The Semantics of Delimited Control in Sequential and Parallel Languages

November 2013

Asami Tanaka

The Semantics of Delimited Control in Sequential and Parallel Languages

Graduate School of Systems and Information Engineering

University of Tsukuba

November 2013

Asami Tanaka

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of programming language research is to make it easier to write, read, maintain, reuse and verify programs written in those languages. The key to achieve it is abstraction, a language mechanism to separate a specific feature from programs, and thus reflect the semantic structure. In the literature, various abstraction mechanisms have been studied: Dijkstra proposed structured programming and Liskov abstract data types, to name just a few.

Control abstraction is one of such abstractions, but to structure control mechanisms. For instance, we often need non-local exits, namely, we want to return from multiple function calls immediately. One can implement non-local exits using standard primitives, for instance, by attaching to all return values one-bit flag, which distinguishes non-local returns from normal returns. Such an implementation is hard to read and write because we need to insert a small code to check the flag at every point of the program which returns a value. Control abstraction helps; if the language has the exception mechanism in C++, Java, Ruby and ML, we can localize the codes for non-local exits, and the above interleaving is not needed.

We study control abstraction for continuations. A continuation is a notion in program execution, which means the rest of a computation. Intuitively it corresponds to the control stack, onto which the remaining computations are pushed. Language primitives to abstract continuations are called control operators, and a quite popular control operator is `call/cc` in Scheme [29]. While a continuation represents the whole rest of a computation, we sometimes want to manipulate only part of the rest of a computation, which is called a delimited continuation. In recent years, control operators for delimited continuations (delimited-control operators, for short) have been studied intensively, and `shift`/`reset` [9] and `control`/`prompt` [12] are such ones.

While many studies about delimited control are done in Call-by-Value calculi, in which programs are executed sequentially, we want to take advantages of delimited continuations in other evaluation strategies such as Call-by-Name and parallel computations. It is natural that we want to abstract controls in programs running under such evaluation strategies. However it is not so easy to define and use control abstraction mechanisms in Call-by-Name setting or parallel computation; one of the problems is that the meaning of (delimited) continuations is not obvious. In our research, we pursue to get the advantages of control operators in these evaluation strategies by addressing their problems. In order to achieve that, we will define several calculi in which delimited control operators are defined as their primitive features and discuss the usefulness of them.

The merits of treating control operators as primitives in programming languages are as follows:

- We can define a rigorous semantics of delimited control operators,
- We can design a proper type system of delimited control operators, and
- We can expect the optimization done by a compiler.

There are other ways to use delimited control operators without having them as primitives. For example, they can be presented as a library module, which is like "DelimCC library" of OCaml [31].

We can also propose control abstraction mechanisms as frameworks for specific domains such as making web applications [53]. Web applications are typical examples whose control structures are scattered around source programs because, as HTTP is a stateless protocol, each interaction between a web application and its client is treated separately so that codes disposing of these interactions are scattered around the web application. This problem forces programmer to

divide a web application into a number of code fragments, thus making himself be concerned with the control flow all the time; the control flow is not abstracted.

In this thesis, we discuss the semantic foundations of delimited-control operators under several evaluation orders. We will mainly discuss two topics: the first one is how delimited control operators are given semantics and formally treated in a Call-by-Name calculus, and the second is how they should behave as useful primitives in a parallel calculus.

The former consists of two themes: the one is about axiomatizing a Call-by-Name calculus with delimited control operators, and the other is about how to extend our results obtained in it to a calculus with layered-delimited control operators. We can extend an expressive power of a Call-by-Name calculus even more if we can introduce delimited control operators in a Call-by-Name calculus. Furthermore, by the second research, we can use layered delimited-control operators, which has more expressive power than delimited-control operators do, in Call-by-Name calculus.

The latter is about a calculus which has two features: delimited control operators and *parallelizing* operators. Given a program, we often find its subprograms (or program points) able to be executed in parallel. However, it is not easy to execute multiple subprograms in parallel if the program contains (delimited) control operators, as the semantics of the latter is defined in terms of sequential computations. In our research, we propose a calculus in which programmers can use delimited control operators and manipulate delimited continuations as in non-parallelized programs. Our key design principle is that parallel computations should not affect the result of the computations even if programs contain delimited-control operators, i.e., parallelization must be transparent. In this paper, we prove the transparency of our calculus by extending Takahashi's parallel reduction method, which was originally invented to prove confluence of the lambda calculus [52]. Furthermore, we will propose an implementation of our transparent parallel semantics. While the implementation is a proof of concept, we can show that our parallelization mechanism have a certain real speed up.

The rest of this paper is as follows. In Chapter 2, we will introduce several basic definitions and results in the literature. Then we introduce our research about Call-by-Name calculus in Chapter 3 and Chapter 4. In Chapter 5, we will propose a calculus with delimited control operators and parallelizing operators, and we will describe our implementation of the parallel semantics in Chapter 6. Finally, we will conclude in Chapter 7.

Chapter 2

Preliminary

2.1 Evaluation Strategy

Evaluation strategies determine the order of evaluations of arguments when function-application expressions are evaluated. In this section, we will describe the differences of two evaluation strategies: Call-by-Value and Call-by-Name.

2.1.1 Call-by-Value and Call-by-Name

There are two most fundamental evaluation strategies. The first one, called Call-by-Value (CbV for short), is to evaluate all the arguments of function-application expressions before the function bodies, and the second one, called Call-by-Name (CbN for short), is to evaluate no arguments before the function bodies, but to evaluate them when their values are needed.

For example, let us consider the following program (`print` is a primitive function to show its arguments to a display, and semicolon denotes sequential executions):

```
1 let f = fun x y -> x + 1
2
3 let g = fun v -> print v; v
4
5 f (g 10) (g 20)
```

In a CbV language, this program prints two integers: 10 and 20. In a CbN language, it prints 10 only. The reason is that, in the CbN language, the arguments of the function-application expression `f (g 10) (g 20)` are not evaluated when the expression is evaluated. Since the body of the function `f` needs the value of the first argument `x` before addition expression is executed, the unevaluated expression `(g 10)` is evaluated. However, the second argument `y` is not used through the evaluation of the function body so the expression `(g 10)` will never be evaluated.

Since CbN does not have to evaluate all the arguments of function-application expressions, CbN languages have some advantages compared to CbV languages as follows:

- it is easier to write meta-level programs such as program translations than that in CbV languages,
- it is able to express infinite computations easily.

2.2 Control Operators and Continuations

Control operators are useful to abstract control in programs, and the continuation is one of the most important concepts for abstracting controls.

Since we will discuss control operators continuously in this paper, we will introduce control operators and continuations in this section: we will describe operators to manipulate continuations and what the continuations are by showing several examples. Then, in Section 2.3, we will show an example of a CbV calculus including control operators with its formal definitions.

2.2.1 Continuations in CbV

The concept of continuations originally means that the rest of the computation, namely, a function-like object which, given the value of the current subcomputation, returns the answer (the final value of the whole computation). Thus, a continuation varies depending on which part of program is being executed.

For example, we have the following expression and let the addition with 10 and 100 be being executed:

$$((10 + 100) + 1000) + 10000$$

The current continuation can be represented as a function-like object, which gets the result of the current subcomputation $(10 + 100)$, and returns the answer of $(v + 1000) + 10000$ where v is the result of the current one.

Continuations will change as the computation goes on. After computing $(10+100)$, the subcomputation $(110+1000)$ is being executed, and hence the current continuation is an object to return the answer of $v' + 10000$ where v' is the result of current subcomputation (namely, 1110).

We show another example, which creates a function and applies it to an integer value:

```
1 let f = fun x -> x + 10
2
3 let g = (f 100) + 1000
```

The above program works as follows: first it defines a variable `f` and binds a function to it. Then a variable `g` is defined and bound to the result of the expression in the right-hand side. In this example, the continuation when the expression `(f 100)` is being computed is `(v + 1000)` where `v` is the result of the current subcomputation.

2.2.2 Control Operators in CbV

A wide variety of control operators have been designed and implemented in many programming languages: `goto` operator in C, exception handler in Java and ML, `yield` operator in Lua and so on.

In particular, control operators which manipulate continuations can be used to abstract the control structures. By capturing, saving or restoring continuations, surprisingly many kinds of control structures may be expressed. A simple example is the exception mechanism with the operators `try` and `raise`, which can be implemented using continuations as follows:

(1) when a `try` operator is executed, the current continuation is captured and saved in a global storage,

(2) during the execution of the body of the `try` operator, if a `raise` operator is executed, the execution aborts, namely, the current continuation is discarded, and the saved continuation (corresponding to the `try` operator) is restored from the global storage, and

(3) during the execution of the body of the `try` operator, if a `raise` operator is not executed, the stored continuation is discarded.

Also coroutines can be implemented by using continuations as follows:

(1) when a coroutine is started (i.e., another process is called first time), the current continuation is saved, and

(2) when a process suspends and another process is called, the current continuation is saved, and instead the saved continuation is restored and used as the current continuation.

An overview of control operators in the literature are shown in Table 2.1. While the similar concepts were realized in the literature, the ones who made clear the concept of continuation were Strachey and Wadsworth [49] in their studies on denotational semantics.

Control operators were developed before the discoveries of continuations. Landin proposed J operator [34] to control the stack of an abstract machine in 1965, and Reynolds proposed the escape operator [44] in 1972. The escape operator works in the same way as `call/cc` which is one of the most popular control operators in Scheme.

In 1980's and 90's, several researchers noticed that it is more useful to *delimit* the continuation. Here, "delimit" means we only extract a certain part of a continuation where the part is specified by some primitives.

	Authors	Implementations
J operator	Landin [34]	J (Algol)
Exception	-	`catch/throw` (Lisp), `exception` (C++, ML et al.)
First-class continuation	Steele and Sussman [48]	`call/cc`(Scheme)
First-class delimited continuation	Felleisen [12], Danvy and Filinski [9]	`shift/reset` (OCaml/delimcc, Racket, Scala)

Table 2.1: Control Operators

	Call-by-Value Calculus	Call-by-Name Calculus	Parallel Calculus
`call/cc`	Felleisen [13] and many others	derivation from λ_μ calculus : Parigot [42] ’	Moreau [38], Komiya and Yuasa [33]
`control/prompt`	a calculus : Felleisen [12]	-	-
`shift/reset`	a calculus : Danvy and Filinski [9], and many others	a calculus : Biernacki [6], axiomatization : This Thesis	This Thesis
layered-`shift/reset`	a calculus : Danvy and Filinski [9], axiomatization : Kameyama [25]	This Thesis	-

Table 2.2: Calculi with Control Operator

In this manner, Felleisen proposed `control/prompt` operators [12] in 1989, and Danvy and Filinski proposed `shift/reset` operators [9] in 1990. These operators are called “delimited-control operators” and the continuations captured by `control` and `shift` are called “delimited continuations”.

Since then, many kinds of delimited-control operators are proposed; Hieb and Dybvig proposed the spawn operator [24] which can be used in a concurrent setting with tree-structured processes, Gunter et al. defined the cupto operator [20] which generalizes the behavior of delimiters, and Dyvbig et al. presented the multi-prompt delimited-control operator. [11].

Though we have many choices which operator(s) we use in this research, we decided to study `shift/reset` because of the important properties of the operators which will mention later. The semantics of control operators have been studied in various evaluation strategies. Examples of such research are shown in Table 2.2, and our contributions are also written in it.

In terms of axiomatization, we study a CbN calculus with the delimited-control operator and propose its equational axioms for the first time. Furthermore, we study layered-`shift/reset` in a CbN calculus and its axiomatization is also our original work. The details of equational axiomatization will be shown in Chapter 3, and, in Chapter 4, we will show how to connect the calculus with layered-`shift/reset` to existing results in other research.

From the point of semantics, our main targets are a CbN calculus with layered-`shift/reset` and a calculus which has both delimited-control operator `shift/reset` and parallel operator `future`. The former makes it possible for us to write programs with multiple effects, which we show in detail in Chapter 4. By using the latter calculus, we can write effectful programs whose sub-programs can be executed in parallel. In Chapter 5, we will propose the calculus with its rigorous semantics, and present comparisons between our parallel semantics and other ones.

In the next subsection, we will explain delimited continuations and delimited-control operators. In this research, `shift/reset` are main tools so we will explain them.

2.3 Shift/Reset in Call-by-Value Calculus

`shift/reset` are the delimited-control operators proposed by Danvy and Filinski. The operator `shift` is for capturing the delimited continuations, and the operator `reset` is for delimiting the extent of the continuations.

In our research, we mainly focus `shift/reset` operators because of as follows:

(1) delimited-control operators are more expressive compared to the non-delimited-control operators such as `call/cc`,

(2) a rigid semantics through a CPS translation is given for `shift/reset`, and

(3) `shift`/`reset` can express all monadic effects [15].

In this section, we show a calculus with `shift`/`reset` and call it $\lambda_{s/r}^{cbv}$. Before we show the formal semantics of $\lambda_{s/r}^{cbv}$, we will describe the behavior of `shift`/`reset` intuitively by showing some examples.

2.3.1 Small Example

In this subsection, we use OCaml-like syntax to write example programs. They may not run directly in OCaml, and one needs an implementation of `shift`/`reset`, such as the DelimCC library [31].

In the following, let $\mathcal{S}$ be the `shift` operator and $\langle - \rangle$ be the `reset` operator. And $\rightsquigarrow$ means one-step reduction and $\rightsquigarrow^*$ is the reflexive, transitive closure of $\rightsquigarrow$.

First we will show a simple arithmetic example.

$$
\begin{aligned}
e &= 1 + \langle 10 + \mathcal{S}k.(100 + k\ (k\ 1000)) \rangle \\
&\rightsquigarrow 1 + \langle 100 + k\ (k\ 1000) \rangle && \text{where } k = \lambda v.\langle 10 + v \rangle \\
&\rightsquigarrow 1 + \langle 100 + k\ \langle 10 + 1000 \rangle \rangle \\
&\rightsquigarrow 1 + \langle 100 + k\ \langle 1010 \rangle \rangle \\
&\rightsquigarrow 1 + \langle 100 + k\ 1010 \rangle \\
&\rightsquigarrow^* 1 + \langle 100 + 1020 \rangle \\
&\rightsquigarrow^* 1121
\end{aligned}
$$

An expression e is defined in the first line and computation starts from this. At the first step, `shift` captures the continuation up to `reset`. In this time, the continuation is $10+\Box$ where $\Box$ means the place to put the result of computation being executed now. The captured continuation is removed from the expression and it is preserved as a function-like object. An important point is that the continuation contains `reset` in the body of the object. In the third line, the continuation was applied so the expression $(k\ 1000)$ turns into an expression $\langle 10+1000 \rangle$, then turns $\langle 1010 \rangle$. The `reset` operator around the value is not significant so it is simply removed at the next step, in the 5th line. Also the expression $k\ 1010$ turns into the value 1020, and finally we get the result 1121 in the last line.

2.3.1.1 List Append

The following example is an append function for lists, which uses `shift`/`reset`. (This example was taken from the literature [8, 4]). In this program, `shift` represents the `shift` operator and `shift k -> e` means that `shift` captures the continuation up to the nearest reset and binds it to the variable `k`, then runs the expression `e`. Also `reset` represents the `reset` operator and `reset e` means the expression surrounded by `reset`.

```
1| let rec append' lst = match lst with                                ||
2|   | [] -> shift k -> k                                             ||
3|   | x :: xs -> x :: (append' xs)                                   ||
4|                                                                    ||
5| let append lst1 lst2 =                                            ||
6|   let append1 = reset (append' lst1) in                           ||
7|   append1 lst2                                                    ||
```

The function `append'` gets one argument `lst`, which is a list, and examines its structure. If `lst` is empty, `shift`-expression is executed, and the delimited continuation (up to the nearest `reset`) is captured and bound to the variable `k`. The return value in this case is the value of `k`, namely, the delimited continuation. Otherwise, the argument is decomposed into a head and a tail, and then the function is recursively called on the tail of the list, and returns the cons of the head of the list with the result of the recursive call. So the second case is the same as the ordinary implementation of the append function except that `append'` gets only one argument rather than two. The trick happens when `lst` is empty. In the ordinary append function, it returns the second argument as its result when the first argument is empty, while this `append'` function returns the current delimited continuation, which serves as a function-like object whose input is the (missing) second argument, and output is the appended list.

Let us see more concretely what is happening here. We take

```
reset (append' [1; 2; 3])
```

as an example [1], which runs as follows:

```
  reset (append [1; 2; 3])
⇝ reset (1 ::  (append [2; 3]))
⇝ reset (1 ::  (2 ::  (append [3])))
⇝ reset (1 ::  (2 ::  (3 ::  (append []))))
⇝ reset (1 ::  (2 ::  (3 ::  (shift k -> k))))
```

At this point, `shift` operator captures the continuation as a function as follows: `fun v -> reset (1 :: (2 :: (3 :: v)))`.

```
⇝ reset (1 ::  (2 ::  (3 ::  (shift k -> k))))
⇝ reset (fun v -> reset (1 ::  (2 ::  (3 ::  v))))
⇝ fun v -> reset (1 ::  (2 ::  (3 ::  v)))
```

The function `append` gets two lists to append one to another. The first argument, `lst1`, is passed to the function `append'` with `reset`. The result, a function above, is bound to the variable `append1` and applied to the second value `lst2`.

