $\vdash [M_2/x]M_i : \tau \rightsquigarrow [N_2/x]N_i$. Thus, $N' = [N_2/x]N_i$ satisfies the required condition.
- Case R-Fix: In this case, $M = \mathtt{fix}\ x.M_0$ and $M' = [\mathtt{fix}\ x.M_0/x]M_0$. By Lemma A.4, we have $x : \tau \vdash M_0 : \tau \rightsquigarrow N_0$ with $N = \mathtt{fix}\ x.N_0$. By Lemma 3.5, we also have $\vdash [\mathtt{fix}\ x.M_0/x]M_0 : \tau \rightsquigarrow [\mathtt{fix}\ x.N_0/x]N_0$. Thus, the required condition holds for $N' = [\mathtt{fix}\ x.N_0/x]N_0$.
- Case R-CApp: In this case, $M = M_0M_1$ and $M' = M_0'M_1$. By Lemma A.4, we have $\vdash M_0 : \tau_2 \to \tau \rightsquigarrow N_0$ and $\vdash M_1 : \tau_2 \rightsquigarrow N_1$ with $N = N_0N_1$. By the induction

hypothesis, we also have $\vdash M_0' : \tau \rightsquigarrow N_0'$ and $N_0 \longrightarrow_e^+ \geq^* N_0'$. Thus, the required condition holds for $N' = N_0' N_1$.

- Case R-CProj: In this case, $M = \pi_i(M_0)$ and $M' = \pi_i(M_0')$. By Lemma A.4, we have $\vdash M_0 : \tau \rightsquigarrow N_0$ with $N = \Lambda u.N_0(i :: u)$. By the induction hypothesis, we also have $\vdash M_0 : \tau \rightsquigarrow N_0$ and $N_0 \longrightarrow_e^+ \geq^* N_0'$. Thus, the required condition holds for $N' = \Lambda u.N_0'(i :: u)$.
- Case R-CCase: In this case, $M = \mathtt{case}\ M_2\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1)$ and $M' = \mathtt{case}\ M_2'\ \mathtt{of}\ (in_0(x) \Rightarrow M_0 \mid in_1(x) \Rightarrow M_1)$. By the assumption $\vdash M : \tau \rightsquigarrow N$, we have $\vdash M_2 : \tau \rightsquigarrow N_2$ with

$$N = \mathtt{if0}\ N_2\ [\,]\ \mathtt{then}\ N_0'\ \mathtt{else}\ N_1',$$
$$N_0' = [\lambda u.N_2\ (0 :: u)/x]N_0,$$
$$N_1' = [\lambda u.N_2\ (0 :: u)/x]N_1.$$

  By the induction hypothesis, we also have $\vdash M_2 : \tau \rightsquigarrow N_2$ and $N_2 \longrightarrow_e^+ \geq^* N_2'$. N can be reduced as follows.

$$\begin{aligned} N &= \mathtt{if0}\ N_2\ [\,]\ \mathtt{then}\ N_0'\ \mathtt{else}\ N_1' \\ &\longrightarrow_e^+ \geq^* \mathtt{if0}\ N_2'\ [\,]\ \mathtt{then}\ N_0'\ \mathtt{else}\ N_1' \\ &\geq^* \mathtt{if0}\ N_2'\ [\,]\ \mathtt{then}\ [\lambda u.N_2'\ (0 :: u)/x]N_0\ \mathtt{else} \\ &\qquad [\lambda u.N_2'\ (0 :: u)/x]N_1 \end{aligned}$$

  Thus, the required condition holds for $N' = \mathtt{if0}\ N_2'\ [\,]\ \mathtt{then}\ [\lambda u.N_2'\ (0 :: u)/x]N_0\ \mathtt{else}\ [\lambda u.N_2'\ (0 :: u)/x]N_1$.

□

A.6 Proof of Lemma 3.13

Proof of Lemma 3.13. Suppose $N \geq N'' \longrightarrow_e N'$. The proof proceeds by case analysis on the context of C of the $N \geq N''$. If C is an evaluation context $\mathcal{E}_e$, then we get $N = \mathcal{E}_e[M_0] \geq \mathcal{E}_e[M_1] = N'' \longrightarrow_e N'$, and the required condition holds for each derivation rule of $\geq$ as follows:

- Case C-Eta In this case, we get $M_0 = \Lambda x.M_1\ x$. N can be reduced as follows.

$$N = \mathcal{E}_e[\Lambda x.M_1\ x] \longrightarrow_e \Lambda x.N'\ x \geq N'$$

- Case C-Beta In this case, we get $\mathcal{E}_e[M_0] \longrightarrow_e \mathcal{E}_e[M_1]$ by definition. Thus, $N \longrightarrow_e N'' \longrightarrow_e N'$ holds.

We now prove the other cases.

- Case $N = \lambda x.P \geq \lambda x.P' = N''$ where $P \geq P'$: In this case, N'' cannot be reduced to some N'. Thus, the required condition holds.
- Case $N = (\lambda x.P)\,Q \geq (\lambda x.P)\,Q'$ where $Q \geq Q'$: In this case, we get $N' = [Q'/x]P$. By Lemma A.2 N can be reduced as follows.

$$N = (\lambda x.P)\,Q \longrightarrow_e [Q/x]P \geq^* [Q'/x]P = N'$$

  Thus, the required condition holds.

- Case $N = \mathtt{fix}\ x.P \geq \mathtt{fix}\ x.P' = N''$ where $P \geq P'$: In this case, we get $N' = [\mathtt{fix}\ x.P'/x]P'$. By Lemma A.2 and A.3 N can be reduced as follows.

$$\begin{aligned} N = \mathtt{fix}\ x.P &\longrightarrow_e [\mathtt{fix}\ x.P/x]P \\ &\geq [\mathtt{fix}\ x.P/x]P' \\ &\geq^* [\mathtt{fix}\ x.P'/x]P' = N' \end{aligned}$$

  Thus, the required condition holds.
- Case $N = \mathtt{if0}\ N_0\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2$ and $N'' = \mathtt{if0}\ N_0\ \mathtt{then}\ N_1'\ \mathtt{else}\ N_2$ where $N_1 \geq N_1'$: If $N_0 = 0$, we get $N' = N_1$ and $N \longrightarrow_e N_1 \geq N_1 = N''$ by assumption. If $N_0 = n$ where $n \neq 0$, we get $N' = N_2$ and $N \longrightarrow_e N_2 = N'$. Otherwise, we can suppose $N' = \mathtt{if0}\ N_0'\ \mathtt{then}\ N_1'\ \mathtt{else}\ N_2$ where $N_0 \longrightarrow_e N_0'$. We get $N \longrightarrow_e \mathtt{if0}\ N_0\ \mathtt{then}\ N_1'\ \mathtt{else}\ N_2 \geq N'$.
- Case $N = \mathtt{if0}\ N_0\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2$ and $N'' = \mathtt{if0}\ N_0\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2'$ where $N_2 \geq N_2'$: The required condition holds as well as the case above.
- Case $N = \mathtt{case}\ N_0\ \mathtt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)$ and $N'' = \mathtt{case}\ N_0\ \mathtt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_i \Rightarrow N_i' \mid \cdots \mid pat_k \Rightarrow N_k)$ where $N_i \geq N_i'$: We first consider the case where N_0 is a value. If $N_0 \vdash pat_i\rho$, we get $N' = \rho N_i'$ and $N \longrightarrow_e \rho N_i \geq^* \rho N_i'$ by Lemma A.2. If $N_0 \vdash pat_j\rho'$ where $i \neq j$, we get $N' = \rho' N_j$ and $N \longrightarrow_e N'$. We next consider the case where N_0 is not a value. We can suppose $N' = \mathtt{case}\ N_0'\ \mathtt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_i \Rightarrow N_i' \mid \cdots \mid pat_k \Rightarrow N_k)$ where $N_0 \longrightarrow_e N_0'$. We get $N \longrightarrow_e \mathtt{case}\ N_0'\ \mathtt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_i \Rightarrow N_i' \mid \cdots \mid pat_k \Rightarrow N_k) \geq N'$.