For example, the expression `append [1; 2; 3] [10; 20; 30]` works as follows: first it computes the expression `reset (append' [1; 2; 3])` and gets a function (as above). Then it applies the function to the list `[10; 20; 30]`. As a result of that, it gets a value `[1; 2; 3; 10; 20; 30]`.

2.3.1.2 Non-deterministic Choice

We can implement non-deterministic computation by using `shift`/`reset`. We first implement two functions `fail` and `flip` in order to make the main procedure more readable.

```
1| let fail () = shift k -> ()
2|
3| let flip () = shift k -> k true; k false
```

The function `fail` simply executes the shift-term, and returns the value `()` which is a meaningless value. When the shift-term in the function `flip` is executed, a delimited continuation is captured and bound to the variable `k`. Then, `flip` uses the captured continuation twice; in `k true` and `k false`.

As a substantial example of these delimited-control operators, we show an implementation of a tree-traversal function. Here, the function `is_prime` represents a predicate to judge whether the argument is prime, and returns a boolean value `true` if the argument is a prime number, and `false` otherwise. Also `print_elm` is a primitive function to print the argument to the screen.

```
 1| type t = Leaf of int | Node of t * t
 2|
 3| let rec trav tree pred proc = match tree with
 4|   | Leaf n ->
 5|     if pred n then proc n
 6|     else fail ()
 7|   | Node (t1, t2) ->
 8|     if flip () then trav t1 pred proc
 9|     else trav t2 pred proc
10|
11| let tree =
12|   Node (Node (Leaf 12, Leaf 23),
13|         Node (Leaf 36,
14|               Node (Leaf 45, Leaf 53)))
15|
16| let pred = fun n -> is_prime n
17|
18| let proc = fun n -> print_elm n
19|
20| reset (trav tree pred proc)
```

[1] the notion vb[1; 2; 3] represents a list which contains three elements

In the first line, a type for binary trees is defined. This type consists of two kinds of values distinguished from each other by the tags `Leaf` and `Node`. The value with `Leaf` has an integer value and `Node` has two values whose type is `t`. An example value of type `t` is defined between 11th line to 14th line in the above program.

The function `trav` gets three arguments:

- `tree` is a value as a type `t`,
- `pred` is a predicate function which gets an integer value and judges whether the value is acceptable or not, and
- `proc` is a task function which is applied to the `tree` (precisely, the value contained in the tree value) when `pred tree` returns `true`.

When the function `trav` gets these kinds of arguments, it first examines the structure of the argument `tree`, and

- if the value `tree` has the tag `Leaf`, the integer value contained in that is bound to the variable `n`, and apply the function `pred` to the `n`. Then,
  - if the application expression `pred n` returns the value `true`, it applies `proc` to the `n`, and returns the result of application,
  - otherwise, the application expression `fail ()` will be executed, or
- if the tree has the tag `Node`, the two other values contained in it are bound to the variable `t1` and `t2` respectively, then the if-expression following the matching-expression is executed.

The `if` expression in the 9th and 10th lines are the most important part of this program. As we described above, `flip` captures the delimited continuation and applies it to two values, `true` and `false` which means that this `if` expression will run twice with the values `true` and `false`.

To show the behavior of the `flip`, we will explain several steps of the evaluation of the expression as follows. In the following program, we will use the variables `tree`, `pred` and `proc` which are defined at the line 11, 16, 18 in the above program.

```
  reset (trav tree pred proc)
⇝ reset (if flip () then trav t1 pred proc else trav t2 pred proc)
  where
   t1 = Node (Leaf 12, Leaf 23)
   t2 = Node (Node (Leaf 36, Node (Leaf 45, Leaf 53)))
⇝ reset (k true; k false)
  where
   k = fun v -> reset (if v then trav t1 pred proc else trav t2 pred proc)
```

Then, the subexpression `k true` of the last line runs as follows:

```
    k true
⇝ reset (if true then trav t1 pred proc else trav t2 pred proc)
⇝ reset (trav t1 pred proc)
= reset (trav (Node (Leaf 12, Leaf 23)) pred proc)
⇝ reset (if flip () then trav (Leaf 12) pred proc
         else trav (Leaf 23) pred proc)
⇝ reset (k' true; k' false)
  where
   k' = fun v -> reset (if v then trav (Leaf 12) pred proc
              else trav (Leaf 23) pred proc)
```

$$
\begin{array}{lll}
v \stackrel{\text{def}}{=} x \mid \lambda x.e & & \textit{value} \\
e \stackrel{\text{def}}{=} v \mid e_1\ e_2 \mid \mathcal{S}k.e \mid \langle e \rangle & & \textit{expression} \\
E \stackrel{\text{def}}{=} \Box \mid E\ e \mid v\ E \mid \langle E \rangle & & \textit{evaluation context} \\
F \stackrel{\text{def}}{=} \Box \mid F\ e \mid v\ F & & \textit{pure-evaluation context}
\end{array}
$$

Figure 2.1: Syntax of $\lambda^{cbv}_{s/r}$

$$
\begin{array}{rll}
E[(\lambda x.e)\ v] \rightsquigarrow E[e\{x := v\}] & & \beta_v \\
E[\langle v \rangle] \rightsquigarrow E[v] & & rv_v \\
E[\langle F[\mathcal{S}k.e] \rangle] \rightsquigarrow E[\langle e\{k := \lambda v.\langle F[v] \rangle\} \rangle] & & rs_v
\end{array}
$$

Figure 2.2: Reduction rules of $\lambda^{cbv}_{s/r}$

The application-expression `k' true` in the last expression above is executed to

```
reset (if (pred (Leaf 12)) then proc () else fail ())
```

Then, since 12 is not a prime number, `pred 12` returns `false` so `fail ()` will be executed. When the function `fail` is invoked, `shift` captures the current delimited continuation, namely, the `if`-expression in the body of the `k'`, and discards the current delimited continuation. Then `k' false` will be evaluated; `pred 23` returns `true`, `proc 23` is evaluated, and the integer `23` is printed on the screen.

In summary, `flip` works as follows:

(1) `shift` captures the `if` expression surrounding the `flip` expression,

(2) the captured continuation is applied to a value `true`, and

(3) the captured continuation is also applied to the other value `false`.

By this way, we can write programs which select appropriate values non-deterministically. In other words, non-deterministic programs using the `flip` function can select appropriate values automatically if we do not write choosing (or traversing) code.

2.3.2 Operational Semantics

In this section, we will define the operational semantics for the calculus $\lambda^{cbv}_{s/r}$. The syntax and reduction rules of $\lambda^{cbv}_{s/r}$ are shown in Figure 2.1 and Figure 2.2, respectively.

In Figure 2.1, the syntax of $\lambda^{cbv}_{s/r}$ is defined. It says that there are two kinds of values, x and $\lambda x.e$, and we use a meta variable v for them. Also there are four kinds of expressions. One of them is a value, and the rest consists of a function application, a shift-expression and a reset-expression. Though we use some additional expression such as list operations and an if-expression in the later subsection but we omit them in the above definition.

The metavariables E and F define evaluation contexts used to define the evaluation order of the function application and the rules of `shift`/`reset`. The definition of evaluation contexts E and F represent that, when an application-expression is evaluated, first the left-hand side of the application-expression (i.e., the applied unction) is evaluated, and then the right-hand side expression (i.e., the argument) is evaluated. This behavior is defined in Figure 2.2. The difference

of the E and F is whether `reset` is included in the context. By writing an expression $E[\langle F[\mathcal{S}k.e]\rangle]$, we can specify this `reset` as the nearest `reset` from this `shift`.

In Figure 2.2, we show the behavior of the expression defined in Figure 2.1. The first line is a β reduction as usual, and $e\{x := v\}$ is a capture-avoiding substitution. The second line is a rule for removing a meaningless occurrence of `reset`. The last rule defines how the `shift` operator works. When `shift` works with pure-evaluation context F, it reifies the continuation as a function $\lambda v.\langle F[v]\rangle$ with a fresh variable v. Then it substitutes the continuation for the variable k.

This direct definition of the semantics of $\lambda^{cbv}_{s/r}$ is simple and clear as to what is defined. Since there are alternative ways to define the semantics of $\lambda^{cbv}_{s/r}$, we will explain the one of the alternatives in the next subsection.

2.3.3 CPS Semantics

A CPS translation is a global program translation from a program to a program in a specific form, namely, in continuation-passing style (CPS). A CPS translation was first proposed by Plotkin [43] which has the following characteristic features:

(1) All CPS functions explicitly pass continuations as their argument, and

(2) All function calls in CPS programs are tail calls.

(3) All intermediate results in CPS programs are explicitly named.

For example, let us consider the following program which computes the factorial of is argument:

```
1| let rec fact n =
2|   if n = 0 then 1
3|   else n * (fact (n - 1))
```

This program is not in CPS because (1) it does not call `fact` as a tail call, and (2) the continuation of each (recursive) expression is implicitly: it is not passed as an explicit expression and not applied explicitly.

We can convert the above program into CPS as follows:

```
1| let rec fact_n n k =
2|   if n = 0 then k 1
3|   else fact_n (n - 1) (fun v -> k (n * v))
```

This program differs from the above `fact` in two points:

(1) it takes two arguments, and

(2) it calls `fact_n` at the end of its body.

The first is because it gets one more argument which represents a continuation. The second is clear if we compare the two programs; instead of multiplication with `n` and the result of `fact (n - 1)`, `fact_n` calls itself.

In CPS, we can explicitly find continuations. It makes it possible for us to manipulate continuations in any point and to modify it as we want. Although it is often difficult to write programs in CPS, we can get any programs in CPS by CPS translating them.

In the literature [43], a CPS translation for λ calculus was defined as in Figure 2.3. The above definition converts an arbitrary expression in λ-calculus into its CPS. By CPS translating a program, we can make explicit the evaluation order in the program, and thus making its meaning clearer. For example, the meaning of the application is as follows:

(1) it gets (captures) its continuation (κ),

(2) evaluates its function part e_1 with a continuation which

  (a) gets the result of the evaluation of e_1 and binds it to the variable f,

  (b) evaluates its argument part e_2 with the continuation which

    i. gets the result of the evaluation of e_2 and binds it to the variable a,

    ii. applies the f to a and κ.

$$
\begin{aligned}
[\![x]\!] &\stackrel{\text{def}}{=} \lambda\kappa.\kappa\ x \\
[\![\lambda x.e]\!] &\stackrel{\text{def}}{=} \lambda\kappa.\kappa\ (\lambda x.[\![e]\!]) \\
[\![e_1\ e_2]\!] &\stackrel{\text{def}}{=} \lambda\kappa.[\![e_1]\!]\ (\lambda f.[\![e_2]\!]\ (\lambda a.f\ a\ \kappa))
\end{aligned}
$$

Figure 2.3: CPS translation for call-by-value λ calculus

Given an ordinary program, we can make its continuation explicit, by translating it into CPS, which allows us to precisely define the meaning of control operators. For instance, the (non-delimited) control operator `call/cc` can be define by CPS translation as follows:

$$
[\![\texttt{call/cc}\ k.e]\!] \stackrel{\text{def}}{=} \lambda\kappa.\texttt{let}\ k = \lambda v.\lambda\kappa'.\kappa\ v\ \texttt{in}\ [\![e]\!]\ \kappa
$$

This definition means that `call/cc` first gets the continuation (κ) and binds the function to the variable k, and then it evaluates the expression e without aborting its the continuation. The bound continuation is a function which gets two values, a value v and a continuation κ', and it aborts κ' (which denotes the continuation of when an application of this continuation function was applied), and applies κ to its argument v (which denotes the continuation of when this continuation function was defined).

As control operators are used to manipulate continuations, CPS is an appropriate tool to define control operators because all continuations are defined and passed explicitly at arbitrary points of a program.

In the same manner, we can define the semantics of `shift`/`reset` by the CPS translation.

$$
\begin{aligned}
[\![\langle e\rangle]\!] &\stackrel{\text{def}}{=} \lambda\kappa.\kappa\ ([\![e]\!]\ (\lambda v.v)) \\
[\![\mathcal{S}k.e]\!] &\stackrel{\text{def}}{=} \lambda\kappa.\texttt{let}\ k = \lambda v.\lambda\kappa'.\kappa'\ (\kappa\ v)\ \texttt{in}\ [\![e]\!]\ (\lambda v.v)
\end{aligned}
$$

The translation of $\langle e\rangle$ reveals the behavior of `reset`: first $[\![e]\!]$ is evaluated by being applied to an empty continuation ($\lambda v.v$), then the result of the application is thrown to the current continuation (κ). This corresponds to the meaning of `reset` defined in the previous subsection, namely, `reset` delimits the extent of a continuation captured by `shift`.

The definition of $\mathcal{S}k.e$ is similar to that of `call/cc` $k.e$ but there are two differences:

(1) like `call/cc`, `shift` captures a continuation, turns it to a function, and binds it to the variable k. Unlike `call/cc`, the function created by `shift` is not abortive in the sense that, when the function is applied to an argument, the current continuation is not discarded.

(2) the expression $[\![e]\!]$ will be evaluated with an empty continuation ($\lambda v.v$), not κ.

We have so far defined two semantics of $\lambda^{cbv}_{s/r}$, one is defined directly and the other is defined through a CPS translation. One advantage of the latter is that the latter gives stronger equivalence (weaker as a binary relation) over open terms so that it can be used to justify many compiler optimizations directly [18]. To introduce equivalence relation using the former semantics, we must switch to contextual equivalence, which is even stronger than the CPS semantics, but is significantly harder to reason about.

However, the semantics based on CPS translations has some weak points such as

- need to translation; one must translate programs into CPS in order to obtain each meaning of programs, hence we cannot reason about programs,
- change of the surface; through CPS translation, programs change their surfaces, which makes it difficult to rewrite, refact or replace parts of the program.

$$\frac{}{\rho, x : \tau; \alpha \vdash x : \tau; \alpha}\ \textit{var} \qquad \frac{\rho, x : \sigma; \alpha \vdash e : \tau; \beta}{\rho; \delta \vdash \lambda x.e : (\sigma/\alpha \to \tau/\beta); \delta}\ \textit{fun}$$

$$\frac{\rho; \delta \vdash e_1 : (\sigma/\alpha \to \tau/\epsilon); \beta \quad \rho; \epsilon \vdash e_2 : \sigma; \delta}{\rho; \alpha \vdash e_1\ e_2 : \tau; \beta}\ \textit{app}$$

$$\frac{\rho; \sigma \vdash e : \sigma; \tau}{\rho; \alpha \vdash \langle e \rangle : \tau; \alpha}\ \textit{reset} \qquad \frac{\rho, k : (\tau/\delta \to \alpha/\delta); \sigma \vdash e : \sigma; \beta}{\rho; \alpha \vdash \mathcal{S}k.e : \tau; \beta}\ \textit{shift}$$

Figure 2.4: Type system of $\lambda^{cbv}_{s/r}$

2.3.4 Types

We will introduce a type systems for $\lambda^{cbv}_{s/r}$ in this subsection. Although there are several kinds of type systems, we will explain one of the type systems which was proposed by Danvy and Filinski, who proposed `shift`/`reset`. The type system is in Figure 2.4.

Unlike an usual type system for simply-typed λ calculus, each typing rule has a form as follows:

$$\rho; \alpha \vdash e : \tau; \beta$$

These type judgements can be read as follows:

> assume that an expression e has a type τ. If it is evaluated in a context C which has a type $\tau \to \alpha$, the resulting type of the evaluation of the expression $C[e]$ becomes β.

The reason that the type system for $\lambda^{cbv}_{s/r}$ must concern about *the type of a context* is that `shift` can change it. In the literature, the resulting type of the evaluation of $C[e]$ is called "answer type" so we call the type like that.

The *var* rule in the figure can be understood as follows:

> assume that there is type information $x : \tau$ in the type environment ρ. Then the variable x has a type τ and when x is evaluated in a context C whose type is $\tau \to \alpha$ (i.e., the answer type of the context is α), the resulting answer type of the expression $C[x]$ is α.

We note that the same type variable α appears in the both side of the judgement. This is because that a value must not change the answer type when it is evaluated in arbitrary contexts.

To understand the *fun* rule, we have to consider the answer types of the function body. Since some expression may change the answer type, a function may change the answer type when it is applied to an argument. Thus, we can understand the typing rule for a function expression as follows:

> assume that when a type information $x : \sigma$ is in the type environment ρ, and then an expression e has a type τ. And the expression e changes the answer type from α to β.
>
> Then the function $\lambda x.e$ has a type $(\sigma/\alpha \to \tau/\beta)$, and the answer type of the function expression must not be changed (i.e., both are δ).

As we explained above, the answer type must not change after the evaluation of a value.

A type of a function-application expression can be read as above. If we ignore the answer type, we can read this typing rule is as usual function-application rule, and the answer types in the rule mean that

(1) if the evaluation of e_1 changes the answer types, the resulting answer type (β) will be the final answer type of the application expression,

(2) if the evaluation of e_1 does not change the answer type and the evaluation of e_2 changes it, the resulting answer type (δ) will be the final answer type of the application expression, and

(3) if both the evaluation of e_1 and e_2 do not change the answer type, the answer type of the function body (ϵ) will be the final answer type of the application expression.

We note that these typing rules can be derived from the definition of CPS translations. For example, when the variable x has a type τ, the CPS translated expression $[\![x]\!]$ has a type $(\tau \to \alpha) \to \alpha$ in which α represents an arbitrary type, and $\tau \to \alpha$ is a type of the continuation. The type α just corresponds to the answer type, namely, a type of the returning value of a context. By CPS translating an expression, we can find the continuation, and the continuation is just the context: a type of a continuation corresponds to that of a context.

Chapter 3

Equational Axiomatization of Call-by-Name Delimited Control

We will explain our study about a Call-by-Name calculus containing `shift`/`reset` [27] in this chapter.

3.1 Introduction

Delimited continuation (or delimited control) has been proved useful in many applications from partial evaluation and Continuation Passing Style (CPS) transformation to representation of arbitrary monads to mobile computation. The traditional, unlimited continuation represents the whole rest of the computation (as the object captured by Scheme's `call/cc`), but the delimited continuation represents part of the rest of the computation.

While many works in the literature studied delimited control in call-by-value [9, 12], several recent work studied it in call-by-name. Herbelin and Ghilezan [23] related the study on classical logic with the study on delimited continuations, and proposed a call-by-name calculus for it. Kiselyov [30] proposed a call-by-name calculus with delimited-control operators and used it in linguistic analysis. Biernacka and Biernacki [6] introduced a type system for call-by-name delimited-control, which is similar to the type system for call-by-value delimited-control by Danvy and Filinski [8], and proved its strong normalization property.

In this chapter, we investigate the call-by-name calculi with delimited-control operators. Specifically, we identify the semantics of such a calculus through a CPS translation, and give a set of simple equations in direct style, which is sound and complete with respect to this semantics.

We think such an axiomatization is useful and worth studying. Given a set of reductions, one can compute every program (closed term), however, one cannot optimize open terms using reductions only. For instance, in a call-by-value calculus, $(\lambda x.x)\ (y\ z)$ is equal to $y\ z$ in any contexts, but $(\lambda x.x)\ (y\ z)$ cannot be reduced using the call-by-value reductions. On the contrary, they are equal after CPS translation. Appel [1] demonstrated a way to compiler construction through CPS translation, and Flanagan et al. [18] showed that, all possible optimizations after CPS-translating source terms may be done before CPS-translating them, if we adopt a sufficiently strong set of equations as axioms. The axioms must be sound in the sense that all reductions are respected (if e_1 reduces to e_2, they must be equal by the axioms), and must be complete in the sense that all possible optimizations for CPS terms can be done for direct-style terms.

In this chapter, we choose the control operators `shift`and `reset`among many delimited-control operators proposed in the literature. One of the most important merits of them is that they have a simple, functional CPS translation. Kameyama and Hasegawa [26] axiomatized the call-by-value calculus with shift and reset, namely, they identified the equational theory which coincides the CPS semantics for shift and reset. The axioms are simple enough to be used as optimizations, and can be used to reason about programs with shift and reset without converting the programs into CPS. In this paper we will carry out the same program for the call-by-name calculi.

Unlike the call-by-value calculi with delimited-control operators, there are a few choices about the semantics of the call-by-name calculi:

- Whether the calculus admits η-equality or not: full η-equality is usually assumed for call-by-name calculi, but it turns every term into a function (a value), and, therefore, interferes with the control operator reset (reset for a value

$e ::= c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathcal{S}k.e \mid k \hookleftarrow e \mid \langle e \rangle$	term
$E ::= \Box \mid Ee \mid \langle E \rangle \mid k \hookleftarrow E$	ev. ctxt.
$F ::= \Box \mid Fe$	pure ev. ctxt.

Figure 3.1: Syntax of the Source Language

does nothing). Hence, we need to restrict either of the two (at least), and we decided to abandon η-equality, since our intended operational semantics does not admit it.

In Section 3.8, we will mention yet another calculus which admits full η-equality but the use of reset is restricted.

- The semantics of the target calculus after the CPS translation: the standard CPS translation for delimited-control operators translates source terms into non-CPS terms, namely, arguments of function applications are not necessarily values, and therefore, their meaning depends on the semantics of the target calculus.

  We choose the call-by-value semantics for the target calculus, because it matches the intended operational behavior of control operators.

  To avoid the complicated equality theory having both call-by-name $\beta\eta$-equality and call-by-value $\beta\eta$-equality, we will translate the target terms once more by a CPS translation, following Danvy and Filinski [8].

It should be noted that, a naive adaptation of Kameyama and Hasegawa's completeness proof for the call-by-value calculus did not work for the call-by-name calculus. To overcome the difficulty, we need to refine the source calculus (before CPS translation) as well as the target calculus (after CPS translation) precisely. The key observations are that (1) the delimited continuations are linear in the call-by-name setting, and (2) we need to distinguish shift-bound variables from the ordinary, lambda-bound variables. As for (1), we note that the relationship between linearity and CPS translations was studied by Filinski as "linear continuation" [14], and by Berdine et al. as "linearly used continuations" [5], and, in this paper we need *both* kinds of linearity. As for (2), we adopt the formulation by Biernacka and Biernacki which has a special construct for the application of shift-bound variables to terms.

In this chapter, we give sound and complete axiomatization for the type-free and typed call-by-name calculi with delimited-control operators, and their relationship with axiomatization for the call-by-value calculus.

The rest of this chapter is organized as follows: in Section 3.2 we introduce the call-by-name calculus with delimited-control operators shift and reset. In Section 3.3 we give a CPS translation by Biernacka and Biernacki, and then a sound and complete axiomatization for the semantics in Section 3.4. The completeness proof is given in Section 3.5, and Section 3.6 discusses these results in typed setting. Section 3.7 discusses the use of linearity in our proofs, and Section 3.8 briefly mentions yet another CPS translation which respects η-equality. Section 3.9 gives concluding remarks.

3.2 Type-free Calculus

The calculus we study in this chapter is $\lambda_{s/r}$, a call-by-name lambda calculus with delimited-control operators `shift`and `reset`. The call-by-value version of these control operators was proposed by Danvy and Filinski [9, 10], and has been used to represent backtracking and various search, let-insertion in partial evaluation, various monads, type safe direct-style implementation of "printf", and others.

This section gives a type-free formulation of $\lambda_{s/r}$. Its type system will be explained later.

3.2.1 Syntax

In this subsection we give the syntax of terms and related expressions. First, we assume that there are two disjoint sets of variables, one for ordinary variables (denoted by $x, y, z, \cdots$) and the other for shift-bound variables (denoted by k). x is (eventually) bound by lambda and a term is substituted for x, while k is (eventually) bound by shift, and a delimited context (delimited continuation) is substituted for k.

Fig. 3.1 gives the syntax of $\lambda_{s/r}$. c is a constant. When we consider a typed calculus, we restrict c be a constant of basic types such as integer. An ordinary variable x is a term, while a shift-bound variable itself is not a term. The term $\lambda x.e$ and $e_1 e_2$ are abstraction and application as usual. The term $\mathcal{S}k.e$ is a shift-term, in which k is a bound variable. The

$$
\begin{array}{lll}
(\lambda x.e_1)\ e_2 \rightsquigarrow e_1\{x := e_2\} & & \\
\langle F[\mathcal{S}k.e]\rangle \rightsquigarrow \langle e\{k \Leftarrow F\}\rangle & & \\
\langle v\rangle \rightsquigarrow v & & \text{for } v = c,\ \lambda x.e
\end{array}
$$

Figure 3.2: Reduction Rules

term $k \hookleftarrow e$ is a throw-term that applies the (delimited) continuation k to a term e. The variable k in $k \hookleftarrow e$ is free. The term $\langle e\rangle$ is a reset-term.

Although the distinction between ordinary (lambda-bound) variables and continuation variables appears in the work of Parigot [42], it was Biernacka and Biernacki [6] who have made a clear distinction between lambda-bound variables and shift-bound variables in the context of call-by-name delimited control. They have also introduced the notation $k \hookleftarrow e$.

We identify α-equivalent terms. $\mathtt{FV}(e)$ denotes the set of free (ordinary and shift-bound) variables in e, and $e_1\{x := e_2\}$ represents the result of capture-avoiding substitution of e_2 for x in e_1. E represents an arbitrary evaluation context in call-by-name. F is a pure evaluation context, or a delimited evaluation context, which does not have resets around the hole $\Box$. In other words, a pure evaluation context is delimited by a reset, and is exactly a delimited continuation. A general evaluation context is a continuation outside of a reset, namely, a metacontinuation. $E[e]$ (or $F[e]$) denotes the term after the hole-filling operation of a term e for a hole in E (or F). Hole-filling of another evaluation context $E_1[E_2]$ is defined similarly.

Remarks. Let us briefly argue the design of the source calculus here, namely, if the distinction between shift-bound and lambda-bound variables is necessary or not.

If we are only concerned with the operational behavior of closed terms, there is no need to distinguish them. In this case, the expression $k \hookleftarrow e$ can be represented by $(k\ e)$.

However, this distinction is necessary for this chapter, since we are concerned with equality theories for open terms: to obtain complete axiomatization, we need an axiom which is valid for shift-bound variables but not valid for lambda-bound variables.

We remark that Biernacka and Biernacki introduced this distinction for a different purpose: to develop a type system for the calculus, which represents the so called answer-type polymorphism without making use of polymorphism [4]. The details may be found in [6].

In summary, two distinct classes of variables are necessary in the call-by-name calculi for delimited continuations. This is a sharp contrast with the call-by-value case, for which the above-mentioned distinction is not necessary [26].

3.2.2 Reduction Rules

To understand the operational behavior of shift and reset, we give the reduction rules of $\lambda_{s/r}$ in Fig. 3.2.

The reduction rules are essentially the same as those in Biernacka and Biernacki, but we have slightly changed them in the following point: their calculus has an expression $F \hookleftarrow e$ for a pure evaluation context F and an expression e. It is the result of substituting F for k in $k \hookleftarrow e$ (here we assume that k does not appear in e freely). On the other hand, our calculus does not have such an expression, and such a substitution is realized by a meta-operation $\{k \Leftarrow F\}$.

The substitution $\{k \Leftarrow F\}$ is defined as follows:

$$
\begin{aligned}
c\{k \Leftarrow F\} &\stackrel{\text{def}}{=} c \\
x\{k \Leftarrow F\} &\stackrel{\text{def}}{=} x \\
(\lambda x.e)\{k \Leftarrow F\} &\stackrel{\text{def}}{=} \lambda x.(e\{k \Leftarrow F\}) \\
&\qquad \text{where } x \notin \mathtt{FV}(F) \\
(e_1\ e_2)\{k \Leftarrow F\} &\stackrel{\text{def}}{=} (e_1\{k \Leftarrow F\})\ (e_2\{k \Leftarrow F\}) \\
(\mathcal{S}k'.e)\{k \Leftarrow F\} &\stackrel{\text{def}}{=} \mathcal{S}k'.(e\{k \Leftarrow F\}) \\
&\qquad \text{where } k' \notin \{k\} \cup \mathtt{FV}(F) \\
(k \hookleftarrow e)\{k \Leftarrow F\} &\stackrel{\text{def}}{=} \langle F[e\{k \Leftarrow F\}]\rangle
\end{aligned}
$$

$$(k' \hookleftarrow e)\{k \Leftarrow F\} \stackrel{\text{def}}{=} k' \hookleftarrow (e\{k \Leftarrow F\}) \quad \text{where } k' \notin \{k\} \cup \mathtt{FV}(F)$$

$$\langle e \rangle\{k \Leftarrow F\} \stackrel{\text{def}}{=} \langle e\{k \Leftarrow F\} \rangle$$

Let us look at the reduction rules in detail. The first reduction in Fig. 3.2 is the standard, call-by-name β-reduction.

In the second reduction rule, the evaluation context up to the nearest delimiter (reset) is F, and it is captured and substituted for k using the substitution $\{k \Leftarrow F\}$. The key case in this substitution is $(k \hookleftarrow e)\{k \Leftarrow F\}$, which is defined to be $\langle F[e\{k \Leftarrow F\}]\rangle$. This means that, by the substitution $\{k \Leftarrow F\}$, a functional object $\lambda x.\langle F[x]\rangle$ is substituted for k, if we identify $k \hookleftarrow e'$ with $(k\ e')$. Hence, the second reduction $\langle F[\mathcal{S}k.e]\rangle \rightsquigarrow \langle e\{k \Leftarrow F\}\rangle$ may be understood as

$$\langle F[\mathcal{S}k.e]\rangle \rightsquigarrow \langle e\{k := \lambda x.\langle F[x]\rangle\}\rangle$$

This is the same reduction rule as the one in the literature [26].

The third rule means: when a value is delimited by reset, then the delimiter is simply discarded. It may be strange to have the notion of values in the call-by-name calculus, but this rule naturally reflects the abstract-machine semantics for shift and reset in call-by-name[1]. Note that, a variable x is not a value in $\lambda_{s/r}$, so $\langle x\rangle$ does not reduce to x.

In Section 3.4, we will need to extend the above substitution to the form $\{k \Leftarrow (k' \hookleftarrow F)\}$, namely, we need to extend the substituted context F to the form $k' \hookleftarrow F$, which is not a pure evaluation context. The definition of the extended substitution remains the same except that F in the above definition may be in the form $k' \hookleftarrow F'$. For example, the key case is:

$$(k \hookleftarrow e)\{k \Leftarrow (k' \hookleftarrow F)\} \stackrel{\text{def}}{=} \langle k' \hookleftarrow (F[e\{k \Leftarrow (k' \hookleftarrow F)\}])\rangle$$

We remark that reset in the right-hand side of this definition is not necessary as $\langle k' \hookleftarrow e'\rangle = k' \hookleftarrow e'$ can be proved by our axioms introduced later.

3.3 Semantics based on CPS Translation

3.3.1 CPS Semantics

A CPS translation is a syntax-directed translation from a source calculus ($\lambda_{s/r}$ in this chapter) to a target calculus. It is a useful theoretical tool to give a precise semantics to a calculus with control operators, and may be used as an intermediate language for compilers, since we can perform various kinds of optimizations on the target terms of a CPS translation. Using the axioms in this chapter, one can perform the optimizations on the source terms.

The following three semantics are often studied for computational calculi:

- Reduction semantics: $e_1 = e_2$ if and only if they are equal up to the equality induced by the reduction rules.[2]
- CPS semantics. $e_1 = e_2$ if and only if they are translated to the same term by a CPS translation.
- Observational equivalence. $e_1 = e_2$ if and only if, for any context C such that $C[e_1]$ and $C[e_2]$ are closed, $C[e_1]$ and $C[e_2]$ both terminate, or both do not terminate under (a certain notion of) the operational semantics.

Under reasonable assumptions, the strength of the three semantics increases in the order above, namely, if two programs are equal in the reduction semantics (CPS semantics, resp.), they are equal in the CPS semantics (observational equivalence, resp.). The converse direction does not, in general, hold. For instance, $(\lambda x.xx)(\lambda x.xx)$ and $(\lambda x.xxx)(\lambda x.xxx)$ are equal under the observational equivalence, but they are not CPS-translated to equal terms for any standard CPS translations.

Although CPS semantics is weaker than observational equivalence, it is in many cases sufficient for reasoning about programs, as shown by literature (for instance, [1]). Indeed we often prefer CPS semantics than observational equivalence, since the latter is fragile in language extensions; under observational equivalence, two equal terms may not be equal after a new construct is added to the calculus, as we may be able to distinguish these terms using the new construct. On the contrary, CPS semantics is robust in the sense that, two equal terms remain equal even after a new construct is added, since their images of a CPS translation do not (usually) change.

[1] This semantics is attributed to Olivier Danvy in the literature [23].

[2] The equality is the reflexive, transitive, and congruent closure of the reduction rules.

$$
\begin{aligned}
[\![c]\!]_1 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa c \\
[\![x]\!]_1 &\stackrel{\text{def}}{=} \lambda\kappa.x\kappa \\
[\![\lambda x.e]\!]_1 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa(\lambda x.[\![e]\!]_1) \\
[\![e_1 e_2]\!]_1 &\stackrel{\text{def}}{=} \lambda\kappa.[\![e_1]\!]_1(\lambda m.m[\![e_2]\!]_1\kappa) \\
[\![\mathcal{S}k.e]\!]_1 &\stackrel{\text{def}}{=} \lambda\kappa.([\![e]\!]_1\{k := \kappa\})\theta_1 \\
[\![k \hookleftarrow e]\!]_1 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa([\![e]\!]_1 k) \\
[\![\langle e \rangle]\!]_1 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa([\![e]\!]_1\theta_1) \\
\theta_1 &\stackrel{\text{def}}{=} \lambda m.m
\end{aligned}
$$

Figure 3.3: 1CPS Translation

3.3.2 1CPS Translation

Our task is to precisely determine the CPS translation we use, and we choose a CPS translation based on Plotkin's one, as we will argue below.

For call-by-name calculi, there are two choices for CPS translations: Plotkin's [43] and Streicher and Reus' [50]. The latter respects full η-equality (η-equal terms are translated to equal terms), while the former does not. Using full η-equality, we can infer that the control operator `reset`becomes totally useless:

$$\langle e \rangle =_\eta \langle \lambda x.ex \rangle \rightsquigarrow \lambda x.ex =_\eta e$$

Without the reset operator, our calculus does not make sense. Consequently, we use a CPS translation based on Plotkin's. We will revisit this choice in Section 3.8.