□

A.7 Proof of Lemma 3.15

To show Lemma 3.15, we introduce a new strong reduction relation $\geq_p^{(n)}$, which is a "parallel reduction" version of $\geq$, and show that $\geq_p^{(n)}$ commutes with $\longrightarrow_e$ (Lemma A.17 given later).

Definition A.10. We will denote the number of occurrence of x in N by $Oc(x, N)$.

Definition A.11. We define weighted strong parallel reduction $\geq_p^{(n)}$ as follows:

$$N \geq_p^{(0)} N \qquad \text{(SP-Id)}$$

$$\frac{N \longrightarrow_e N'}{N \geq_p^{(1)} N'} \qquad \text{(SP-Beta)}$$

$$\frac{x \text{ is not free in } N \qquad N \geq_p^{(n)} N'}{\Lambda x.N\ x \geq_p^{(n+1)} N'} \qquad \text{(SP-Eta)}$$

$$\frac{N \geq_p^{(n)} N'}{m :: N \geq_p^{(n)} m :: N'} \qquad \text{(SP-Cons)}$$

$$\frac{N \succeq_p^{(n)} N'}{\lambda x.N \succeq_p^{(n)} \lambda x.N'} \quad \text{(SP-Abs)}$$

$$\frac{N \succeq_p^{(n)} N'}{\Lambda x.N \succeq_p^{(n)} \Lambda x.N'} \quad \text{(SP-Abs2)}$$

$$\frac{N_0 \succeq_p^{(n)} N_0' \qquad N_1 \succeq_p^{(m)} N_1'}{N_0\ N_1 \succeq_p^{(n+m)} N_0'\ N_1'} \quad \text{(SP-App)}$$

$$\frac{N_0 \succeq_p^{(n)} N_0' \qquad N_1 \succeq_p^{(m)} N_1' \qquad N_2 \succeq_p^{(\ell)} N_2'}{\texttt{if0}\ N_0\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \succeq_p^{(n+m+\ell)} \texttt{if0}\ N_0'\ \texttt{then}\ N_1'\ \texttt{else}\ N_2'} \quad \text{(SP-If0)}$$

$$\frac{N_i \succeq_p^{(n_i)} N_i \quad (\text{for } i \in \{0,\dots,k\})}{\begin{array}{l}\texttt{case}\ N_0\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \\ \succeq_p^{(\sum_0^k n_i)} \texttt{case}\ N_0'\ \texttt{of}\ (pat_1 \Rightarrow N_1' \mid \cdots \mid pat_k \Rightarrow N_k')\end{array}} \quad \text{(SP-Case)}$$

$$\frac{N \succeq_p^{(n)} N'}{\texttt{fix}\ x.N \succeq_p^{(n)} \texttt{fix}\ x.N'} \quad \text{(SP-Fix)}$$

$$\frac{N_1 \succeq_p^{(m)} N_1' \qquad N_2 \succeq_p^{(n)} N_2'}{(\lambda x.N_1)\ N_2 \succeq_p^{(m+n\times Oc(x,N_1')+1)} [N_2'/x]N_1'} \quad \text{(SP-Beta1)}$$

$$\frac{N \succeq_p^{(n)} N'}{\texttt{fix}\ x.N \succeq_p^{(n\times(Oc(x,N')+1)+1)} [\texttt{fix}\ x.N'/x]N'} \quad \text{(SP-Beta2)}$$

$$\frac{N_1 \succeq_p^{(n)} N_1'}{\texttt{if0}\ 0\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \succeq_p^{(n+1)} N_1'} \quad \text{(SP-Beta3)}$$

$$\frac{N_2 \succeq_p^{(n)} N_2' \qquad m \neq 0}{\texttt{if0}\ m\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \succeq_p^{(n+1)} N_2'} \quad \text{(SP-Beta4)}$$

$$\frac{N_i \succeq_p^{(n_i)} N_i' \qquad L \vdash pat_i \Rightarrow [\cdot]}{\texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \succeq_p^{(n_i+1)} N_i'} \quad \text{(SP-Beta5)}$$

$$\frac{L_1 \succeq_p^{(m)} L_1' \qquad N_i \succeq_p^{(n_i)} N_i' \qquad L \vdash pat_i \Rightarrow [L_1/x]}{\begin{array}{l}\texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \\ \succeq_p^{(n_i+m\times Oc(x,N_i')+1)} [L_1'/x]N_i'\end{array}} \quad \text{(SP-Beta6)}$$

If the weight n on $\succeq_p^{(n)}$ does not matter, we omit n and write $\succeq_p$ instead.

Lemma A.12. *If $P \succeq_p^{(m)} P'$, then*

$$[P/x]N \succeq_p^{(m\times Oc(x,N))} [P'/x]N$$

Proof. The proof follows by induction on structure of N.

- Case $N = x$: In this case, we get $[P/x]N = P$ and $Oc(x,N) = 1$. By assumption, $[P/x]N \succeq_p^{(m\times Oc(x,N))} [P'/x]N$ holds.
- Case $N = y$: In this case, we get $[P/x]N = y$ and $Oc(x,N) = 0$. By SP-Id, $[P/x]N \succeq_p^{(0)} [P'/x]N$ holds.
- Case $N = n$: In this case, we get $[P/x]N = n$ and $Oc(x,N) = 0$. By SP-Id, $[P/x]N \succeq_p^{(0)} [P'/x]N$ holds.
- Case $N = [\,]$: In this case, we get $[P/x]N = [\,]$ and $Oc(x,N) = 0$. By SP-Id, $[P/x]N \succeq_p^{(0)} [P'/x]N$ holds.
- Case $N = n :: L$: In this case, we get $[P/x]N = n :: [P/x]L$ and $Oc(x,N) = Oc(x,L)$. By the induction hypothesis, $[P/x]L \succeq_p^{(m\times Oc(x,N_0))} [P'/x]L$. By SP-Cons, $[P/x]N \succeq_p^{(m\times Oc(x,N))} [P'/x]N$ holds.
- Case $N = \lambda y.N_0$: In this case, we get $[P/x]N = \lambda y.[P/x]N_0$ and $Oc(x,N) = Oc(x,N_0)$. By the induction hypothesis, $[P/x]N_0 \succeq_p^{(m\times Oc(x,N_0))} [P'/x]N_0$ By SP-Abs, $[P/x]N \succeq_p^{(m\times Oc(x,N))} [P'/x]N$ holds.
- Case $N = \Lambda y.N_0$: The required condition holds as $N = \lambda y.N_0$.
- Case $N = N_0\,N_1$: In this case, we get $Oc(x,N) = Oc(x,N_0) + Oc(x,N_1)$. By the induction hypothesis,
$$[P/x]N_0 \succeq_p^{(m\times Oc(x,N_0))} [P'/x]N_0$$
$$[P/x]N_1 \succeq_p^{(m\times Oc(x,N_1))} [P'/x]N_1$$
holds. By SP-App,
$$([P/x]N_0)\,([P/x]N_1) \succeq_p^{(m\times Oc(x,N))} ([P'/x]N_0)\,([P'/x]N_1)$$
holds.
- Case $N = \texttt{if0}\ N_0\ \texttt{then}\ N_1\ \texttt{else}\ N_2$: In this case, we get $Oc(x,N) = Oc(x,N_0) + Oc(x,N_1) + Oc(x,N_2)$. By the induction hypothesis,
$$[P/x]N_0 \succeq_p^{(m\times Oc(x,N_0))} [P'/x]N_0$$
$$[P/x]N_1 \succeq_p^{(m\times Oc(x,N_1))} [P'/x]N_1$$
$$[P/x]N_2 \succeq_p^{(m\times Oc(x,N_2))} [P'/x]N_2$$
holds. By SP-If0,
$$\texttt{if0}\ [P/x]N_0\ \texttt{then}\ [P/x]N_1\ \texttt{else}\ [P/x]N_2 \succeq_p^{(m\times Oc(x,N))} \texttt{if0}\ [P'/x]N_0\ \texttt{then}\ [P'/x]N_1\ \texttt{else}\ [P'/x]N_2$$
holds.
- Case $N = \texttt{case}\ N_0\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)$: In this case, we get $Oc(x,N) = \sum_{i=0}^{k} Oc(x,N_i)$. By the induction hypothesis,
$$[P/x]N_i \succeq_p^{(m\times Oc(x,N_i))} [P'/x]N_i$$