Fig. 3.3 gives the CPS translation for the calculus $\lambda_{s/r}$ where we assume that the variables κ and m are fresh. The CPS translation is essentially due to Biernacka and Biernacki, which is based on Plotkin's call-by-name CPS translation, and is extended to the calculus with shift and reset. The translation $[\![\_]\!]_1$ maps a term in $\lambda_{s/r}$ to a term in the type-free lambda calculus. We call it 1CPS translation. For terms without control operators, this CPS translation is identical to Plotkin's one. Unlike the call-by-value one, a variable x is translated to $\lambda\kappa.x\kappa$. This reflects the fact that a variable in the call-by-name calculus $\lambda_{s/r}$ is not a value, but represents a computation, so its CPS counterpart should receive a continuation κ. We will assume that the target calculus of the CPS translation admits η-equality for the continuation variable κ, hence $[\![x]\!]_1$ could be defined as x.

The translation for a reset-term $\langle e \rangle$ is identical to that in the call-by-value setting [9]. It is also not difficult to understand the translation of a shift-term $\mathcal{S}k.e$. It captures the current continuation κ, binds it to the shift-bound variable k, and then installs the identity continuation θ_1. The translation of a throw-term $k \hookleftarrow e$ is to install the continuation k, compute e, and then re-activates the continuation κ by applying κ to the result of e. The translation of a reset-term is to save the current (delimited) continuation κ, and install the identity continuation θ_1 (which corresponds to an empty evaluation context) as the current continuation.

3.3.3 Semantics of CPS terms

Our purpose is to axiomatize the CPS semantics, and in order to do so, we need to precisely define the semantics of the target calculus. In the presence of delimited-control operators in the source calculus $\lambda_{s/r}$, the image of 1CPS translation is not strictly in CPS. Namely, the arguments in function applications may not be values. For instance, the image of a reset-term contains a subterm $\kappa([\![e]\!]_1\theta_1)$, in which κ is applied to a non-value term $[\![e]\!]_1\theta_1$, thus the evaluation of this term is dependent on the evaluation strategy. If we evaluate it in call-by-value, $[\![e]\!]_1\theta_1$ is computed first, and κ is applied to its result. If we evaluate it in call-by-name, $[\![e]\!]_1\theta_1$ is not computed now, and κ is applied to the unevaluated subterm $[\![e]\!]_1\theta_1$.

As for this choice, we again follow Biernacka and Biernacki who have chosen the call-by-value semantics for the target calculus. Rather than introducing the call-by-value semantics into the target calculus directly (such as Moggi's computational lambda calculus [36]), we translate the CPS terms once again [9], by the call-by-value CPS translation.

$$
\begin{aligned}
[\![x]\!]_2 &\stackrel{\text{def}}{=} x \\
[\![c]\!]_2 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa c \\
[\![\lambda x.e]\!]_2 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa(\lambda x.[\![e]\!]_2) \\
[\![e_1\ e_2]\!]_2 &\stackrel{\text{def}}{=} \lambda\kappa.[\![e_1]\!]_2(\lambda m.m[\![e_2]\!]_2\kappa) \\
[\![\mathcal{S}k.e]\!]_2 &\stackrel{\text{def}}{=} \lambda\kappa.([\![e]\!]_2\{k := \kappa\})\theta_2 \\
[\![k \hookleftarrow e]\!]_2 &\stackrel{\text{def}}{=} \lambda\kappa.\lambda\gamma.[\![e]\!]_2 k(\kappa \oplus \gamma) \\
[\![\langle e \rangle]\!]_2 &\stackrel{\text{def}}{=} \lambda\kappa.\lambda\gamma.[\![e]\!]_2\theta_2(\kappa \oplus \gamma) \\
\theta_2 &\stackrel{\text{def}}{=} \lambda m.\lambda\gamma.\gamma m \\
(\kappa \oplus \gamma) &\stackrel{\text{def}}{=} \lambda m.\kappa m\gamma
\end{aligned}
$$

Figure 3.4: 2CPS Translation

The successive CPS translations may be represented by a single CPS translation, which we call a 2CPS translation. The images of the 2CPS translation are terms in CPS, and we no longer have to be worried about different semantics.

3.3.4 2CPS Translation

Fig. 3.4 defines the 2CPS translation for $\lambda_{s/r}$, which introduces two continuation variables κ and γ. These variables as well as m are assumed to be fresh. If we decompose the 2CPS translation into two CPS translations, κ is introduced by the first (call-by-name) CPS translation, and γ is introduced by the second (call-by-value) CPS translation.

The definition of the 2CPS translation looks like that of the 1CPS translation. The trick is that we η-reduce the results of 2CPS translation whenever possible. For instance, $[\![x]\!]_2$ should be $\lambda\kappa.\lambda\gamma.x\kappa\gamma$, however, since we will introduce η-equality for the variable γ, it is equal to $\lambda\kappa.x\kappa$, so we define it as $[\![x]\!]_2 \stackrel{\text{def}}{=} \lambda\kappa.x\kappa$ (which is in turn equal to x by η-reduction for κ). Since the images of the 1CPS translation of the throw-term $k \hookleftarrow e$ and the reset-term $\langle e \rangle$ contain non-CPS forms, their images of the 2CPS translations must use the second continuation variable γ.

We also note that the identity continuation θ_1 ($= \lambda m.m$) becomes θ_2 ($= \lambda m.\lambda\gamma.\gamma m$), and that $(\kappa \oplus \gamma)$ (which is simply an abbreviation for $\lambda m.\kappa m\gamma$) corresponds to the push-operation for the stack of meta-continuations in the abstract machine semantics. (See [6] for details.)

In the following, we simply write $[\![e]\!]$ for $[\![e]\!]_2$, since we do not use the 1CPS translation.

3.3.5 Analysis of Target Calculus

In order to define the CPS semantics, we shall precisely define the semantics of the target calculus of the 2CPS translation.

By inspecting the images of the translation, we can easily show that terms in the target calculus are generated by the following grammar:

$$
\begin{aligned}
T &::= x \mid \lambda\kappa.P \mid VT & V &::= c \mid m \mid \lambda x.T \\
P &::= KV \mid TK \mid \lambda\gamma.A & K &::= \kappa \mid \lambda m.P \\
A &::= GV \mid PG & G &::= \gamma \mid \lambda m.A
\end{aligned}
$$

The target calculus has six sorts: T is the image of expressions in $\lambda_{s/r}$. P is a pre-term and A is an answer. V is the image of values (constant or abstraction). K is the image of pure evaluation contexts (delimited context). G is the image of (general) evaluation contexts. Variables belong to some specific sorts: x is a variable for T, m for V, κ for K, and γ for G.

It is important to note that $\beta\eta$-reductions in the target calculus preserve the sorts, which is easily proved.

For a term in the target calculus t_1 and t_2, we write $\vdash_{\text{CPS}} t_1 = t_2$ if they are proved equal using the following equations and the standard equality rules (reflexivity, symmetry, transitivity, and substitution):

- β-equality for x: $(\lambda x.T_1)\ T_2 = T_1\{x := T_2\}$.
- $\beta\eta$-equality for m: $(\lambda m.P)V = P\{m := V\}$, $\lambda m.Km = K$ (where $m \notin \mathtt{FV}(K)$), $(\lambda m.A)V = A\{m := V\}$, and $\lambda m.Gm = G$ (where $m \notin \mathtt{FV}(G)$).
- $\beta\eta$-equality for κ: $(\lambda\kappa.P)K = P\{\kappa := K\}$ and $\lambda\kappa.T\kappa = T$ (where $\kappa \notin \mathtt{FV}(T)$).
- $\beta\eta$-equality for γ: $(\lambda\gamma.A)G = A\{\gamma := G\}$ and $\lambda\gamma.P\gamma = P$ (where $\gamma \notin \mathtt{FV}(P)$).

Note that we have excluded η-equality for x, namely, $\lambda x.Vx = V$ is not allowed in general. This choice reflects the fact that the source calculus $\lambda_{s/r}$ does not allow η-equality as we argued before.

3.3.6 Properties of 2CPS Translation

In this subsection we show several useful properties of the 2CPS translation.

We first define $|F|$ for a pure evaluation context F. The purpose of $|\_|$ is to CPS-translate a pure evaluation context F, and it is defined as follows:

$$|\Box| \stackrel{\text{def}}{=} \lambda\kappa.\kappa$$
$$|F\ e| \stackrel{\text{def}}{=} \lambda\kappa.|F|\ (\lambda m.m[\![e]\!]\kappa)$$

Now we can state the following lemma which is useful in the soundness proof.

Lemma 3.1. *We have the following properties for the 2CPS translation.*

(1) $\vdash_{CPS}\ [\![F[e]]\!] = \lambda\kappa.[\![e]\!](|F|\kappa)$ *where* $\kappa \notin \mathit{FV}(e)$.

(2) $\vdash_{CPS}\ [\![e\{x := e'\}]\!] = [\![e]\!]\{x := [\![e']\!]\}$.

(3) $\vdash_{CPS}\ [\![e\{k \Leftarrow F\}]\!] = [\![e]\!]\{k := |F|\ \theta_2\}$.

(4) $\vdash_{CPS}\ [\![e\{k \Leftarrow (k' \hookleftarrow F)\}]\!] = [\![e]\!]\{k := |F|\ k'\}$.

Proof. (1) The equation is proved by induction on F. When $F = \Box$, it is trivial. When $F = F_1\ e_1$, we can prove it as follows:

$$\begin{aligned}
[\![F[e]]\!] &\equiv [\![F_1[e]\ e_1]\!] \\
&\equiv \lambda\kappa.[\![F_1[e]]\!]\ (\lambda m.m[\![e_1]\!]\kappa) \\
&= \lambda\kappa.(\lambda\kappa'.[\![e]\!](|F_1|\kappa'))\ (\lambda m.m[\![e_1]\!]\kappa) \ \text{ by I.H.} \\
&= \lambda\kappa.[\![e]\!](|F_1|(\lambda m.m[\![e_1]\!]\kappa)) \\
&= \lambda\kappa.[\![e]\!](|F_1 e_1|\ \kappa)
\end{aligned}$$

From the second line to the third, we used induction hypothesis on F_1.

(2) The equation is easily proved by induction on e.

(3) The equation is proved by induction on e. The only interesting case is $e \equiv k \hookleftarrow e_1$, and we can prove the case as follows:

$$\begin{aligned}
&[\![(k \hookleftarrow e_1)\{k \Leftarrow F\}]\!] \\
&\equiv [\![\langle F[e_1\{k \Leftarrow F\}]\rangle]\!] \\
&\equiv \lambda\kappa.\lambda\gamma.[\![F[e_1\{k \Leftarrow F\}]]\!]\theta_2(\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![e_1\{k \Leftarrow F\}]\!](|F|\ \theta_2)(\kappa \oplus \gamma) \ \text{ by (1)} \\
&= \lambda\kappa.\lambda\gamma.[\![e_1]\!]\{k := |F|\ \theta_2\}(|F|\ \theta_2)(\kappa \oplus \gamma) \ \text{ by I.H.} \\
&= (\lambda\kappa.\lambda\gamma.[\![e_1]\!]k(\kappa \oplus \gamma))\{k := |F|\ \theta_2\} \\
&\equiv [\![(k \hookleftarrow e_1)]\!]\{k := |F|\ \theta_2\}
\end{aligned}$$

$$
\begin{array}{rll}
(\lambda x.e_1)\ e_2 = e_1\{x := e_2\} & & \beta \\
\langle F[\mathcal{S}k.e]\rangle = \langle e\{k \Leftarrow F\}\rangle & & \text{reset-shift} \\
k' \hookleftarrow (F[\mathcal{S}k.e]) = \langle e\{k \Leftarrow (k' \hookleftarrow F)\}\rangle & & \text{throw-shift} \\
\langle v\rangle = v \quad \text{for } v = c,\ \lambda x.e & & \text{reset-value} \\
\mathcal{S}k.(k \hookleftarrow e) = e \quad \text{where } k \notin \mathtt{FV}(e) & & \text{shift-elim} \\
\mathcal{S}k.\langle e\rangle = \mathcal{S}k.e & & \text{shift-reset}
\end{array}
$$

Figure 3.5: Axioms for $\lambda_{s/r}$ (call-by-name)

(4) This case is proved in the same way as (3). Here we show the case $e \equiv k \hookleftarrow e_1$.

$$
\begin{aligned}
&[\![(k \hookleftarrow e_1)\{k \Leftarrow (k' \hookleftarrow F)\}]\!] \\
&\equiv [\![k' \hookleftarrow F[e_1\{k \Leftarrow (k' \hookleftarrow F)\}]]\!] \\
&\equiv \lambda\kappa.\lambda\gamma.[\![F[e_1\{k \Leftarrow (k' \hookleftarrow F)\}]]\!]k'(\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![e_1\{k \Leftarrow (k' \hookleftarrow F)\}]\!](|F|\ k')(\kappa \oplus \gamma) \ \text{ by (1)} \\
&= \lambda\kappa.\lambda\gamma.[\![e_1]\!]\{k := |F|\ k'\}(|F|\ k')(\kappa \oplus \gamma) \ \text{ by I.H.} \\
&= (\lambda\kappa.\lambda\gamma.[\![e_1]\!]k(\kappa \oplus \gamma))\{k := |F|\ k'\} \\
&\equiv [\![k \hookleftarrow e_1]\!]\{k := |F|\ k'\}
\end{aligned}
$$

□

3.4 Axiomatizing the Semantics

In this section, we provide a set of axioms for $\lambda_{s/r}$, and show that it is sound with respect to the CPS semantics. Its completeness will be proved in the next section.

3.4.1 Axioms

Fig. 3.5 gives our axioms for $\lambda_{s/r}$. We write $\vdash^{\text{cbn}}_{\text{s/r}}\ e_1 = e_2$ if we can prove the equation using these axioms and the standard inference rules for equality.

For the purpose of comparison, Fig. 3.6 gives the axioms for the call-by-value calculus $\lambda^{cbv}_{s/r}$ by Kameyama and Hasegawa [26]. Although their calculus does not have the throw expression $k \hookleftarrow e$, we introduce it for the purpose of comparison. Their axioms are slightly modified accordingly. Note that the definitions for a value v and the pure evaluation context F for the call-by-value calculus are different from those for $\lambda_{s/r}$.

We will explain the axioms in Fig. 3.5 by comparing them to those in Fig. 3.6.

- The call-by-name calculus $\lambda_{s/r}$ has a single strong axiom for β, while the call-by-value one $\lambda^{cbv}_{s/r}$ has three weaker axioms β_v, β_Ω, and reset-lift. The former subsumes the latter.
- No axiom for η-equality exists in $\lambda_{s/r}$, while $\lambda^{cbv}_{s/r}$ has the call-by-value version of η-equality as η_v.
- The axioms shift-elim and shift-reset are contained in both calculi.
- Two axioms reset-shift and reset-value are contained in both calculi, but their actual meanings differ between the calculi, since the definitions of F and v differ.
- The axiom throw-shift is contained in both calculi.

  For the call-by-value calculus, this is only because $k \hookleftarrow e$ is a special form. If we do not distinguish shift-bound and lambda-bound variables, then $k \hookleftarrow e$ may be represented by $(k\ e)$, and the axiom throw-shift is subsumed by reset-shift (note that $(k\ F)$ is an evaluation context in $\lambda^{cbv}_{s/r}$ if k is a variable).

$$
\begin{array}{rll}
(\lambda x.e_1)\ v = e_1\{x := v\} & & \beta_v \\
\lambda x.v\ x = v \ \text{ where } x \notin \mathtt{FV}(v) & & \eta_v \\
(\lambda x.F[x])e = F[e] \ \text{ where } x \notin \mathtt{FV}(F) & & \beta_\Omega \\
\langle F[\mathcal{S}k.e]\rangle = \langle e\{k \Leftarrow F\}\rangle & & \text{reset-shift} \\
k' \hookleftarrow (F[\mathcal{S}k.e]) = \langle e\{k \Leftarrow (k' \hookleftarrow F)\}\rangle & & \text{throw-shift} \\
\langle v\rangle = v & & \text{reset-value} \\
\mathcal{S}k.(k \hookleftarrow e) = e \ \text{ where } k \notin \mathtt{FV}(e) & & \text{shift-elim} \\
\mathcal{S}k.\langle e\rangle = \mathcal{S}k.e & & \text{shift-reset} \\
\langle(\lambda x.e_1)\langle e_2\rangle\rangle = (\lambda x.\langle e_1\rangle)\langle e_2\rangle & & \text{reset-lift}
\end{array}
$$

where v and F are defined as follows:

$$
\begin{aligned}
v &::= c \mid x \mid \lambda x.e \\
F &::= \Box \mid F\ e \mid v\ F
\end{aligned}
$$

Figure 3.6: Axioms for $\lambda^{cbv}_{s/r}$ (call-by-value)

On the contrary, the axiom throw-shift is necessary for the call-by-name calculus, and it seems difficult to formulate this axiom without distinguishing shift-bound variables from lambda-bound variables. Without such a distinction, throw-shift would become unsound.

We believe that our axioms for the call-by-name calculus are simple enough to be used for reasoning about open programs, and that the similarity of the axioms for the two calculi is an evidence that our axioms are natural and stable.

We emphasize that establishing this kind of axioms is not trivial even after getting to know the axiomatization for the call-by-value case, since the CPS translation of the shift term in call-by-name is not quite the same as that in call-by-value.

It is easy to prove that the reduction semantics is subsumed by our axioms.

Theorem 3.1. *Let e_1 and e_2 be expressions in $\lambda_{s/r}$. If $e_1 \rightsquigarrow e_2$ by one of the reduction rules in Fig. 3.2, then $\vdash^{cbn}_{s/r}\ e_1 = e_2$ is derivable.*

Proof. The three reductions, respectively, are subsumed by the axioms β, reset-shift, and reset-value, respectively. □

The rest of this chapter is devoted to establish that the axioms in Fig. 3.5 are sound and complete with respect to the CPS semantics induced by the 2CPS translation.