for $i \in \{0, \ldots, k\}$ holds. By SP-Case,

$$\texttt{case}\ [P/x]N_0\ \texttt{of}\ (pat_1 \Rightarrow [P/x]N_1 \mid \cdots \mid pat_k \Rightarrow [P/x]N_k) \geq_p^{(m \times Oc(x,N))} \texttt{case}\ [P'/x]N_0\ \texttt{of}\ (pat_1 \Rightarrow [P'/x]N_1 \mid \cdots \mid pat_k \Rightarrow [P'/x]N_k)$$

- Case $N = \texttt{fix}\ y.N'$: In this case, we get $[P/x]N = \texttt{fix}\ y.[P/x]N'$ and $Oc(x, N) = Oc(x, N')$. By the induction hypothesis, $[P/x]N' \geq_p^{(m \times Oc(x,N'))} [P'/x]N'$ By SP-Fix, $[P/x]N \geq_p^{(m \times Oc(x,N))} [P'/x]N$ holds.

□

Lemma A.13. *If $N \longrightarrow_e N'$ and $P \geq_p^{(m)} P'$ then*

$$[P/x]N \geq_p^{(m \times Oc(x,N')+1)} [P'/x]N'$$

Proof. The proof follows by induction on the structure of the evaluation context $\mathcal{E}_e$ of $N \longrightarrow_e N'$. If $\mathcal{E}_e = \langle\,\rangle$, the proof proceeds by case analysis of the reduction.

- Case $N = (\lambda y.N_1)\ N_2, N' = [N_2/y]N_1$: By α-conversion, we can assume the variable y does not appear in P. By Lemma A.12, we get

$$[P/x]N_1 \geq_p^{(m \times Oc(x,N_1))} [P'/x]N_1$$
$$[P/x]N_2 \geq_p^{(m \times Oc(x,N_2))} [P'/x]N_2$$

  Since $Oc(x, N') = Oc(x, N_1) + Oc(y, N_1) \times Oc(x, N_2)$, we get

$$[P/x]N = (\lambda y.[P/x]N_1)\ [P/x]N_2 \geq_p^{(m \times Oc(x,N')+1)} [P'/x][N_2/y]N_1$$

  by SP-Beta1.
- Case $N = \texttt{fix}\ y.N_1, N' = [\texttt{fix}\ y.N_1/y]N_1$: By Lemma A.12, we get

$$[P/x]N_1 \geq_p^{(m \times Oc(x,N_1))} [P'/x]N_1$$

  Since $Oc(x, N') = Oc(x, N_1) \times (Oc(y, N_1) + 1)$,

$$m \times Oc(x, N_1) \times (Oc(y, N_1) + 1) + 1 = m \times Oc(x, N') + 1$$

  holds. Thus, we get

$$[P/x]N = \texttt{fix}\ y.[P/x]N_1 \geq_p^{(m \times Oc(x,N')+1)} [P'/x][\texttt{fix}\ y.N_1/y]N_1$$

  by SP-Beta2.
- Case $N = \texttt{if0}\ 0\ \texttt{then}\ N_1\ \texttt{else}\ N_2, N' = N_1$: By Lemma A.12, we get

$$[P/x]N_1 \geq_p^{(m \times Oc(x,N_1))} [P'/x]N_1$$

  By applying SP-Beta3, we get

$$[P/x]N = \texttt{if0}\ 0\ \texttt{then}\ [P/x]N_1\ \texttt{else}\ [P/x]N_2 \geq_p^{(m \times Oc(x,N_1)+1)} [P'/x]N_1.$$

- Case $N = \texttt{if0}\ l\ \texttt{then}\ N_1\ \texttt{else}\ N_2, N' = N_2$ where $l \neq 0$: By Lemma A.12, we get

$$[P/x]N_2 \geq_p^{(m \times Oc(x,N_2))} [P'/x]N_2$$

  By applying SP-Beta4, we get

$$[P/x]N = \texttt{if0}\ l\ \texttt{then}\ [P/x]N_1\ \texttt{else}\ [P/x]N_2 \geq_p^{(m \times Oc(x,N_2)+1)} [P'/x]N_2.$$

- Case $N = \texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k), L \vdash pat_i \Rightarrow [\cdot], N' = N_i$: By Lemma A.12, we get

$$[P/x]N_i \geq_p^{(m \times Oc(x,N_i))} [P'/x]N_i$$

  By applying SP-Beta5, we get

$$[P/x]N = \texttt{case}\ [P/x]L\ \texttt{of}\ (pat_1 \Rightarrow [P/x]N_1 \mid \cdots \mid pat_k \Rightarrow [P/x]N_k) \geq_p^{(m \times Oc(x,N_i)+1)} [P'/x]N_i.$$

- Case $N = \texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k), L \vdash pat_i \Rightarrow [L_1/y], N' = [L_1/y]N_i$: By Lemma A.12, we get

$$[P/x]L_1 \geq_p^{(m \times Oc(x,L_i))} [P'/x]L_1$$
$$[P/x]N_i \geq_p^{(m \times Oc(x,N_i))} [P'/x]N_i$$

  Since $Oc(x, N') = Oc(x, N_i) + Oc(y, N_i) \times Oc(x, L_1)$, we get

$$[P/x]N = \texttt{case}\ [P/x]L\ \texttt{of}\ (pat_1 \Rightarrow [P/x]N_1 \mid \cdots \mid pat_k \Rightarrow [P/x]N_k) \geq_p^{(m \times Oc(x,N')+1)} [P'/x][L_1/y]N_i$$

  by SP-Beta6.

Otherwise the proof proceeds by case analysis on $\mathcal{E}_e$.

- Case $\mathcal{E}_e = \mathcal{E}'_e\ N_0$: In this case, we can suppose that

$$N = \mathcal{E}'_e[Q]N_0 \longrightarrow_e \mathcal{E}'_e[Q']N_0 = N'$$

  where $Q \longrightarrow_e Q'$. By induction hypothesis, we get $[P/x]\mathcal{E}'_e[Q] \geq_p^{(m \times Oc(x,\mathcal{E}'_e[Q'])+1)} [P'/x]\mathcal{E}'_e[Q']$

  By Lemma A.12, we get

$$[P/x]N_0 \geq_p^{(m \times Oc(x,N_0))} [P'/x]N_0$$

  Since $Oc(x, N') = Oc(x, \mathcal{E}'_e[Q']) + Oc(x, N_0)$, we get

$$[P/x]N \geq_p^{(m \times Oc(x,N')+1)} [P'/x]N'$$

  by SP-App.
- Case $\mathcal{E}_e = \texttt{if0}\ \mathcal{E}'_e\ \texttt{then}\ N_1\ \texttt{else}\ N_2$: In this case, we can suppose that

$$N = \texttt{if0}\ \mathcal{E}'_e[Q]\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \longrightarrow_e \texttt{if0}\ \mathcal{E}'_e[Q']\ \texttt{then}\ N_1\ \texttt{else}\ N_2 = N'$$

  where $Q \longrightarrow_e Q'$. By the induction hypothesis, we get

$$[P/x]\mathcal{E}'_e[Q] \geq_p^{(m \times Oc(x,\mathcal{E}'_e[Q'])+1)} [P'/x]\mathcal{E}'_e[Q'].$$

By Lemma A.12, we get

$$[P/x]N_1 \succeq_p^{(m\times Oc(x,N_1))} [P'/x]N_1$$
$$[P/x]N_2 \succeq_p^{(m\times Oc(x,N_2))} [P'/x]N_2$$

Since $Oc(x,N') = Oc(x,\mathcal{E}'_e[Q']) + Oc(x,N_1) + Oc(x,N_2)$, we get

$$[P/x]N \succeq_p^{(m\times Oc(x,N')+1)} [P'/x]N'$$

by SP-If0.