3.4.2 Soundness of Axioms

Soundness of the axioms means that equal terms in the source calculus $\lambda_{s/r}$ are CPS-translated to equal terms. Soundness may be thought as a sort of correctness for a CPS translation.

Theorem 3.2 (Soundness)**.** *Let e_1 and e_2 be terms in $\lambda_{s/r}$. If $\vdash^{cbn}_{s/r}\ e_1 = e_2$ is derivable, then $\vdash_{CPS}\ [\![e_1]\!] = [\![e_2]\!]$ is derivable.*

Proof. It suffices to prove the theorem when $e_1 = e_2$ is obtained by one of the axioms. We list the proofs of all the cases, since they are instructive to know the details of the CPS translation.

(β)

$$
\begin{aligned}
[\![(\lambda x.e_1)\ e_2]\!] &\equiv \lambda\kappa.[\![\lambda x.e_1]\!]\ (\lambda m.m\ [\![e_2]\!]\ \kappa) \\
&= \lambda\kappa.(\lambda\kappa'.\kappa'\ (\lambda x.[\![e_1]\!]))\ (\lambda m.m\ [\![e_2]\!]\ \kappa) \\
&= \lambda\kappa.(\lambda x.[\![e_1]\!])\ [\![e_2]\!]\ \kappa \\
&= \lambda\kappa.[\![e_1]\!]\{x := [\![e_2]\!]\}\ \kappa \\
&= [\![e_1]\!]\{x := [\![e_2]\!]\}
\end{aligned}
$$

$$= [\![e_1\{x := e_2\}]\!] \text{ by Lemma 3.1 (2)}$$

(reset-shift)

$$\begin{aligned}
&[\![\langle F[\mathcal{S}k.e]\rangle]\!] \\
&\equiv \lambda\kappa.\lambda\gamma.[\![F[\mathcal{S}k.e]]\!]\ \theta_2\ (\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![\mathcal{S}k.e]\!]\ (|F|\ \theta_2)\ (\kappa \oplus \gamma) \text{ by Lemma 3.1 (1)} \\
&\equiv \lambda\kappa.\lambda\gamma.((\lambda\kappa'.[\![e]\!]\{k := \kappa'\})\ \theta_2)(|F|\ \theta_2)\ (\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.([\![e]\!]\{k := |F|\ \theta_2\})\ \theta_2\ (\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![e\{k \Leftarrow F\}]\!]\ \theta_2\ (\kappa \oplus \gamma) \text{ by Lemma 3.1 (3)} \\
&\equiv [\![\langle e\{k \Leftarrow F\}\rangle]\!]
\end{aligned}$$

(throw-shift, $k' \neq k$)

$$\begin{aligned}
&[\![k' \hookleftarrow (F[\mathcal{S}k.e])]\!] \\
&\equiv \lambda\kappa.\lambda\gamma.[\![F[\mathcal{S}k.e]]\!]\ k'\ (\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![\mathcal{S}k.e]\!](|F|\ k')\ (\kappa \oplus \gamma) \text{ by Lemma 3.1 (1)} \\
&\equiv \lambda\kappa.\lambda\gamma.(\lambda\kappa'.[\![e]\!]\{k := \kappa'\}\theta_2)(|F|\ k')\ (\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.([\![e]\!]\{k := |F|\ k'\})\ \theta_2\ (\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![e\{k \Leftarrow (k' \hookleftarrow F)\}]\!]\ \theta_2\ (\kappa \oplus \gamma) \text{ by Lemma 3.1 (4)} \\
&\equiv [\![\langle e\{k \Leftarrow (k' \hookleftarrow F)\}\rangle]\!]
\end{aligned}$$

(reset-value) We define v^* for a value v by:

$$c^* \stackrel{\text{def}}{=} c$$
$$(\lambda x.e)^* \stackrel{\text{def}}{=} \lambda x.[\![e]\!]$$

Then we can prove:

$$\begin{aligned}
[\![\langle v\rangle]\!] &\equiv \lambda\kappa.\lambda\gamma.(\lambda\kappa'.\kappa' v^*)\ \theta_2\ (\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.\theta_2\ v^*\ (\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.(\kappa \oplus \gamma)\ v^* \\
&= \lambda\kappa.\lambda\gamma.\kappa\ v^*\ \gamma \\
&= \lambda\kappa.\kappa\ v^* \\
&\equiv [\![v]\!]
\end{aligned}$$

(shift-elim, $k \notin \mathtt{FV}(e)$)

$$\begin{aligned}
[\![\mathcal{S}k.(k \hookleftarrow e)]\!] &\equiv \lambda\kappa.([\![k \hookleftarrow e]\!]\{k := \kappa\})\ \theta_2 \\
&= \lambda\kappa.(\lambda\kappa'.\lambda\gamma.[\![e]\!]\ k\ (\kappa' \oplus \gamma))\{k := \kappa\})\ \theta_2 \\
&= \lambda\kappa.\lambda\gamma.[\![e]\!]\ \kappa\ (\theta_2 \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![e]\!]\ \kappa\ \gamma \\
&= [\![e]\!]
\end{aligned}$$

(shift-reset)

$$\begin{aligned}
[\![\mathcal{S}k.\langle e\rangle]\!] &= \lambda\kappa.([\![\langle e\rangle]\!]\{k := \kappa\})\ \theta_2 \\
&= \lambda\kappa.(\lambda\kappa'.\lambda\gamma.[\![e]\!]\ \theta_2\ (\kappa' \oplus \gamma))\{k := \kappa\})\ \theta_2 \\
&= \lambda\kappa.\lambda\gamma.([\![e]\!]\ \theta_2\ (\theta_2 \oplus \gamma))\{k := \kappa\} \\
&= \lambda\kappa.\lambda\gamma.([\![e]\!]\ \theta_2\ \gamma)\{k := \kappa\} \\
&= [\![\mathcal{S}k.e]\!]
\end{aligned}$$

□

3.5 Completeness

Completeness is the converse of soundness: if the images of the CPS translation of two terms are equal in the target calculus, then they are equal in the source calculus. Completeness ensures that all equational reasoning for CPS terms can be done in the source calculus.

Theorem 3.3 (Completeness)**.** *Let e_1 and e_2 be (type-free) terms in $\lambda_{s/r}$. If $\vdash_{CPS} [\![e_1]\!] = [\![e_2]\!]$ is derivable, then $\vdash^{cbn}_{s/r} e_1 = e_2$ is derivable.*

Let us give an overview of the proof of completeness.
We will basically follow the general recipe by Sabry and Felleisen [45] as follows:

(1) Analyze the structure of target terms of CPS translation, and define a grammar for it, which should be closed under the reductions in the target calculus.

(2) Define a translation from the target to the source calculus, and prove that it is an inverse of the CPS translation.

(3) Prove that equality in the target calculus is preserved by the inverse translation.

However, we encountered a problem when the grammar in Section 3.3.5 is used for this purpose; we could not define a suitable image of the inverse function for the variable m (the variable for the V-sort) with the desired property. Let us explain this in more detail. In the completeness proof, we will need the property $\langle v^\circ \rangle = v^\circ$ for any term v of the V-sort where v° is the image of v by the inverse function. As a special case, we need $\langle m^\circ \rangle = m^\circ$. However, if m° is a variable in $\lambda_{s/r}$, this property does hold in general. (Note that a variable in $\lambda_{s/r}$ represents an arbitrary expression, and not necessarily a value, thus $\langle x \rangle = x$ does not hold in general.) In addition, no other choice for m° seems to exist as m is a variable.

We can overcome this difficulty by investigating the target calculus more carefully. The variable m in the target calculus is not used in an arbitrary way, but is used *linearly*. Linearity of the variable m reflects the fact that a delimited continuation (a pure evaluation context) in the call-by-name calculus is linear in its hole: $F ::= \Box \mid F\ e$. [3]

Since m is linear, we do not have to keep the name of m through the inverse function, and we may discard it. The same technique was used by Kameyama and Hasegawa's axiomatization of shift and reset in call-by-value, where the metacontinuation variable γ is linear, and thus can be discarded by the inverse function. Here, we discard not only the name of γ, but also the name of m.

In the following proof, we do not directly formulate linear lambda calculus as the target calculus. Instead, we refine the grammar to reflect the linearity of the variable m. Therefore we do not explicitly mention the linearity in the proof, but it is embedded in the grammar of the target calculus, and the definition of the inverse translation from the target calculus. The point here is that this simple trick is sufficient to avoid the problem above.

3.5.1 Refined Grammar and Inverse Function

We give a refined grammar for the target calculus as:

$T ::= x \mid \lambda\kappa.P \mid VT$	term
$P ::= KV \mid TK \mid \lambda\gamma.A$	preterm
$A ::= GV \mid PG$	answer
$V ::= c \mid \lambda x.T$	value
$K ::= \kappa \mid \lambda m.m\ T\ K \mid \lambda m.\lambda\gamma.\gamma\ m$	continuation
$G ::= \gamma \mid \lambda m.K\ m\ G$	metacontinuation

The difference from the previous grammar is that we no longer regard m as a member of the V-sort. Instead, the K- and G-sorts contain a few more terms using m, and m is a linear variable.

The equality of the target calculus remains the same, however, since η-redex for m does not exist in the terms generated by the new grammar, it is discarded. In summary, the equality of the terms generated by the new grammar is induced by β-equality for the variables x and m, and $\beta\eta$-equality for the variables κ and γ.

[3] This linearity has been already mentioned in the literature, for instance, in Herbelin and Ghilezan [23].

$$
\begin{array}{rlrl}
x^\circ &\stackrel{\text{def}}{=} x & (\lambda\kappa.P)^\circ &\stackrel{\text{def}}{=} \mathcal{S}\kappa.P^\circ \\
(VT)^\circ &\stackrel{\text{def}}{=} V^\circ\ T^\circ & & \\
(KV)^\circ &\stackrel{\text{def}}{=} K^\circ[V^\circ] & (TK)^\circ &\stackrel{\text{def}}{=} K^\circ[T^\circ] \\
(\lambda\gamma.A)^\circ &\stackrel{\text{def}}{=} A^\circ & & \\
(PG)^\circ &\stackrel{\text{def}}{=} G^\circ[\langle P^\circ\rangle] & (GV)^\circ &\stackrel{\text{def}}{=} G^\circ[V^\circ] \\
c^\circ &\stackrel{\text{def}}{=} c & (\lambda x.T)^\circ &\stackrel{\text{def}}{=} \lambda x.T^\circ \\
\kappa^\circ &\stackrel{\text{def}}{=} \kappa \leftarrow \Box & (\lambda m.m\ T\ K)^\circ &\stackrel{\text{def}}{=} K^\circ[\Box\ T^\circ] \\
& & (\lambda m.\lambda\gamma.\gamma\ m)^\circ &\stackrel{\text{def}}{=} \Box \\
\gamma^\circ &\stackrel{\text{def}}{=} \Box & (\lambda m.K\ m\ G)^\circ &\stackrel{\text{def}}{=} G^\circ[\langle K^\circ\Box\rangle]
\end{array}
$$

Figure 3.7: Inverse Function

In the following, we use T, P, A, V, K and G not only for the names of sorts, but also for meta-variables of the corresponding sorts, for instance, T is used as a meta-variable for terms of the T-sort.

We can easily prove the following lemma for the refined grammar.

Lemma 3.2.

- *If e is a $\lambda_{s/r}$-term, then $[\![e]\!]$ belongs to the T-sort.*
- *Each of the above sorts is closed under β-reduction for x and m, and $\beta\eta$-reduction for κ and γ.*

We then define an inverse function $(\_)^\circ$ from the target calculus to source calculus in Fig. 3.7. By the inverse function, a term in the T, P, A, or V-sort is mapped to a term in $\lambda_{s/r}$, a term in the K or G-sort to an evaluation context.

Note that reset is introduced in $(PV)^\circ$ and $(\lambda m.K\ m\ G)^\circ$, and the names of the variables m and γ are discarded through the inverse, since these variables are linear.

We prove that the above "inverse" function is actually a (left) inverse of the 2CPS translation.

Lemma 3.3 (Inverse)**.** *Let e be an expression in $\lambda_{s/r}$. We have $\vdash^{cbn}_{s/r}\ [\![e]\!]^\circ = e$.*

Proof. This lemma can be proved by a straightforward induction on e. Let us show a few interesting cases.

For $e = \langle e'\rangle$, we have:

$$
\begin{aligned}
[\![e]\!]^\circ &= (\lambda\kappa.\lambda\gamma.[\![e']\!]\theta_2(\kappa \oplus \gamma))^\circ \\
&= \mathcal{S}\kappa.(\lambda m.\kappa\ m\ \gamma)^\circ[\langle\theta_2{}^\circ[[\![e']\!]^\circ]\rangle] \\
&= \mathcal{S}\kappa.\kappa \leftarrow \langle[\![e']\!]^\circ\rangle = \langle[\![e']\!]^\circ\rangle = \langle e'\rangle
\end{aligned}
$$

For $e = \mathcal{S}k.e'$, we have:

$$
\begin{aligned}
[\![e]\!]^\circ &= (\lambda\kappa.([\![e']\!]\{k := \kappa\})\theta_2)^\circ \\
&= (\lambda k.[\![e']\!]\theta_2)^\circ \\
&= \mathcal{S}k.\theta_2{}^\circ[[\![e']\!]^\circ] = \mathcal{S}k.e'
\end{aligned}
$$

□

3.5.2 Properties of Inverse Function

We present several technical properties about substitution and the inverse.

Lemma 3.4. *Substitutions for x, κ, and γ commute with the inverse function in the following sense.*

(1) $\vdash^{cbn}_{s/r} (t\{x := T_1\})^\circ = t^\circ\{x := {T_1}^\circ\}$ *is derivable if* t *is a* T*-,* P*-,* A*-,* V*-,* K*-, or* G*-term.*

(2) *Let* ϕ *be* $\{\kappa := K_1\}$ *and* ψ *be* $\{\kappa \Leftarrow {K_1}^\circ\}$*. Then we have:*

- $\vdash^{cbn}_{s/r} (t\phi)^\circ = t^\circ\psi$ *if* t *is a* T*- or* V*-term.*
- $\vdash^{cbn}_{s/r} \langle(t\phi)^\circ\rangle = \langle t^\circ\psi\rangle$ *if* t *is a* P*- or* A*-term.*
- $\vdash^{cbn}_{s/r} \langle(K\phi)^\circ[e]\rangle = \langle(K^\circ\psi)[e]\rangle$ *for any term* e *in* $\lambda_{s/r}$.
- $\vdash^{cbn}_{s/r} (G\phi)^\circ[\langle e\rangle] = (G^\circ\psi)[\langle e\rangle]$ *for any term* e *in* $\lambda_{s/r}$.

(3) $\vdash^{cbn}_{s/r} \langle(A\{\gamma := G_1\})^\circ\rangle = \langle {G_1}^\circ[\langle A^\circ\rangle]\rangle$.

(4) $\vdash^{cbn}_{s/r} (G\{\gamma := G_1\})^\circ[\langle e\rangle] = {G_1}^\circ[\langle G^\circ[\langle e\rangle]\rangle]$ *for any term* e *in* $\lambda_{s/r}$.

Proof. This lemma can be proved by simultaneous induction on each term. Note that $(P\phi)^\circ = P^\circ\psi$ does not hold in general, but it suffices to prove $\langle(P\phi)^\circ\rangle = \langle P^\circ\psi\rangle$ to make the induction go through, since P always appears immediately under reset. Similarly for K, we only have to consider the form $\langle K^\circ[e]\rangle$. We also make use of the fact that γ does not appear free in T, P, V or K.

Here, we prove a few interesting cases.

(Case $K \equiv \kappa$ for $\langle(K\phi)^\circ[e]\rangle = \langle(K^\circ\psi)[e]\rangle$)

We may assume that $\kappa \notin \mathtt{FV}(e)$. The left-hand side is $\langle {K_1}^\circ[e]\rangle$, and the right-hand side is $\langle(\kappa^\circ\psi)[e]\rangle$, which is equal to $\langle(\kappa \hookleftarrow e)\{\kappa \Leftarrow {K_1}^\circ\}\rangle$. Then we have:

$$\vdash^{cbn}_{s/r} \langle(\kappa \hookleftarrow e)\{\kappa \Leftarrow {K_1}^\circ\}\rangle = \langle\langle {K_1}^\circ[e]\rangle\rangle = \langle {K_1}^\circ[e]\rangle$$

(Case $G \equiv \lambda m.KmG_2$ for (4))

The left-hand side of (4) is $(G_2\{\gamma := G_1\})^\circ[\langle K^\circ[\langle e\rangle]\rangle]$, which is equal to ${G_1}^\circ[\langle {G_2}^\circ[\langle K^\circ[\langle e\rangle]\rangle]\rangle]$ by induction hypothesis, which is equal to the right-hand side. □

Lemma 3.5 (Inverse preserves reduction)**.** *Suppose* $\vdash_{CPS} t_1 = t_2$ *is derivable for target terms* t_1 *and* t_2*, then we have:*

- $\vdash^{cbn}_{s/r} {t_1}^\circ = {t_2}^\circ$ *if* t_1 *and* t_2 *are* T*- or* V*-terms.*
- $\vdash^{cbn}_{s/r} \langle {t_1}^\circ\rangle = \langle {t_2}^\circ\rangle$ *if* t_1 *and* t_2 *are* P*- or* A*-terms.*
- $\vdash^{cbn}_{s/r} \langle {t_1}^\circ[e]\rangle = \langle {t_2}^\circ[e]\rangle$ *if* t_1 *and* t_2 *are* K*-terms and* e *is a term in* $\lambda_{s/r}$.
- $\vdash^{cbn}_{s/r} {t_1}^\circ[\langle e\rangle] = {t_2}^\circ[\langle e\rangle]$ *if* t_1 *and* t_2 *are* G*-terms and* e *is a term in* $\lambda_{s/r}$.