- Case $\mathcal{E}_e = \texttt{case}\ \mathcal{E}'_e\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)$: In this case, we can suppose that

$$N = \texttt{case}\ \mathcal{E}'_e[Q]\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \longrightarrow_e \texttt{case}\ \mathcal{E}'_e[Q']\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) = N'$$

where $Q \longrightarrow_e Q'$. By the induction hypothesis, we get

$$[P/x]\mathcal{E}'_e[Q] \succeq_p^{(m\times Oc(x,\mathcal{E}'_e[Q'])+1)} [P'/x]\mathcal{E}'_e[Q']$$

By Lemma A.12, we get

$$[P/x]N_i \succeq_p^{(m\times Oc(x,N_i))} [P'/x]N_i \quad (i \in \{1,\ldots,k\})$$

Since $Oc(x,N') = Oc(x,\mathcal{E}'_e[Q']) + \sum_{i=1}^{k} Oc(x,N_i)$, we get

$$[P/x]N \succeq_p^{(m\times Oc(x,N')+1)} [P'/x]N'$$

by SP-Case.

- Case $\mathcal{E}_e = \Lambda y.\mathcal{E}_e$ In this case, we can suppose that

$$N = \Lambda y.\mathcal{E}'_e[Q] \longrightarrow_e \Lambda y.\mathcal{E}'_e[Q'] = N'$$

where $Q \longrightarrow_e Q'$. By induction hypothesis, we get

$$[P/x]\mathcal{E}'_e[Q] \succeq_p^{(m\times Oc(x,\mathcal{E}'_e[Q'])+1)} [P'/x]\mathcal{E}'_e[Q'].$$

Since $Oc(x,N') = Oc(x,\mathcal{E}'_e[Q'])$, we get

$$[P/x]N \succeq_p^{(m\times Oc(x,N')+1)} [P'/x]N'$$

by SP-Abs2.

□

Lemma A.14. *If $N \succeq_p^{(n)} N'$ and $P \succeq_p^{(m)} P'$, then*

$$[P/x]N \succeq_p^{(n+m\times Oc(x,N'))} [P'/x]N'$$

holds.

Proof. The proof follows by induction on the derivation of $N \succeq_p^{(n)} N'$.

- Case SP-Id This follows by Lemma A.12.
- Case SP-Beta This follows by Lemma A.13.
- Case SP-Eta In this case, we can suppose that

$$N = \Lambda y.N_0\, y \succeq_p^{(k+1)} N'_0 = N'$$

where $N_0 \succeq_p^{(k)} N'_0$. By induction hypothesis, we get

$$[P/x]N_0 \succeq_p^{(k+m\times Oc(x,N'_0))} [P'/x]N'_0.$$

Since $n = k+1$ holds, we get

$$[P/x]\Lambda y.N_0\ y \succeq_p^{(n+m\times Oc(x,N'_0))} [P'/x]N'_0.$$

by applying SP-Eta.

- Case SP-Cons In this case, we can suppose that

$$N = l :: L \succeq_p^{(n)} l :: L' = N'$$

where $L \succeq_p^{(n)} L'$. By induction hypothesis, we get

$$[P/x]L \succeq_p^{(n+m\times Oc(x,N'_0))} [P'/x]L'$$

Then, we get

$$[P/x]N = l :: [P/x]L \succeq_p^{(n+m\times Oc(x,N'))} l :: L' = [P'/x]N'$$

- Case SP-Abs In this case, we can suppose that

$$N = \lambda y.N_0 \succeq_p^{(n)} \lambda y.N'_0 = N'$$

where $N_0 \succeq_p^{(n)} N'_0$. By induction hypothesis, we get

$$[P/x]N_0 \succeq_p^{(n+m\times Oc(x,N'_0))} [P'/x]N'_0$$

Then, we get

$$[P/x]N = \lambda y.[P/x]N_0 \succeq_p^{(n+m\times Oc(x,N'))} \lambda y.[P'/x]N'_0 = [P'/x]N'$$

by applying SP-Abs.

- Case SP-Abs2 This similarly holds as the case SP-Abs.
- Case SP-App In this case, we can suppose that

$$N = N_0\ N_1 \succeq_p^{(n_0+n_1)} N'_0\ N_1 = N'$$

where $N_0 \succeq_p^{(n_0)} N'_0, N_1 \succeq_p^{(n_1)} N'_1$. By induction hypothesis, we get

$$[P/x]N_0 \succeq_p^{(n_0+m\times Oc(x,N'_0))} [P'/x]N'_0$$
$$[P/x]N_1 \succeq_p^{(n_1+m\times Oc(x,N'_1))} [P'/x]N'_1$$

Since $n = n_0+n_1$, $Oc(x,N') = Oc(x,N'_0) + Oc(x,N'_1)$, we get

$$[P/x]N = ([P/x]N_0)\,([P/x]N_1) \succeq_p^{(n+m\times Oc(x,N'))} ([P'/x]N'_0)\,([P'/x]N'_1) = [P'/x]N'$$

by applying SP-App.

- Case SP-If0 In this case, we can suppose that

$$N = \texttt{if0}\ N_0\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \succeq_p^{(\sum_0^2 n_i)} \texttt{if0}\ N'_0\ \texttt{then}\ N'_1\ \texttt{else}\ N'_2$$

where $N_i \succeq_p^{(n_i)} N'_i (i \in \{0,1,2\})$. By induction hypothesis, we get

$$[P/x]N_i \succeq_p^{(n_i+m\times Oc(x,N'_i))} [P'/x]N'_i$$

for $i \in \{0,1,2\}$. Since $n = \sum_0^2 n_i, Oc(x,N') = \sum_0^2 Oc(x,N_i')$, we get

$$\begin{aligned}
&[P/x]N \\
&= \texttt{if0}\ [P/x]N_0\ \texttt{then}\ [P/x]N_1\ \texttt{else}\ [P/x]N_2 \\
&\geq_p^{(n+m\times Oc(x,N'))}\ \texttt{if0}\ [P'/x]N_0'\ \texttt{then}\ [P'/x]N_1'\ \texttt{else}\ [P'/x]N_2' \\
&= [P'/x]N'
\end{aligned}$$

by SP-If0.