Proof. It suffices to prove the lemma when t_2 is obtained from t_1 by β or η reduction.

There are six β-redexes, and two η-redexes. For each reduction $t_1 \rightsquigarrow t_2$, it is not difficult to prove $\vdash^{cbn}_{s/r} {t_1}^\circ = {t_2}^\circ$.

The most interesting case is β-reduction for κ, namely, the case $\vdash^{cbn}_{s/r} \langle((\lambda\kappa.P)K)^\circ\rangle = \langle(P\{\kappa := K\})^\circ\rangle$. We have:

$$\langle((\lambda\kappa.P)K)^\circ\rangle = \langle K^\circ[\mathcal{S}\kappa.P^\circ]\rangle$$

and

$$\langle(P\{\kappa := K\})^\circ\rangle = \langle P^\circ\{\kappa \Leftarrow K^\circ\}\rangle$$

by Lemma 3.4. By inspecting the definition of K°, it is either a pure evaluation context, or the form $\kappa' \hookleftarrow F$ for some κ' and F. In the former case, we use the axiom reset-shift to prove:

$$\vdash^{cbn}_{s/r} \langle K^\circ[\mathcal{S}\kappa.P^\circ]\rangle = \langle P^\circ\{\kappa \Leftarrow K^\circ\}\rangle$$

while in the latter case, we use the axiom throw-shift to prove the same equation. Hence we are done.

The other cases can be proved easily. □

It should be emphasized that the proof of this theorem actually needs the axiom throw-shift.

Therem 3.3. Let e and e' be terms in $\lambda_{s/r}$. Suppose $\vdash_{\text{CPS}} [\![e]\!] = [\![e']\!]$ is derivable. Since $[\![e]\!]$ and $[\![e']\!]$ are T-terms, we have $\vdash^{\text{cbn}}_{\text{s/r}} [\![e]\!]^\circ = [\![e']\!]^\circ$ by Lemma 3.5, and then $\vdash^{\text{cbn}}_{\text{s/r}} e = e'$ by Lemma 3.3. □

$$\frac{}{\Gamma \cup \{x : (\alpha \mid \sigma \mid \beta)\} \mid \alpha \vdash x : \sigma \mid \beta} \text{ var}$$

$$\frac{(c \text{ is a constant of base type } b)}{\Gamma \mid \alpha \vdash c : b \mid \alpha} \text{ const}$$

$$\frac{\Gamma, x : (\alpha \mid \sigma \mid \beta) \mid \alpha' \vdash e : \sigma' \mid \beta'}{\Gamma \mid \gamma \vdash \lambda x.e : (\alpha \mid \sigma \mid \beta) \to (\alpha' \mid \sigma' \mid \beta') \mid \gamma} \text{ fun}$$

$$\frac{\Gamma \mid \beta' \vdash e_0 : (\alpha \mid \sigma \mid \beta) \to (\alpha' \mid \sigma' \mid \beta') \mid \gamma \quad \Gamma \mid \alpha \vdash e_1 : \sigma \mid \beta}{\Gamma \mid \alpha' \vdash e_0 e_1 : \sigma' \mid \gamma} \text{ app}$$

$$\frac{\Gamma \mid \alpha \vdash e : \alpha \mid \beta}{\Gamma \mid \gamma \vdash \langle e \rangle : \beta \mid \gamma} \text{ reset}$$

$$\frac{\Gamma \cup \{k : \sigma \rhd \tau\} \mid \alpha \vdash e : \alpha \mid \beta}{\Gamma \mid \tau \vdash \mathcal{S}k.e : \sigma \mid \beta} \text{ shift}$$

$$\frac{\Gamma \mid \tau \vdash e : \sigma \mid \beta}{\Gamma \cup \{k : \sigma \rhd \tau\} \mid \alpha \vdash k \hookleftarrow e : \beta \mid \alpha} \text{ throw}$$

Figure 3.8: Type System

3.6 Typed Calculus

We have so far studied the type-free calculus only, and a natural question is whether our axioms work for typed calculus. In this section we give a positive answer to it.

3.6.1 Type System

We define a type system for our call-by-name calculus with delimited control. The type system defined in this section is essentially the same as the type system by Biernacka and Biernacki modulo the following modifications.

- Our syntax does not have an expression $F \hookleftarrow e$ for a pure evaluation context F and an expression e, so the corresponding typing rule is not contained.
- As a consequence, we do not need typing rules for delimited contexts and metacontexts, so they are omitted.
- We use a slightly different notation for function types. This is only notational difference: we write $(\alpha \mid \sigma \mid \beta) \to (\alpha' \mid \sigma' \mid \beta')$ for their type $\sigma^{\alpha,\beta}{}_{\alpha'} \to_{\beta'} \sigma'$.

A type (denoted by $\alpha, \beta, \sigma, \tau, \cdots$) is either a basic type b (such as integer and boolean), or a function type $(\alpha \mid \tau \mid \beta) \to (\alpha' \mid \tau' \mid \beta')$. The function type may be understood as the type of a function whose argument has the type τ with an effect of type α to β, and whose return type is τ' with an effect of type α' to β'. Here, "an effect of type α to β" is a computational effect invoked by a shift-expression, which changes the answer type of the current continuation from α to β. See [4] for the detailed discussion on the answer-type modification. If there is no effect by a shift-expression, then α and β are the same type (which can be an arbitrary type [57].)

We also define a context-type $\sigma \rhd \tau$ where σ and τ are types. A context-type is used for typing a delimited continuation, or a pure evaluation context.

A typing context Γ is a (possibly empty) set consisting of $x : (\alpha \mid \tau \mid \beta)$ and $k : \sigma \rhd \tau$. The former intuitively means that an ordinary variable x has type τ with an effect of type α to β, while the latter represents the type of a delimited continuation corresponding to k. Since a delimited continuation is always a pure function, we do not have to consider its effects, so the type of k needs only two subtypes. A judgement takes the form $\Gamma \mid \alpha \vdash e : \tau \mid \beta$ where Γ is a typing context, α, τ, β are types, and e is an expression.

Fig. 3.8 gives the type system.

As Biernacka and Biernacki proved, the reduction rules in Fig. 3.2 enjoy the subject reduction property under this type system.

3.6.2 Typed CPS Translation

For the purpose of this chapter, namely, axiomatization, we define a CPS translation for the typed calculus. Not surprisingly, we use the same CPS translation, the 2CPS translation in Fig. 3.4 for expressions in the typed calculus. The remaining task is to define the CPS translation of types and typing contexts.

For a simple type[4] A, we define $\neg A$ as $A \to \bot$ where $\bot$ is an arbitrary fixed type.

The CPS translation of a type is defined by $b^* \stackrel{\text{def}}{=} b$ and

$$\begin{aligned}&((\alpha \mid \tau \mid \beta) \to (\alpha' \mid \tau' \mid \beta'))^* \\ &\stackrel{\text{def}}{=} ((\tau^* \to \neg\neg\alpha^*) \to \neg\neg\beta^*) \to ((\tau'^* \to \neg\neg\alpha'^*) \to \neg\neg\beta'^*)\end{aligned}$$

The definition above needs several occurrences of double negation $\neg\neg\_$ since we use 2CPS translation.

Finally, we define the CPS translation of a typing context as:

$$(x : (\alpha \mid \tau \mid \beta))^* \stackrel{\text{def}}{=} x : (\tau^* \to \neg\neg\alpha^*) \to \neg\neg\beta^*$$

$$(k : \sigma \rhd \tau)^* \stackrel{\text{def}}{=} k : \sigma^* \to \neg\neg\tau^*$$

Theorem 3.4. *If $\Gamma \mid \alpha \vdash e : \tau \mid \beta$ is derivable, then $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \neg\neg\alpha^*) \to \neg\neg\beta^*$ is derivable in the simply typed lambda calculus.*

Proof. The theorem can be proved by induction on the derivation of $\Gamma \mid \alpha \vdash e : \tau \mid \beta$. □

3.6.3 Axiomatization

We now axiomatize the CPS semantics for the typed calculus. Perhaps surprisingly, exactly the same axioms in Fig. 3.5 work for the typed calculus.

Theorem 3.5 (Soundness and Completeness)**.** *Let e_1 and e_2 be typed terms in $\lambda_{s/r}$, then $\vdash^{cbn}_{s/r} \; e_1 = e_2$ if and only if $\vdash_{CPS} \; [\![e_1]\!] = [\![e_2]\!]$.*

Proof. Since the proof is mostly the same as the type-free case, we only show an overview of the proof.

- We first establish that, if the term in the left-hand side of each axiom is typable, then the term in its right-hand side is typable and has the same typing judgement.
- We must check that all intermediate terms used in the completeness proof preserve the typing judgement.
- We must also check that the inverse translation preserves typing, and also, if e is typable in the source calculus, $[\![e]\!]^\circ$ has the same typing judgement.
- Finally, we must check that the reductions in the target calculus are mapped to typable equations in the source calculus.

We must check a lot of things, but each check can be done straightforwardly. □

3.7 Discussion on the use of linearity

We briefly discuss the use of linearity in the target of a CPS translation, which plays a significant role in our proof.

A (rough) type structure of the image of the 2CPS translation can be given as:

$$\begin{aligned}\texttt{Term} &= \texttt{Cont} \to_1 \texttt{MetaCont} \to_2 \texttt{Answer} \\ \texttt{Cont} &= \texttt{Value} \to_3 \texttt{MetaCont} \to_4 \texttt{Answer} \\ \texttt{MetaCont} &= \texttt{Value} \to_5 \texttt{Answer}\end{aligned}$$

where the indices $1, 2, \ldots, 5$ are for reference, and we have omitted to write the definition of type `Value`. As we have seen, some function types can be refined to linear function types:

[4] A simple type means a type in the simply typed lambda calculus.

$$
\begin{aligned}
[\![c]\!]_2^{SR} &= \lambda\kappa.\kappa c \\
[\![x]\!]_2^{SR} &= x \\
[\![\lambda x.e]\!]_2^{SR} &= \lambda(x,\kappa).[\![e]\!]_2^{SR}\kappa \\
[\![e_1\ e_2]\!]_2^{SR} &= \lambda\kappa.[\![e_1]\!]_2^{SR}([\![e_2]\!]_2^{SR},\ \kappa) \\
[\![\mathcal{S}k.e]\!]_2^{SR} &= \lambda\kappa.([\![e]\!]_2^{SR}\{k := \kappa\})\theta_2 \\
[\![k \hookleftarrow e]\!]_2^{SR} &= \lambda\kappa.\lambda\gamma.[\![e]\!]_2^{SR}k(\kappa \oplus \gamma) \\
[\![\langle e \rangle]\!]_2^{SR} &= \lambda\kappa.\lambda\gamma.[\![e]\!]_2^{SR}\theta_2(\kappa \oplus \gamma)
\end{aligned}
$$

Figure 3.9: SR-style 2CPS Translation

- $\to_2$ and $\to_4$ can be made linear function space $\multimap$, which corresponds to the metacontinuation variable γ. In the terminology of Berdine et al. [5], it is a *linearly used continuation*.
- $\to_3$ and $\to_5$ can be made linear function space $\multimap$, which corresponds to the variable m. This linearity reflects the fact that evaluation contexts are linear, and is specific to the call-by-name calculus. This is related to what Filinski called *linear continuations* [14].

In our proof, we have used both linearity, hence $\to_i$ for $i = 2, 3, 4, 5$ is linear function space. The remaining (non-linear) function space $\to_1$ corresponds to the non-linear uses of delimited continuations, which are useful for representing, e.g., backtracking. Berdine et al. [5] proposed to restrict this function space to be linear, and claimed that it is a "stylized" use of control.

3.8 Revisiting η-equality

We have argued the interaction between reset and full η-equality, and concluded that we should give up full η-equality, because it is unsound with respect to the intended operational semantics.

However, there can be a typed calculus which admits full η-equality if the use of reset is limited. More precisely, if we restrict the type of reset-terms to be basic types (no function types are allowed as the return type of any delimited continuation captured by shift), we can formulate a calculus in which η-equality is sound.

To develop the semantics of such a calculus, we extend Streicher and Reus' CPS translation [50]. Fig. 3.9 gives a 2CPS translation for $\lambda_{s/r}$ as an extension of Streicher and Reus' CPS translation.

The target calculus of this CPS translation is the simply typed lambda calculus augmented with pairing, denoted by $(t_1,\ t_2)$. Pairs are decomposed by an abstraction $\lambda(x,y).t$ with the additional equality $(\lambda(x,y).t_0)\ (t_1,\ t_2) = t_0\{x := t_1\}\{y := t_2\}$, and $\lambda(x,y).e(x,y) = e$ for $x, y \notin \mathtt{FV}(e)$. It should be noted the CPS translation preserves η-equality as:

$$[\![\lambda x.ex]\!]_2^{SR} =_\beta \lambda(x,k).[\![e]\!]_2^{SR}(x,k) =_{\text{pair}} [\![e]\!]_2^{SR}$$

where $=_{\text{pair}}$ means the equality induced by the two equations for pairs.

Using the CPS translation in Fig. 3.9, we can develop yet another type system for $\lambda_{s/r}$. The type system obtained through Streicher and Reus' CPS translation is different from the one obtained by Biernacka and Biernacki CPS translation.

We could axiomatize such a calculus in a similar way to this chapter. Unfortunately, the result is not really illuminating; the obtained axioms are the same axioms as those given in this chapter with η-equality ($\lambda x.ex = e$ for $x \notin \mathtt{FV}(e)$) being added. Note that a reset-term $\langle e \rangle$ is restricted to be of basic types, and therefore $\langle \lambda x.e \rangle$ is not well-typed in this calculus.

We did not develop this calculus in this chapter, because we do not know any practical meaning of such a calculus.

3.9 Concluding Remarks

In this chapter we have investigated the CPS semantics of the call-by-name calculi with delimited-control operators. We have obtained a sound and complete axiomatization for the calculus in the sense that:

$$\vdash^{\text{cbn}}_{\text{s/r}} e_1 = e_2 \text{ if and only if } \vdash_{\text{CPS}} [\![e_1]\!] = [\![e_2]\!].$$

Since η-equality interferes with reset, our axioms for the source calculus do not contain η-equality. We have proved that the same axioms work for type-free and typed cases, where the type system is (essentially) given by Biernacka and Biernacki.

Our theoretical tool was the 2CPS translation, which mixes a call-by-name translation with a call-by-value one. For the completeness proof, we need to use linearity of evaluation contexts in a non-trivial way. Also we have formulated the source calculus carefully so that lambda-bound variables and shift-bound variables must be treated differently.

We also briefly mentioned yet another typed calculus for the call-by-name shift and reset. In this calculus, η-equality is admitted with no restriction, but the type of a reset-term $\langle e \rangle$ is restricted to a basic type.

Although we do not claim that the calculus proposed by Biernacka and Biernacki (or our calculus as its small variant) is the right one for call-by-name delimited control, we believe our results can be basis for further study on call-by-name delimited-control.

Relation with Other Works. In this chapter, we have concentrated on the delimited-control operators `shift`and `reset`in the style of Danvy and Filinski and of Biernacka and Biernacki, and have not considered the relationship with others.

Herbelin and Ghilezan proposed a call-by-name calculus with control operators and connected it with classical logic [23]. They have obtained the calculus from the call-by-value calculus with delimited-control operators, but the counterpart of the reset-operator behaves quite differently from those presented in this chapter. In their call-by-value calculus, $\langle M \rangle$ is represented by $\mu \hat{\mathtt{tp}}.[\hat{\mathtt{tp}}]M$ where $\hat{\mathtt{tp}}$ is a variable for the top-level continuation. However, in their call-by-name calculus, this term reduces to M, so its behavior is quite different from the usual behavior of reset. Consequently, it is difficult to relate their calculus with other calculi. Their calculus is also different from the calculus in Section 3.8, since, in the latter calculus $\langle e \rangle = e$ does not hold if e has a basic type.

As an application of a call-by-name calculus with delimited control, Kiselyov proposed to use it in linguistic analysis [30]. His calculus has an explicit representation of evaluation contexts, and therefore it is similar to Biernacka and Biernacki's formulation. He has not given a CPS translation for his calculus, so we do not know if we can axiomatize his calculus as in this chapter.

Future work. There are a number of future works, and let us list only a few of them. (1) To investigate other delimited-control operators such as Felleisen's "control" and "prompt" in call-by-name, and if possible axiomatizing them. (2) To relate the call-by-name calculi with the call-by-value one in the sense of duality, and also with classical logic. (3) To relate our calculus with logic-based calculi such as Herbelin and Ghilezan's, and give the logical account to `reset`.

Chapter 4

A Call-by-Name CPS Hierarchy

We will explain our study about a CPS hierarchy [54] in this chapter.