- Case SP-Case In this case, we can suppose that

$$\begin{aligned}
N &= \texttt{case}\ N_0\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \\
&\geq_p^{(\sum_0^k n_i)}\ \texttt{case}\ N_0'\ \texttt{of}\ (pat_1 \Rightarrow N_1' \mid \cdots \mid pat_k \Rightarrow N_k')
\end{aligned}$$

where $N_i \geq_p^{(n_i)} N_i' (i \in \{0,\ldots,k\})$. By induction hypothesis, we get

$$[P/x]N_i \geq_p^{(n_i+m\times Oc(x,N_i'))}\ [P'/x]N_i'$$

for $i \in \{0,\ldots,k\}$. Since

$$n = \sum_0^k n_i, Oc(x,N') = \sum_0^k Oc(x,N_i'),$$

we get

$$\begin{aligned}
&[P/x]N \\
&= \texttt{case}\ [P/x]N_0\ \texttt{of} \\
&\quad (pat_1 \Rightarrow [P/x]N_1 \mid \cdots \mid pat_k \Rightarrow [P/x]N_k) \\
&\geq_p^{(n+m\times Oc(x,N'))}\ \texttt{case}\ [P'/x]N_0'\ \texttt{of} \\
&\quad (pat_1 \Rightarrow [P'/x]N_1' \mid \cdots \mid pat_k \Rightarrow [P'/x]N_k') \\
&= [P'/x]N'
\end{aligned}$$

by SP-Case.

- Case SP-Fix In this case, we can suppose that

$$N = \texttt{fix}\ y.N_0 \geq_p^{(n)}\ \texttt{fix}\ y.N_0' = N'$$

where $N_0 \geq_p^{(n)} N_0'$. By induction hypothesis, we get

$$[P/x]N_0 \geq_p^{(n+m\times Oc(x,N_0'))}\ [P'/x]N_0'$$

By α-conversion, we can assume the variable y does not appear in P' and

$$Oc(y,[P'/x]N_0') = Oc(y,N_0')$$

holds. Then, we get

$$\begin{aligned}
&[P/x]N = \texttt{fix}\ y.[P/x]N_0 \geq_p^{(n+m\times Oc(x,N'))} \\
&\quad \texttt{fix}\ y.[P'/x]N_0' = [P'/x]N'
\end{aligned}$$

by applying SP-Fix.

- Case SP-Beta1 In this case, we can suppose

$$N = (\lambda y.N_1)\ N_2 \geq_p^{(n_1+n_2\times Oc(y,N_1')+1)}\ [N_2'/x]N_1' = N'$$

where $N_i \geq_p^{(n_i)} N_i'$ for $(i \in \{1,2\})$. By induction hypothesis, we get

$$\begin{aligned}
[P/x]N_1 &\geq_p^{(n_1')}\ [P'/x]N_1' \\
[P/x]N_2 &\geq_p^{(n_2')}\ [P'/x]N_2'
\end{aligned}$$

where $n_1' = n_1 + m\times Oc(x,N_1'), n_2' = n_2 + m\times Oc(x,N_2')$. By α-conversion, we can assume the variable y does not appear in P' and

$$\begin{aligned}
&Oc(y,[P'/x]N_1') \\
&= Oc(y,N_1') + Oc(x,N_i')\times Oc(y,P') \\
&= Oc(y,N_1')
\end{aligned}$$

holds. By applying SP-Beta1, we get

$$\begin{aligned}
&[P/x]N \\
&= (\lambda y.[P/x]N_1)\ [P/x]N_2 \\
&\geq_p^{(n_1'+n_2'\times Oc(y,N_1')+1)}\ [P'/x][N_2'/y]N_1'.
\end{aligned}$$

Since

$$\begin{aligned}
n &= n_1 + Oc(y,N_1')\times n_2 + 1 \\
Oc(x,N') &= Oc(x,N_1') + Oc(x,N_2')\times Oc(y,N_1')
\end{aligned}$$

holds, we get

$$\begin{aligned}
&n_1' + n_2'\times Oc(y,N_1') + 1 \\
&= n + m\times Oc(x,N_1') + m\times Oc(x,N_2')\times Oc(y,N_1') \\
&= n + m\times Oc(x,N').
\end{aligned}$$

Thus, the required condition holds.

- Case SP-Beta2 In this case, we can assume

$$N = \texttt{fix}\ y.N_1 \geq_p^{(n_1\times(Oc(y,N_1')+1)+1)}\ [N_2'/y]N_1' = N'$$

where $N_i \geq_p^{(n_i)} N_i'$ for $(i \in \{1,2\})$. By induction hypothesis, we get

$$[P/x]N_1 \geq_p^{(n_1')}\ [P'/x]N_1'$$

where $n_1' = n_1 + m\times Oc(x,N_1')$. By α-conversion, we can assume the variable y does not appear in P' and

$$\begin{aligned}
&Oc(y,[P'/x]N_1') \\
&= Oc(y,N_1') + Oc(x,N_i')\times Oc(y,P') \\
&= Oc(y,N_1')
\end{aligned}$$

holds. By applying SP-Beta2, we get

$$\begin{aligned}
&[P/x]N = \texttt{fix}\ y.[P/x]N_1 \geq_p^{(n_1'\times(Oc(y,N_1')+1)+1)} \\
&\quad [P'/x][\texttt{fix}\ y.N_1'/y]N_1'.
\end{aligned}$$

Since

$$\begin{aligned}
n &= n_1\times(Oc(y,N_1')+1)+1 \\
Oc(x,N') &= Oc(x,N_1')\times(Oc(y,N_1')+1)
\end{aligned}$$

holds, we get

$$\begin{aligned} & n_1' \times (Oc(y, N_1') + 1) + 1 \\ = \ & n + m \times Oc(x, N_1') \times (Oc(y, N_1') + 1) \\ = \ & n + m \times Oc(x, N'). \end{aligned}$$

Thus, the required condition holds.

- Case SP-Beta3 In this case, we can assume

$$\begin{aligned} N = \ & \texttt{if0}\ 0\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \\ & \geq_p^{(n_1+1)}\ N_1' = N' \end{aligned}$$

where $N_1 \geq_p^{(n_1)} N_1'$. By induction hypothesis, we get

$$[P/x]N_1 \geq_p^{(n_1')}\ [P'/x]N_1'$$

where $n_1' = n_1 + m \times Oc(x, N_1')$. By applying SP-Beta3, we get

$$[P/x]N = \texttt{if0}\ 0\ \texttt{then}\ [P/x]N_1\ \texttt{else}\ [P/x]N_2 \geq_p^{(n_1'+1)}\ [P'/x]N_1'.$$

Since

$$\begin{aligned} n &= n_1 + 1 \\ Oc(x, N') &= Oc(x, N_1') \end{aligned}$$

$n = n_1 + 1$holds, we get

$$\begin{aligned} n_1' + 1 &= n + m \times Oc(x, N_1') \\ &= n + m \times Oc(x, N'). \end{aligned}$$

Thus, the required condition holds.

- Case SP-Beta4 This similarly holds as the case SP-Beta3.
- Case SP-Beta5 In this case, we can assume

$$\begin{aligned} N = \ & \texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \\ & \geq_p^{(n_i+1)}\ N_i' = N' \end{aligned}$$

where $L \vdash pat_i \Rightarrow [\cdot], N_i \geq_p^{(n_i)} N_i'$. By induction hypothesis, we get

$$[P/x]N_i \geq_p^{(n_i')}\ [P'/x]N_i'$$

where $n_i' = n_i + m \times Oc(x, N_i')$. By applying SP-Beta5, we get

$$\begin{aligned} & [P/x]N \\ = \ & \texttt{case}\ [P/x]L\ \texttt{of}\ (pat_1 \Rightarrow [P/x]N_1 \mid \cdots \mid pat_k \Rightarrow [P/x]N_k) \\ \geq_p^{(n_i'+1)} \ & [P'/x]N_i'. \end{aligned}$$

Since

$$\begin{aligned} n &= n_i + 1 \\ Oc(x, N') &= Oc(x, N_i') \end{aligned}$$

holds, we get

$$\begin{aligned} n_i' + 1 &= n + m \times Oc(x, N_i') \\ &= n + m \times Oc(x, N'). \end{aligned}$$

Thus, the required condition holds.

- Case SP-Beta6 In this case, we can suppose

$$\begin{aligned} N = \ & \texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k) \\ & \geq_p^{(n_i+l\times Oc(y,N_i)+1)}\ [L_1'/y]N_i' = N' \end{aligned}$$

where $L \vdash pat_i \Rightarrow [L_1/y], N_i \geq_p^{(n_i)} N_i'.L_1 \geq_p^{(l)} L_1'$. By induction hypothesis, we get

$$\begin{aligned} [P/x]N_i &\geq_p^{(n_i')}\ [P'/x]N_i' \\ [P/x]L_1 &\geq_p^{(l')}\ [P'/x]L_1' \end{aligned}$$

where $n_i' = n_i + m \times Oc(x, N_i'), l' = l + m \times Oc(x, L')$. By applying SP-Beta6, we get

$$\begin{aligned} & [P/x]N \\ = \ & \texttt{case}\ [P/x]L\ \texttt{of}\ (pat_1 \Rightarrow [P/x]N_1 \mid \cdots \mid pat_k \Rightarrow [P/x]N_k) \\ \geq_p^{(n_i'+l'\times Oc(y,[P'/x]N_i')+1)} \ & [[P'/x]L_1'/y][P'/x]N_i' \\ = \ & [P'/x][L'/y]N_i'. \end{aligned}$$

By α-conversion, we can assume the variable y does not appear in P and

$$\begin{aligned} & Oc(y, [P'/x]N_i') \\ = \ & Oc(y, N_i') + Oc(x, N_i') \times Oc(y, P') \\ = \ & Oc(y, N_i') \end{aligned}$$

holds. Since

$$\begin{aligned} n &= n_i + l \times Oc(y, N_i) + 1 \\ Oc(x, [L'/y]N_i') &= Oc(x, N_i') + Oc(y, N_i) \times Oc(x, L') \end{aligned}$$

holds, we get

$$\begin{aligned} & n_i' + l' \times Oc(y, [P'/x]N_i') + 1 \\ = \ & n + m \times (Oc(x, N_i') + Oc(y, N_i) \times Oc(x, L')) \\ = \ & n + m \times Oc(x, [L'/y]N_i') \\ = \ & n + m \times Oc(x, N'). \end{aligned}$$

Thus, the required condition holds.

□

Lemma A.15. *Let V be a value or a variable of a base type for some type environment. If $N \geq_p^{(n)} V$, then the following statements hold.*

1. *$N \longrightarrow_e^* n :: L$ holds for some L such that $L \geq_p L'$ (if $V = n :: L'$)*
2. *$N \longrightarrow_e^* V$ (otherwise)*

Proof. The proof proceeds by induction on the derivation of $N \geq_p^{(n)} V$.

- Case SP-Id: trivial.
- Case SP-Beta: trivial.
- Case SP-Cons: trivial.

- Case SP-Beta1: In this case, we can suppose that

$$N = (\lambda x.N_1)\ N_2$$
$$V = [N_2'/x]N_1'$$

where $N_1 \succeq_p N_1', N_2 \succeq_p N_2'$. Since V is a value or a variable, one of the following holds.
  - Case $V = N_1', x \notin FV(N_1')$ In this case, we get $N = (\lambda x.W)N_2$ where $W \succeq_p V$. By induction hypothesis, $W \longrightarrow_e^* V$ or $W \longrightarrow_e^* n :: L'$ where $n :: L \succeq_p n :: L' = V$ holds. Since $N \longrightarrow_e W$, the required condition holds.
  - Case $V = N_2', N_1' = x$ In this case, we get $N = (\lambda x.W_1)W_2$ where $W_1 \succeq_p x, W_2 \succeq_p V$. By induction hypothesis, $W_1 \longrightarrow_e^* x$ holds. Also, we get $W_2 \longrightarrow_e^* V$ or $W_2 \longrightarrow_e^* n :: L'$ where $n :: L \succeq_p n :: L' = V$. By Lemma A.1, we get $[W_2/x]W_1 \longrightarrow_e^* [W_2/x]x = W_2$. Since $N \longrightarrow_e [W_2/x]W_1$, the required condition holds.
  - Case $V = n :: [N_2'/x]L', N_1' = n :: L'$ In this case, we get $N = (\lambda x.W_1)W_2$ where $W_1 \succeq_p n :: L'$. By induction hypothesis, $W_1 \longrightarrow_e^* n :: L\ (L \succeq_p L')$ holds. By Lemma A.1, we get $[W_2/x]W_1 \longrightarrow_e^* n :: [W_2/x]L$. By Lemma A.14, $[W_2/x]L \succeq_p [N_2'/x]L'$ holds. By SP-Cons $n :: [W_2/x]L \succeq_p n :: [N_2'/x]L'$ holds. Since $N \longrightarrow_e n :: [W_2/x]W_1 \longrightarrow_e^* n :: [W_2/x]L$, the required condition holds.
- Case SP-Beta2: In this case, we can suppose that

$$N = \texttt{fix}\ x.N_1$$
$$V = [\texttt{fix}\ x.N_1'/x]N_1'$$

where $N_1 \succeq_p N_1'$. Since V is value or variable, $V = N_1', x \notin FV(N_1')$ or $V = n :: [\texttt{fix}\ x.N_1'/x]L', N_1' = n :: L'$ holds. In both case, the required condition holds as well as Case SP-Beta1.
- Case SP-Beta3: In this case, we can suppose that

$$N = \texttt{if0}\ 0\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$
$$V = N_1'$$

where $N_1 \succeq_p N_1'$. Since $N \longrightarrow_e N_1$, the required condition follows by the induction hypothesis.
- Case SP-Beta4: In this case, we can suppose that

$$N = \texttt{if0}\ m\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$
$$V = N_2'$$

where $m \neq 0, N_2 \succeq_p N_2'$. Since $N \longrightarrow_e N_2$, the required condition follows by the induction hypothesis.
- Case SP-Beta5: In this case, we can suppose that

$$N = \texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)$$
$$V = N_i'$$

where $L \vdash N_i \Rightarrow [\cdot], N_i \succeq_p N_i'$. Since $N \longrightarrow_e N_i$, the required condition follows by the induction hypothesis.
- Case SP-Beta6: In this case, we can suppose that

$$N = \texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)$$
$$V = [L_1'/x]N_i'$$

where $L \vdash N_i \Rightarrow [L_1/x], N_i \succeq_p N_i', L_1 \succeq_p L_1'$. Since V is value or variable, $N_i' = V, x \notin FV(V)$ or $N_i' = x, L_1' = V$ or $V = n :: [L_1'/x]L', N_i' = n :: L'$ holds. In all cases, the required condition follows as well as Case SP-Beta1

Since V is a value or a variable, $N \succeq_p V$ cannot hold in the other rules. □

Lemma A.16. *Let V be value or variable of function type for some type environment. If $N \succeq_p V$, then the following statements hold for any term C.*

1. *$N\ C \longrightarrow_e^* (\lambda x.P')\ C$ (if $V = \lambda x.P$) for some term P' such that $P' \succeq_p P$.*
2. *$N\ C \longrightarrow_e^* x\ C$ (if $V = x$)*

Proof. The proof proceeds by induction on the derivation of $N \succeq_p V$.