4.1 Introduction

Translating terms into Continuation Passing Style (CPS) is a key to define and understand control operators such as first-class continuations (Scheme's `call/cc`). By iterating the CPS translation, Danvy and Filinski [9] have obtained a hierarchy for CPS translations, and found a series of control operators indexed by natural numbers. Among the series, the first one corresponds to the delimited-control operators `shift` and `reset`, which allows to capture a *delimited continuation*, part of the rest of computation. In the last two decades, these control operators have found many applications such as partial evaluation [35], CPS translations [9], mobile computing [51], and dependently typed programming such as type-safe `printf` [3].

The second and the rest of the series correspond to the higher-level versions of `shift` and `reset`, similar to layered monads [16], and are useful when we combine two or more computational effects in a single program [7]. Although the hierarchical (layered) control operators are less expressive than the arbitrarily nested control operators [32], the former can express many interesting programs, and also the existence of the purely functional CPS transform for the former is beneficial for studying its semantics and foundational issues.

While the above mentioned work has been done for call-by-value calculi, several authors have recently studied delimited-control operators in the call-by-name calculi. Herbelin and Ghilezan [23] and Saurin [47] studied variants of Parigot's $\lambda\mu$-calculus and interpreted their computational meaning by call-by-name delimited-control operators. Kiselyov [30] proposed a call-by-name calculus with delimited-control operators and used it in linguistic analysis. Biernacka and Biernacki [6] studied both call-by-value and call-by-name calculi for delimited-control operators in a uniform way. In the previous section (our previous work of this section), we gave a complete equational axiomatization for the call-by-name calculi with delimited-control operators.

In this Chapter, we introduce the call-by-name variant of the CPS hierarchy, in which we can find a series of delimited-control operators. The key tool we use is Hatcliff and Danvy's factorization of a CPS translation [22], which connects Plotkin's call-by-name and call-by-value CPS translations by a thunk translation. They have shown that all Plotkin's criteria for correctness can be established for the thunk translation. We use their methodology in a slightly different way: rather than defining the call-by-name CPS hierarchy independently from the call-by-value one and connect the two by the thunk translation, we use the thunk translation to *define*, or *derive*, the call-by-name CPS hierarchy from the call-by-value one. The effectiveness of our method is supported by the fact that Biernacka and Biernacki's call-by-name calculus for `shift` and `reset` [6] can be obtained by simply taking the first level of our hierarchy. Note that the thunk translation is much less complicated than the CPS translation for delimited-control operators, and we can obtain the CPS hierarchy rather smoothly.

The contribution of this Chapter is summarized as follows: (1) we extend Hatcliff and Danvy's thunk translation to the calculi with delimited-control operators, (2) we introduce a typed call-by-name CPS hierarchy, and (3) we obtain several properties of the hierarchy by the connection made by the thunk translation.

The rest of this Chapter is organized as follows. Section 4.2 gives informal explanation for the background of this work. Section 4.3 formally introduces the call-by-name calculus in a CPS hierarchy and a thunk translation. In Section

(term in Λ)	$e ::= c \mid x \mid \lambda x.e \mid ee$
(term in $\Lambda_{\mathtt{fd}}$)	$e ::= \cdots \mid \mathtt{force}(e) \mid \mathtt{delay}(e)$
(cbn-value)	$v ::= c \mid \lambda x.e$
(cbv-value)	$v ::= c \mid x \mid \lambda x.e$
(β)	$(\lambda x.e_1)e_2 \rightsquigarrow e_1\{x := e_2\}$
(β_v)	$(\lambda x.e_1)v \rightsquigarrow e_1\{x := v\}$
(fd)	$\mathtt{force}(\mathtt{delay}(e)) \rightsquigarrow e$

Figure 4.1: Syntax and Reduction Rules of the Basic Calculus

4.4, we derive a type system for our calculus and prove basic properties. Section 4.5 gives an equational theory for our calculus. Section 4.6 compares our work to others and state concluding remarks.

4.2 Preliminaries

4.2.1 Delimited-Control Operators and a CPS Hierarchy

As we mentioned in the subsection 2.3.1, explicit manipulation of continuations allows various programming styles.

However, we cannot mix two or more uses of `shift` and `reset` in one program if they do not have names to distinguish one from the other. To avoid unwanted interference between different uses of control operators, we attach natural numbers to each control operator as their indices. As the second and third examples show, `shift` chooses the nearest `reset` as the one with the same index as `shift`.

The above explanation is not completely precise: the index is not merely a name. Rather, they are linearly ordered (hence a natural number is used). In the fourth example, `shift` indexed by 1 chooses the nearest `reset` as the one indexed by 2, rather than the one by 1. In general, a higher-level `reset` delimits the continuation captured by a lower-level `shift`. Put differently, a lower-level `shift` cannot escape from a higher-level `reset`, thus control operators are layered or hierarchical.

4.2.2 The Thunk Translation

The thunk translation introduces a "thunk" (a lambda closure) to freeze a computation, and can be regarded as a translation from a call-by-name calculus to a call-by-value calculus. Hatcliff and Danvy [22] showed that Plotkin's call-by-name CPS translation can be factored into the thunk translation and Plotkin's call-by-value CPS translation, and that all Plotkin's correctness criteria [43] can be derived using this factorization. In this subsection, we briefly review their results to the extent that is necessary for this Chapter.

Fig. 4.1 gives the source and target calculi where c is a constant and v is a value in call-by-value. Λ denotes the set of terms defined in this figure. The reductions (β) and (β_v), resp., are the call-by-name and call-by-value β-reductions, resp. Bound and free variables are defined in the standard way, and we identify α-equivalent terms. The term $e_1\{x := e_2\}$ denotes the result of capture-avoiding substitution.

The thunk translation gives a simulation of the call-by-name calculus in the call-by-value calculus in terms of the following two functions: (1) `delay` for creating a suspended computation as a value (thunk), and (2) `force` for re-invoking the suspended computation. For a term e, the term $\mathtt{delay}(e)$ is a value, and $\mathtt{force}(\mathtt{delay}(e))$ reduces to e. Although we can express $\mathtt{delay}(e)$ by $\lambda x.e$, and $\mathtt{force}(e)$ by ex, for a fresh variable x, it is convenient to distinguish thunks from the ordinary lambda abstractions. $\Lambda_{\mathtt{fd}}$ denotes the set of terms with `delay` and `force`.

The thunk translation is a syntactic translation from Λ to $\Lambda_{\mathtt{fd}}$ defined in Fig. 4.2. It is easy to see that a one-step β-reduction in the source calculus one-to-one corresponds, by the thunk translation, to a one-step β-reduction in the target calculus modulo the (fd) reduction. Also, since all the arguments of functions in $\Lambda_{\mathtt{fd}}$ are values, the reductions (β) and (β_v) coincide on $\Lambda_{\mathtt{fd}}$.

Theorem 4.1 (Simulation [Hatcliff and Danvy]). *Let e_1 and e_2 be terms in Λ. We have that e_1 reduces to e_2 by the (β) reduction if and only if $\mathcal{T}[\![e_1]\!]$ reduces to $\mathcal{T}[\![e_2]\!]$ by the (β_v) reduction followed by the (fd) reductions. Moreover, (β_v) may*

$$\mathcal{T}[\![c]\!] \stackrel{\text{def}}{=} c \qquad \mathcal{T}[\![\lambda x.e]\!] \stackrel{\text{def}}{=} \lambda x.\mathcal{T}[\![e]\!]$$
$$\mathcal{T}[\![x]\!] \stackrel{\text{def}}{=} \texttt{force}(x) \qquad \mathcal{T}[\![e_1 e_2]\!] \stackrel{\text{def}}{=} \mathcal{T}[\![e_1]\!]\ (\texttt{delay}(\mathcal{T}[\![e_2]\!]))$$

Figure 4.2: Thunk Translation

$$\mathcal{C}^n[\![c]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa c \qquad \mathcal{C}^v[\![c]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa c$$
$$\mathcal{C}^n[\![x]\!] \stackrel{\text{def}}{=} \lambda\kappa.x\kappa \qquad \mathcal{C}^v[\![x]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa x$$
$$\mathcal{C}^n[\![\lambda x.e]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa(\lambda x.\mathcal{C}^n[\![e]\!]) \qquad \mathcal{C}^v[\![\lambda x.e]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa(\lambda x.\mathcal{C}^v[\![e]\!])$$
$$\mathcal{C}^n[\![e_1\ e_2]\!] \stackrel{\text{def}}{=} \lambda\kappa.\mathcal{C}^n[\![e_1]\!](\lambda m.m\mathcal{C}^n[\![e_2]\!]\kappa) \qquad \mathcal{C}^v[\![e_1\ e_2]\!] \stackrel{\text{def}}{=} \lambda\kappa.\mathcal{C}^v[\![e_1]\!](\lambda m.\mathcal{C}^v[\![e_2]\!](\lambda m'.mm'\kappa))$$
$$\mathcal{C}^v[\![\texttt{force}(e)]\!] \stackrel{\text{def}}{=} \lambda\kappa.\mathcal{C}^v[\![e]\!](\lambda m.m\kappa)$$
$$\mathcal{C}^v[\![\texttt{delay}(e)]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa\mathcal{C}^v[\![e]\!]$$

Figure 4.3: Call-by-Name (left) and Call-by-Value (right) CPS Translations

be replaced by (β) *in the above sentence.*

We consider equality over terms induced by a set of reduction rules $r_1, \cdots, r_n$, which is defined as the least congruence relation that subsumes $r_1, \cdots, r_n$. We write $(r_1, \cdots, r_n) \vdash e_1 = e_2$ if $e_1 = e_2$ holds under the equality induced by $r_1, \cdots, r_n$. For instance, $(\beta) \vdash e_1 = e_2$ means that e_1 and e_2 are equal under β equality.

Fig. 4.3 defines the call-by-value and call-by-name CPS transformations due to Plotkin [43], where the variables κ, m, and m' are fresh.

The CPS translations in Fig. 4.3 are standard except the cases for `delay` and `force`. The terms $\texttt{force}(e)$ and $\texttt{delay}(e)$ are intuitively understood as $e\ x$ and $\lambda x.e$ for a fresh variable x. Then $\mathcal{C}^v[\![\texttt{force}(e)]\!]$ is understood as $\mathcal{C}^v[\![ex]\!] = \lambda\kappa.\mathcal{C}^v[\![e]\!](\lambda m.\mathcal{C}^v[\![x]\!](\lambda n.m\ n\ k))$, which reduces to $\lambda\kappa.\mathcal{C}^v[\![e]\!](\lambda m.m\ x\ k))$. We also understand $\mathcal{C}^v[\![\texttt{delay}(e)]\!]$ as $\mathcal{C}^v[\![\lambda x.e]\!]$, which reduces to $\lambda\kappa.\kappa(\lambda x.\mathcal{C}^v[\![e]\!])$. Since x is useless, we omit it in both terms, and obtain the translation in Fig. 4.3.

Hatcliff and Danvy proved the following key theorem.

Theorem 4.2 (Factorization [Hatcliff and Danvy])**.** *For any term* e *in* Λ*, we have* $(\beta) \vdash \mathcal{C}^n[\![e]\!] = \mathcal{C}^v[\![\mathcal{T}[\![e]\!]]\!]$.

Note that η-equality (and η-reduction) is not preserved by the thunk translation: we have

$$\mathcal{T}[\![\lambda x.\ y\ x]\!] = \lambda x.\texttt{force}(y)\ (\texttt{delay}(\texttt{force}(x)))$$

and

$$\mathcal{T}[\![y]\!] = \texttt{force}(y)$$

. In order to equate these terms, we need $\texttt{delay}(\texttt{force}(x)) = x$ and $\lambda x.\texttt{force}(y)x = \texttt{force}(y)$, and the latter subsumes full η-equality, but it is not admissible in the target of the thunk translation (a call-by-value calculus).

4.3 The Calculi: Syntax and Reduction Rules

We introduce the calculi $\lambda^n_{s/r}$ for a call-by-name CPS hierarchy where n is a natural number which denotes an upper bound of indices (or levels). In other words, the indices of `shift` and `reset` must be equal to or less than n. We fix this n throughout this Chapter.

We assume that there are two disjoint sets of variables, one for ordinary variables (ranged over by $x, y, z, \cdots$) and the other for continuation variables (ranged over by k). An ordinary variable is bound by lambda, and a term may be substituted for it, while a continuation variable is bound by `shift`, and a delimited continuation may be substituted for

$(\Lambda_{\mathtt{sr}})$	$e ::= c \mid x \mid \lambda x.e \mid ee \mid \mathcal{S}_i k.e \mid \langle e \rangle_i \mid k \hookleftarrow_i e$	where $1 \leq i \leq n$
$(\Lambda_{\mathtt{sr,fd}})$	$e ::= \cdots \mid \mathtt{force}(e) \mid \mathtt{delay}(e)$	

Figure 4.4: Terms with Delimited-Control Operators

(cbv-value)	$v ::= c \mid x \mid \lambda x.e \mid \mathtt{delay}(e)$	
(cbv-context)	$E^i ::= \Box \mid E^i e \mid v E^i \mid \mathtt{force}(E^i) \mid \langle E^i \rangle_h$	where $h < i$
(β_v)	$(\lambda x.e)v \rightsquigarrow e\{x := v\}$	
(fd)	$\mathtt{force}(\mathtt{delay}(e)) \rightsquigarrow e$	
(rv_v)	$\langle v \rangle_i \rightsquigarrow v$	
(rs_v)	$\langle E^i[\mathcal{S}_i k.e] \rangle_j \rightsquigarrow \langle e\{k \Leftarrow_i \langle E^i \rangle_i\} \rangle_j$	where $i \leq j$

Figure 4.5: Call-by-Value Reductions

it. We also assume that each continuation variable k is (implicitly) annotated by a level i (for $1 \leq i \leq n$), and that $\mathcal{S}_i k.e$ and $k \hookleftarrow_i e$ are terms only when the (implicit) level of k is i.

Fig. 4.4 defines the syntax of the calculi with delimited-control operators where c is a basic constant. The term $\mathcal{S}_i k.e$ is a `shift`-term of level i, in which k is bound. The term $\langle e \rangle_i$ is a `reset`-term of level i. The term $k \hookleftarrow_i e$ is a `throw`-term of level i which applies k to the term e. Note that the continuation variable k is free in $k \hookleftarrow_i e$.

Following Biernacka and Biernacki [6], we explicitly distinguish lambda-bound (ordinary) variables from `shift`-bound (continuation) variables. This distinction is important to get simpler definitions, and is necessary to give an axiomatization for call-by-name calculus as in the previous chapter.

We identify α-equivalent terms. $\mathtt{FV}(e)$ and $\mathtt{FCV}(e)$, resp., denote the set of free ordinary variables and the set of free continuation variables, resp., in e, and $e_1\{x := e_2\}$ represents the result of capture-avoiding substitution of e_2 for x in e_1. E^i is an evaluation context which does not have a level-i or higher `reset` enclosing the hole E^1 is a pure evaluation context, or a delimited continuation, which does not have `reset` around the hole. We write E instead of E^i if the index does not matter. $E[e]$ denotes the term after the hole-filling operation of a term e for a hole in E. Hole-filling of an evaluation context $E_1[E_2]$ is defined similarly.

Fig. 4.5 gives the reduction rules for the call-by-value calculi, which are essentially due to Biernacka and Biernacki [6] with the following difference. While their calculus reifies an evaluation context E^i, and substitute it for a continuation variable k (thus $k \hookleftarrow_i e$ becomes $E^i \hookleftarrow_i e'$), our calculus uses structural substitution $\{k \Leftarrow_i E^i\}$ defined in Fig. 4.6.

The thunk translation is naturally extended to this calculus as in Fig. 4.7.

Fig. 4.8 defines the reduction rules of the call-by-name calculus. The reduction (β) is standard. The reduction (rv_n) (meaning "reset-value") eliminates a `reset` if its body is a value. The only interesting reduction is (rs_n) (meaning "reset-shift") for a level-i `shift`-term. By this reduction, `shift` captures a level-i (delimited) evaluation context E^i, and substitutes it for k. The corresponding `reset` for this `shift` is the nearest one which has the level i or higher, because E^i does not have such a `reset` that encloses the hole.

The reduction rules in both calculi are extended to arbitrary contexts as $e_1 \rightsquigarrow e_2$ implies $C[e_1] \rightsquigarrow C[e_2]$ for any context C. We write $(r_1, \cdots, r_i) \vdash e_1 \rightsquigarrow e_2$ if e_1 reduces to e_2 by these reductions. We also write $\rightsquigarrow^*$ for zero or more step reductions.

We can show that the notions of the reductions in call-by-name/call-by-value calculi correspond to each other via the thunk translation.