- Case SP-Id: trivial.
- Case SP-Beta: trivial.
- Case SP-Eta: In this case, we get $N = (\Lambda x.N_0\ x)$ where $N_0 \succeq_p V$. Since $N\,C \longrightarrow_e N_0\,C$, the required condition follows by the induction hypothesis.
- Case SP-Beta1: In this case, N and V is of the form

$$N = (\lambda x.N_1)\ N_2$$
$$V = [N_2'/x]N_1'$$

where $N_1 \succeq_p N_1', N_2 \succeq_p N_2'$. One of the following holds:
  - Case $V = N_1', x \notin FV(N_1')$: In this case, we get

$$N\,C = (\lambda y.W_1)\ W_2\,C \longrightarrow_e W_1\,C$$

where $W_1 \succeq_p V, W_2 \succeq_p N_2'$. Thus, the required condition follows by induction hypothesis of $W_1 \succeq_p V$.
  - Case $V = N_2', N_1' = x$: In this case, we get

$$N\,C = (\lambda y.W_1)\ W_2\,C \longrightarrow_e [W_2/y]W_1\,C.$$

where $W_1 \succeq_p y, W_2 \succeq_p V$. By induction hypothesis, we get

$$W_1\,C \longrightarrow_e^* y\,C.$$

By Lemma A.1, we get

$$[W_2/y]W_1\,C \longrightarrow_e^* [W_2/y]y = W_2\,C.$$

Thus, the required condition holds by induction hypothesis of $W_2 \succeq_p V$.
  - Case $V = \lambda y.[N_2'/x]Q', N_1' = \lambda y.Q'$: In this case, we get

$$N\,C = (\lambda x.N_1)\ N2\,C$$
$$\longrightarrow_e ([N_2/x]N_1)\,C.$$

By the induction hypothesis, we get

$$N_1\, C \longrightarrow_e^* (\lambda y.Q)\, C$$

for some Q such that $Q \succeq_p Q'$. By Lemma A.1,

$$([N_2/x]N_1)\, C \longrightarrow_e^* (\lambda y.[N_2/x]Q)\, C$$

holds. By Lemma A.14, we get $[N_2/x]Q \succeq_p [N_2'/x]Q'$. Thus, the required condition holds.

- Case SP-Beta2 In this case, N and V is of the form

$$N = \texttt{fix}\ y.N_1$$
$$V = [\texttt{fix}\ y.N_1'/y]N_1'$$

where $N_1 \succeq_p N_1'$. Since V is value or variable, $V = N_1', x \notin FV(N_1')$ or $V = \lambda x.[\texttt{fix}\ y.N_1'/y]Q', N_1' = \lambda x.Q'$ holds. In both case, the required condition holds as well as Case SP-Beta1.

- Case SP-Beta3 In this case, N and V is of the form

$$N = \texttt{if0}\ 0\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$
$$V = N_1'$$

where $N_1 \succeq_p N_1'$. Since $N\,C \longrightarrow_e N_1\,C$, the required condition holds by induction hypothesis of $N_1 \succeq_p N_1'$.

- Case SP-Beta4 In this case, N and V is of the form

$$N = \texttt{if0}\ m\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$
$$V = N_2'$$

where $N_2 \succeq_p N_2'$. Since $N\,C \longrightarrow_e N_2\,C$, the required condition holds by induction hypothesis of $N_2 \succeq_p N_2'$.

- Case SP-Beta5 In this case, N and V is of the form

$$N = \texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)$$
$$V = N_i'$$

where $V' \vdash N_i \Rightarrow [\cdot], N_i \succeq_p N_i'$. Since $N\,C \longrightarrow_e N_i\,C$, the required condition holds by induction hypothesis of $N_i \succeq_p N_i'$.

- Case SP-Beta6 In this case, N and V is of the form

$$N = \texttt{case}\ L\ \texttt{of}\ (pat_1 \Rightarrow N_1 \mid \cdots \mid pat_k \Rightarrow N_k)$$
$$V = [L_1'/x]N_i'$$

where $L \vdash N_i \Rightarrow [L_1/x], N_i \succeq_p N_i', L_1 \succeq_p L_1'$. Since V is value or variable, $N_i' = V, x \notin FV(V)$ or $N_i' = x, L_1' = V$ or $V = \lambda y.[L_1'/x]Q', N_1' = \lambda y.Q'$ holds. In all cases, the required condition follows as well as Case SP-Beta1.

□

Lemma A.17. *Suppose* $\vdash N : \sigma$ *and* $N \succeq_p^{(n)} N' \longrightarrow_e N_1'$, *then* $N \longrightarrow_e^* N_1 \succeq_p^{(m)} N_1'$ *for some* N_1 *and* m.

Proof. The proof proceeds by induction on lexicographic order on n and the depth of derivation of $N \succeq_p^{(n)} N'$, with case analysis on the last rule used for deriving $N \succeq_p^{(n)} N'$.

- Case SP-Id: In this case, we have $N = N'$. Thus, the required condition holds for $N_1 = N_1'$ and $m = 0$.
- Case SP-Beta: In this case, we have $N \longrightarrow_e N_1'$. Thus, the required condition holds for $N_1 = N_1'$ and $m = 0$.
- Case SP-Eta: In this case, we have $N = \Lambda u.N_0\ u$ and $N_0 \succeq_p^{(n-1)} N'$. Since $N' \longrightarrow_e N_1'$, By applying the induction hypothesis to

$$N_0 \succeq_p^{(n-1)} N' \longrightarrow_e N_1',$$

we obtain $N_0 \longrightarrow_e^* N_2 \succeq_p^{(m')} N_1'$ for some N_2 and m'. Let $N_1 = \Lambda x.N_2\ x$. Then we have $N \longrightarrow_e^* N_1$ and $N_1 \succeq_p^{(m'+1)} N_1'$ as required.

- Cases SP-Cons and SP-Abs: These cases cannot happen as N' is irreducible with respect to $\longrightarrow_e$.
- Case SP-Abs2: In this case, we have:

$$N = \Lambda x.N_0$$
$$N' = \Lambda x.N_0'$$
$$N_1' = \Lambda x.N_0''$$

where $N_0 \succeq_p^{(n)} N_0' \longrightarrow_e N_0''$. By the induction hypothesis, we have $N_0 \longrightarrow_e P^* \succeq_p^{(m)} N_0''$ for some P and m. Thus, the required condition holds for $N_1 = \Lambda x.P$.

- Case SP-App: In this case, we have:

$$N = P\,Q \succeq_p^{(p+q)} P'\,Q' = N'$$

with $P \succeq_p^{(p)} P'$ and $Q \succeq_p^{(q)} Q'$. We perform case analysis on N_1'.
  - Case $N_1' = P_1'Q'$ where $P' \longrightarrow_e P_1'$ In this case, we have $P \longrightarrow_e^* P_1 \succeq_p P_1'$ for some P_1 by the induction hypothesis. Thus, the required condition holds for $N_1 = P_1\,Q'$
  - Case $P' = \lambda x.R'$ and $N_1' = [Q'/x]R'$ In this case, by Lemma A.16, we have $N \longrightarrow_e^* (\lambda x.R)\,Q$ and $R \succeq_p R'$ for some R. Let $N_1 = (\lambda x.R)\,Q$. Then we have $N \longrightarrow_e^* N_1 \succeq_p [Q'/x]R' = N_1'$ as required.
- Case SP-If0: In this case, we have:

$$N = \texttt{if0}\ P\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2$$
$$N' = \texttt{if0}\ P'\ \texttt{then}\ Q_1'\ \texttt{else}\ Q_2'$$

where $P \succeq_p^{(p)} P'$ and $Q_i \succeq_p^{(q_i)} Q_i'$. We perform case analysis on N_1'.
  - Case $N_1' = \texttt{if0}\ P_1'\ \texttt{then}\ Q_1'\ \texttt{else}\ Q_2'$ where $P' \longrightarrow_e P_1'$. In this case, we have $P \longrightarrow_e^* P_1 \succeq_p P_1'$ for some P_1 by the induction hypothesis. Thus, the required condition holds for

$$N_1 = \texttt{if0}\ P_1\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2.$$

  - Case $P' = k$ with $N_1' = Q_i$. In this case, we have $P \longrightarrow_e^* k$ by Lemma A.15. Let

$$N_1 = \texttt{if0}\ k\ \texttt{then}\ Q_1'\ \texttt{else}\ Q_2'.$$

Then $N_1 \succeq_p N_1'$ is obtained from SP-Beta3 or SP-Beta4. Thus, the required condition holds.

- Case SP-Case: In this case, we have:

  $N = \mathtt{case}\ P\ \mathtt{of}\ (pat_1 \Rightarrow Q_1 \mid \cdots \mid pat_k \Rightarrow Q_k)$

  $N' = \mathtt{case}\ P'\ \mathtt{of}\ (pat_1 \Rightarrow Q'_1 \mid \cdots \mid pat_k \Rightarrow Q'_k)$

  where $P \succeq_p^{(p)} P'$ and $Q_i \succeq_p^{(q_i)} Q'_i$. We perform case analysis on $N' \longrightarrow_e N'_1$.
  - Case where $N' \longrightarrow_e N'_1$ is obtained by reducing P'. In this case, we have: $N'_1 = \mathtt{case}\ P'_1\ \mathtt{of}\ (pat_1 \Rightarrow Q'_1 \mid \cdots \mid pat_k \Rightarrow Q'_k)$ where $P' \longrightarrow_e P'_1$. By the induction hypothesis, we have $P \longrightarrow_e^* P_1 \succeq_p P'_1$ for some P_1 Thus, the required condition holds for $N_1 = \mathtt{case}\ P_1\ \mathtt{of}\ (pat_1 \Rightarrow Q'_1 \mid \cdots \mid pat_k \Rightarrow Q'_k)$
  - Case where $N' \longrightarrow_e N'_1$ is obtained by reducing the case expression. In this case, we have $P' = L' \vdash pat_i \Rightarrow \rho'$ and $N'_1 = \rho' Q'_i$. By Lemma A.15, we have $P \longrightarrow_e^* L \succeq_p L'$ with $L \vdash pat_i \Rightarrow \rho$ and $\rho \succeq_p \rho'$; here the relation $\succeq_p$ has been pointwise extended to the relation on substitutions. By Lemma A.14, we have $\rho Q_i \succeq_p \rho' Q'_i$. Thus, the required conditions hold for $N_1 = \rho Q_i$.
- Case SP-Fix: In this case, we have:

  $$N = \mathtt{fix}\ x.N_0$$
  $$N' = \mathtt{fix}\ x.N'_0$$
  $$N'_1 = [\mathtt{fix}\ x.N'_0/x]N'_0$$

  where $N_0 \succeq_p N'_0$. By Lemma A.14, we have $[\mathtt{fix}\ x.N_0/x]N_0 \succeq_p [\mathtt{fix}\ x.N'_0/x]N'_0$. Thus, the required conditions hold for $N_1 = [\mathtt{fix}\ x.N_0/x]N_0$.
- Case SP-Beta1 In this case, we have:

  $N = (\lambda x.P_1)\ P_2 \succeq_p^{(p_1+p_2\times Oc(x,P'_1)+1)} [P'_2/x]P'_1 = N'$

  where

  $$P_1 \succeq_p^{(p_1)} P'_1$$
  $$P_2 \succeq_p^{(p_2)} P'_2.$$

  By Lemma A.14, we get

  $[P_2/x]P_1 \succeq_p^{(p_1+p_2\times Oc(x,P'_1))} [P'_2/x]P'_1 (= N')$.

  By applying the induction hypothesis to

  $[P_2/x]P_1 \succeq_p^{(p_1+p_2\times Oc(x,P'_1))} N' \longrightarrow_e N'_1$,

  we obtain

  $$[P_2/x]P_1 \longrightarrow_e^* N_1 \succeq_p N'_1$$

  for some N_1. Thus, we have $N \longrightarrow_e^* N_1 \succeq_p N'_1$ as required.
- Case SP-Beta2: In this case, we have:

  $N = \mathtt{fix}\ x.P_1$
  $\succeq_p^{(p_1\times(Oc(x,P'_1)+1)+1)} [\mathtt{fix}\ x.P'_1/x]P'_1$
  $= N'$

  where

  $$P_1 \succeq_p^{(p_1)} P'_1.$$

  By Lemma A.14, we have

  $[\mathtt{fix}\ x.P_1/x]P_1 \succeq_p^{(p_1\times(Oc(x,P'_1)+1))} N'$.

  By applying the induction hypothesis to

  $[\mathtt{fix}\ x.P_1/x]P_1 \succeq_p^{(p_1\times(Oc(x,P'_1)+1))} N' \longrightarrow_e N_1$,

  we obtain

  $$[\mathtt{fix}\ x.P_1/x]P_1 \longrightarrow_e^* N_1 \succeq_p N'_1$$

  for some N_1. We have thus $N \longrightarrow_e^* N_1 \succeq_p N'_1$ as required.
- Case SP-Beta3 or SP-Beta4 In this case, we have:

  $N = \mathtt{if0}\ m\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$
  $\succeq_p^{(p_i+1)} P'_i$

  where

  $$P_i \succeq_p^{(p_i)} P'_i.$$

  By applying the induction hypothesis to $P_i \succeq_p^{(p_i)} P'_i = N \longrightarrow_e N'_1$, we obtain

  $$P_i \longrightarrow_e^* P''_i \succeq_p N'_1$$

  for some P''_i. Thus, the required conditions hold for $N_1 = P''_i$.
- Case SP-Beta5 In this case, we have:

  $N = \mathtt{case}\ L\ \mathtt{of}\ (pat_1 \Rightarrow P_1 \mid \cdots \mid pat_k \Rightarrow P_k)$
  $\succeq_p^{(p_i+1)} P'_i$
  $= N'$

  where

  $$P_i \succeq_p^{(p_i)} P'_i.$$

  By applying the induction hypothesis to $P_i \succeq_p^{(p_i)} P'_i = N' \longrightarrow_e N_1$, we obtain

  $$P_i \longrightarrow_e^* P''_i \succeq_p N'_1$$

  for some P''_i. Thus, the required conditions hold for $N_1 = P''_i$.
- Case SP-Beta6 In this case, we have:

  $N = \mathtt{case}\ L\ \mathtt{of}\ (pat_1 \Rightarrow P_1 \mid \cdots \mid pat_k \Rightarrow P_k)$
  $\succeq_p^{(p_i+l\times Oc(y,P'_i)+1)} [L'_1/y]P'_i$
  $= N'$

  where $L \vdash pat_i \Rightarrow [L_1/y]$ with $L_1 \succeq_p^{(l)} L'_1$ and $P_i \succeq_p^{(p_i)} P'_i$. By Lemma A.14, we have

  $$[L_1/y]P_i \succeq_p^{(p_i+l\times Oc(y,P'_i))} [L'_1/y]P'_i.$$

  By applying the induction hypothesis to

  $[L_1/y]P_i \succeq_p^{(p_i+l\times Oc(y,P'_i))} [L'_1/y]P'_i = N' \longrightarrow_e N'_1$,

we obtain

$$[L_1/y]P_i \longrightarrow_e^* Q' \geq_p N_1'$$

for some Q'. Thus, the required condition holds for $N_1 = Q'$.

□

We are now ready to show Lemma 3.15.

Proof of Lemma 3.15. Suppose N_0 is a closed term of type `int` and $N_0 \geq N_1 \longrightarrow_e^* V$. By Lemma A.17, we have $N_0 \longrightarrow_e^* N \geq_p V$ for some N. By Lemma A.15, we have $N \longrightarrow_e^* V$. Thus, we have $N_0 \longrightarrow_e^* V$ as required. □