Theorem 4.3 (Simulation)**.** *Let e_1 and e_2 be terms in Λ_{sr}. Then we have $(\beta, rv_n, rs_n) \vdash e_1 \rightsquigarrow^* e_2$ if and only if $(\beta, fd, rv_v, rs_v) \vdash \mathcal{T}[\![e_1]\!] \rightsquigarrow^* \mathcal{T}[\![e_2]\!]$.*

Proof . *From call-by-name to call-by-value, the proof is straightforward.*

For the inverse direction, we only have to consider the image of the thunk translation, namely, for any application $e_1 e_2$, the term e_2 has the form $\mathtt{delay}(e_3)$, which is a value. Then it is not difficult to prove the inverse direction.

$$
\begin{aligned}
c\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} c && \\
x\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} x && \\
(\lambda x.e)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} \lambda x.(e\{k \Leftarrow_i E\}) && \text{where } x \notin \mathtt{FV}(E) \\
(e_1\ e_2)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} (e_1\{k \Leftarrow_i E\})\ (e_2\{k \Leftarrow_i E\}) && \\
(\mathcal{S}_p k'.e)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} \mathcal{S}_p k'.(e\{k \Leftarrow_i E\}) && \text{where } k' \notin \{k\} \cup \mathtt{FCV}(E) \\
(k \hookleftarrow_i e)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} E[e\{k \Leftarrow_i E\}] && \\
(k' \hookleftarrow_p e)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} k' \hookleftarrow_p (e\{k \Leftarrow_i E\}) && \text{where } k' \neq k \\
\langle e \rangle_p\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} \langle e\{k \Leftarrow_i E\} \rangle_p &&
\end{aligned}
$$

Figure 4.6: Substitution for Continuation Variables

$$
\begin{aligned}
\mathcal{T}[\![e]\!] &\stackrel{\text{def}}{=} \cdots \text{for } e = c,\ x,\ \lambda x.e,\ e_1 e_2 & \mathcal{T}[\![\langle e \rangle_i]\!] &\stackrel{\text{def}}{=} \langle \mathcal{T}[\![e]\!] \rangle_i \\
\mathcal{T}[\![\mathcal{S}_i k.e]\!] &\stackrel{\text{def}}{=} \mathcal{S}_i k.\mathcal{T}[\![e]\!] & \mathcal{T}[\![k \hookleftarrow_i e]\!] &\stackrel{\text{def}}{=} k \hookleftarrow_i \mathcal{T}[\![e]\!]
\end{aligned}
$$

Figure 4.7: Thunk Translation

4.4 Type System

The thunk translation is not only useful to investigate the operational aspect, but can be also used to design a type system. As a concrete instance, we design a type system for the call-by-name calculus with delimited-control operators from the one for the call-by-value calculus. In this process, we do not have to consult with the CPS translation. Although the hierarchical control operators are complicated, and in particular, a CPS translation for them is hard to understand, we can give the type system quite smoothly thanks to Hatcliff and Danvy's factorization.

Biernacka and Biernacki [6] proposed a call-by-name typed calculus with the level-1 `shift` and `reset`. While they developed the calculus and its properties independently from the call-by-value counterpart, they can be *derived* from the call-by-value counterpart using the thunk translation. In this section, we show that it is possible for the calculus of an arbitrarily higher level.

4.4.1 Type System for Call-by-Value CPS Hierarchy

We review a monomorphic type system for the call-by-value calculus with hierarchical delimited-control operators.

Murthy [39] was the first to give a type system for this calculus, which was derived from the CPS translation. The basic idea of his type system is that the typing judgment of a term e should carry the same information as its CPS image $\mathcal{C}^v[\![e]\!]$. For instance, if $n = 1$, namely, we have only one level, the CPS image of a term takes the form $\lambda\kappa_1.\lambda\kappa_2.\cdots$, which suggests the type of the CPS image is $(\sigma \to (\tau \to *) \to *) \to (\rho \to *) \to *$. Here we assume that the images of the CPS translation are in (strictly) continuation-passing style, so the (final) answer type is polymorphic [57, 58], and we write it as an anonymous type $*$. In this CPS type, σ is the type corresponding to the source term, τ and ρ are so called answer types of level 1. Murthy further assumed that, the answer types do not change, which means that τ and ρ in the above type must be the same. This assumption greatly simplifies the type system, and we only need n answer types for any term if the maximum level is n.

Answer types may be regarded as computational effects, and therefore we need to modify a function type $\sigma \to \tau$ to an effectful function type $\text{Fun}^{\text{cbv}}[\sigma \to \tau/\overline{\alpha}]$, whose inhabitants are functions from σ to τ that works under the answer types $\overline{\alpha}$. Here $\overline{\alpha}$ is a sequence of types $\alpha_1, \alpha_2, \cdots, \alpha_n$.

Let us formally define the type system. The syntax of types is given by:

$$
(\text{type})\ \ \sigma, \tau, \alpha ::= b \mid \text{Susp}[\sigma/\overline{\alpha}] \mid \text{Fun}^{\text{cbv}}[\sigma \to \tau/\overline{\alpha}]
$$

(cbn-value)	$v ::= c \mid \lambda x.e$	
(cbn-context)	$E^i ::= \Box \mid E^i e \mid \langle E^i \rangle_h$	where $h < i$
(β)	$(\lambda x.e_1)e_2 \rightsquigarrow e_1\{x := e_2\}$	
(rv_n)	$\langle v \rangle_i \rightsquigarrow v$	
(rs_n)	$\langle E^i[\mathcal{S}_i k.e] \rangle_j \rightsquigarrow \langle e\{k \Leftarrow_i \langle E^i \rangle_i\} \rangle_j$	where $i \leq j$

Figure 4.8: Call-by-Name Reductions

$$\frac{}{\Gamma, x : \sigma \vdash^{cbv} x : \sigma \mid \overline{\alpha}}\ \text{var} \qquad \frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash^{cbv} c : b \mid \overline{\alpha}}\ \text{const}$$

$$\frac{\Gamma, x : \sigma \vdash^{cbv} e : \tau \mid \overline{\alpha}}{\Gamma \vdash^{cbv} \lambda x.e : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}] \mid \overline{\beta}}\ \text{fun}$$

$$\frac{\Gamma \vdash^{cbv} e_0 : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}] \mid \overline{\alpha} \quad \Gamma \vdash^{cbv} e_1 : \sigma \mid \overline{\alpha}}{\Gamma \vdash^{cbv} e_0 e_1 : \tau \mid \overline{\alpha}}\ \text{app}$$

$$\frac{\Gamma \vdash^{cbv} e : \sigma \mid \sigma, \cdots, \sigma, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} \langle e \rangle_i : \sigma \mid \overline{\alpha}}\ \text{reset}$$

$$\frac{\Gamma, k : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}] \vdash^{cbv} e : \tau \mid \tau, \cdots, \tau, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} \mathcal{S}_i k.e : \sigma \mid \beta_1, \cdots, \beta_{i-1}, \tau, \alpha_{i+1}, \cdots, \alpha_n}\ \text{shift}$$

$$\frac{\Gamma, k : \mathrm{Fun}^{\mathrm{cbv}}[\tau \to \sigma/\sigma, \cdots, \sigma, \alpha_{i+1}, \cdots, \alpha_n] \vdash^{cbv} e : \tau \mid \sigma, \cdots, \sigma, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} k \hookleftarrow_i e : \sigma \mid \overline{\alpha}}\ \text{throw}$$

$$\frac{\Gamma \vdash^{cbv} e : \mathrm{Susp}[\sigma/\overline{\alpha}] \mid \overline{\alpha}}{\Gamma \vdash^{cbv} \mathtt{force}(e) : \sigma \mid \overline{\alpha}}\ \text{force} \qquad \frac{\Gamma \vdash^{cbv} e : \sigma \mid \overline{\alpha}}{\Gamma \vdash^{cbv} \mathtt{delay}(e) : \mathrm{Susp}[\sigma/\overline{\alpha}] \mid \overline{\beta}}\ \text{delay}$$

Figure 4.9: Type System for the Call-by-Value Calculus (First Version)

where b is a metavariable for basic types such as integer and boolean. The type $\mathrm{Susp}[\sigma/\overline{\alpha}]$ is the one for suspended computation generated by `delay`. The type $\mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}]$ is an effectful function type. A typing context Γ is a (possibly empty) sequence consisting of the form $x : \tau$ for a lambda-bound variable x, or $k : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}]$ for a continuation variable k. A judgment takes the form $\Gamma \vdash^{cbv} e : \tau \mid \overline{\alpha}$ where Γ is a typing context, τ is a type, $\overline{\alpha}$ is a sequence of types, and e is a term. The length of the sequence $\overline{\alpha}$ must be n.

Fig. 4.9 gives the type system for the call-by-value calculus.

The types for a variable, a constant, lambda, and application are standard if we take into account the answer types. The typing rule for `reset` means that, for a level-i `reset`, the answer type of the body e must be the same as the type of e itself. This reflects the hierarchical nature of this calculus: as a lower-level `shift` cannot escape from a higher-level `reset`, whenever there is a level-i `reset`, the answer types of lower levels must be identical. After the level-i `reset`, the answer types of these levels can be arbitrary types, so the types $\alpha_1, \cdots, \alpha_i$ can be chosen arbitrarily in the typing judgment of $\langle e \rangle_i$. The typing rule for the `shift`-term can be understood by noting the facts that $\mathcal{S}_i k.e$ has the same denotation as $\mathcal{S}_i k.\langle e \rangle_i$, and a level-$i$ `shift`-term captures a delimited continuation whose type is roughly a function from some type to the level-i answer type. As for the typing rule for `throw`, it is instructive to know $k \hookleftarrow_i e$ may be represented by $\langle k\ e \rangle_i$ if we forget the distinction between lambda-bound and `shift`-bound variables. The typing rules for `force` and `delay` can be understood by the following intuition: `force`(e) is ex and `delay`(e) is $\lambda x.e$ for a fresh variable x.

$$\frac{\Gamma, k : \mathrm{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n] \vdash^{cbv} e : \tau \mid \tau, \cdots, \tau, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} \mathcal{S}_i k.e : \sigma \mid \beta_1, \cdots, \beta_{i-1}, \tau, \alpha_{i+1}, \cdots, \alpha_n} \text{ shift}$$

$$\frac{\Gamma, k : \mathrm{Cont}^i[\tau \to \sigma/\alpha_{i+1}, \cdots, \alpha_n] \vdash^{cbv} e : \tau \mid \sigma, \cdots, \sigma, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} k \hookleftarrow_i e : \sigma \mid \overline{\alpha}} \text{ throw}$$

Other typing rules are the same as Fig. 4.9.

Figure 4.10: Type System for the Call-by-Value Calculus (Second Version)

4.4.2 Refining the Type System

We can refine the type system to the one in Fig. 4.10 where a continuation variable k has a distinguished continuation type $\mathrm{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n]$ (for $1 \le i \le n$), and $\sigma, \tau, \alpha_{i+1}, \cdots, \alpha_n$ are types.

The continuation variable is treated differently from the ordinary functions, and also its type does not have the information of answer types of level $\le i$, which means that it is polymorphic over these answer types. The answer type polymorphism of continuations have been studied by several authors [57, 4, 6] for the level-1 (single level) delimited-control operators, and here we use it in higher levels.

Theorem 4.4 (Type Soundness)**.** *(1) If $\Gamma \vdash^{cbv} e_1 : \tau \mid \overline{\alpha}$ is derivable, and $e_1 \rightsquigarrow^* e_2$, then $\Gamma \vdash^{cbv} e_2 : \tau \mid \overline{\alpha}$ is derivable. (2) If $\vdash^{cbv} \langle e_1 \rangle_n : \tau \mid \overline{\alpha}$ is derivable, then there exists a term e_2 such that $\langle e_1 \rangle_n \rightsquigarrow e_2$. Moreover, if e_2 is not a value, it must be $\langle e_2' \rangle_n$ for some term e_2'.*

The second part (progress) takes an unusual form since the term $\langle e_1 \rangle_n$ has an outermost reset of the maximum level n. Having such a reset is necessary, since a term with "free shift" (such as $\mathcal{S}_1 k.e$) can be a closed and typable term, but it is not a value.

The first part of this theorem is essentially due to Murthy [39], and the proof of the second part is standard.

4.4.3 Type System for Call-by-Name CPS Hierarchy

We now derive a type system for the call-by-name calculus from the one for call-by-value calculus. Our design principle is to have the property: a term e is typable in the former if and only if its translation $\mathcal{T}[\![e]\!]$ is typable in the latter.

First, we consider the case $\mathcal{T}[\![x]\!] = \mathtt{force}(x)$ which can be typed (and is only typed) in the call-by-value calculus as:

$$\frac{\overline{\Gamma,\ x : \mathrm{Susp}[\sigma/\overline{\alpha}] \vdash^{cbv} x : \mathrm{Susp}[\sigma/\overline{\alpha}] \mid \overline{\alpha}}}{\Gamma,\ x : \mathrm{Susp}[\sigma/\overline{\alpha}] \vdash^{cbv} \mathtt{force}(x) : \sigma \mid \overline{\alpha}}$$

Hence, in the call-by-name calculus we should have the following typing rule:

$$\frac{}{\Gamma,\ x : \mathrm{Susp}[\sigma/\overline{\alpha}] \vdash^{cbn} x : \sigma \mid \overline{\alpha}}$$

For notational reasons, we will write $(\sigma \mid \overline{\alpha})$ for $\mathrm{Susp}[\sigma/\overline{\alpha}]$ in typing contexts. The change in the type of a variable affects the function type: the call-by-value type $\mathrm{Fun}^{\mathrm{cbv}}[\mathrm{Susp}[\sigma/\overline{\alpha}] \to \tau/\overline{\beta}]$ will be written as the call-by-name type $\mathrm{Fun}^{\mathrm{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})]$, and the typing rule for (fun) is changed accordingly.

For the case $\mathcal{T}[\![e_0\ e_1]\!] = \mathcal{T}[\![e_0]\!](\mathtt{delay}(\mathcal{T}[\![e_1]\!]))$, we have:

$$\frac{\Gamma \vdash^{cbv} \mathcal{T}[\![e_0]\!] : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}] \mid \overline{\alpha} \quad \dfrac{\Gamma \vdash^{cbv} \mathcal{T}[\![e_1]\!] : \rho \mid \overline{\beta}}{\Gamma \vdash^{cbv} \mathtt{delay}(\mathcal{T}[\![e_1]\!]) : \sigma \mid \overline{\alpha}}}{\Gamma \vdash^{cbv} \mathcal{T}[\![e_0]\!](\mathtt{delay}(\mathcal{T}[\![e_1]\!])) : \tau \mid \overline{\alpha}}$$

where $\sigma = \mathrm{Susp}[\rho/\overline{\beta}]$. Hence we should have the following rule:

$$\frac{\Gamma \vdash^{cbn} e_0 : \mathrm{Fun}^{\mathrm{cbv}}[\mathrm{Susp}[\rho/\overline{\beta}] \to \tau/\overline{\alpha}] \mid \overline{\alpha} \quad \Gamma \vdash^{cbn} e_1 : \rho \mid \overline{\beta}}{\Gamma \vdash^{cbn} e_0\ e_1 : \tau \mid \overline{\alpha}}$$

$$\frac{}{\Gamma, x : (\sigma \mid \overline{\alpha}) \vdash^{cbn} x : \sigma \mid \overline{\alpha}}\ \text{var}$$

$$\frac{\Gamma, (x : \sigma \mid \overline{\alpha}) \vdash^{cbn} e : \tau \mid \overline{\beta}}{\Gamma \vdash^{cbn} \lambda x.e : \text{Fun}^{\text{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})] \mid \overline{\gamma}}\ \text{fun}$$

$$\frac{\Gamma \vdash^{cbn} e_0 : \text{Fun}^{\text{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})] \mid \overline{\beta} \quad \Gamma \vdash^{cbn} e_1 : \sigma \mid \overline{\alpha}}{\Gamma \vdash^{cbn} e_0 e_1 : \tau \mid \overline{\beta}}\ \text{app}$$

The typing rules for const, `shift`, `reset` and `throw` are the same as those in Fig. 4.10.

Figure 4.11: Type System for Call-by-Name Calculus

These changes are all what we need to do for the call-by-name type system. In particular, since the thunk translation is homomorphic for control operators, no essential changes are necessary in the typing rules for them.

To summarize, the types are defined by:

$$\begin{aligned} \text{(type)}\ \ \sigma, \tau, \alpha, \beta &::= b \mid \text{Fun}^{\text{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})] \\ \text{(cont-type)}\ \ \phi &::= \text{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n] \end{aligned}$$

A typing context Γ is a finite sequence of the form $x : (\sigma \mid \overline{\alpha})$ or $k : \text{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n]$. A judgment in this type system takes the form of $\Gamma \vdash^{cbn} e : \sigma \mid \overline{\alpha}$, and the call-by-name type system is given in Fig. 4.11.

Typability is preserved by the thunk translation. To state this property formally, we define the thunk-translation for types, typing contexts, and judgment as follows (the following definitions extend to sequences naturally):

$$\begin{aligned} \mathcal{T}[\![b]\!] &\stackrel{\text{def}}{=} b \\ \mathcal{T}[\![\text{Fun}^{\text{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})]]\!] &\stackrel{\text{def}}{=} \text{Fun}^{\text{cbv}}[\text{Susp}[\mathcal{T}[\![\sigma]\!]/\mathcal{T}[\![\overline{\alpha}]\!]] \to \mathcal{T}[\![\tau]\!]/\mathcal{T}[\![\overline{\beta}]\!]] \\ \mathcal{T}[\![x : (\sigma \mid \overline{\alpha})]\!] &\stackrel{\text{def}}{=} x : \text{Susp}[\mathcal{T}[\![\sigma]\!]/\mathcal{T}[\![\overline{\alpha}]\!]] \\ \mathcal{T}[\![k : \text{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n]]\!] &\stackrel{\text{def}}{=} k : \text{Cont}^i[\mathcal{T}[\![\sigma]\!] \to \mathcal{T}[\![\tau]\!]/\mathcal{T}[\![\alpha_{i+1}, \cdots, \alpha_n]\!]] \end{aligned}$$

Then we can prove the following theorem easily.

Theorem 4.5 (Thunk Translation Preserves Typability)**.** *We have that $\Gamma \vdash^{cbn} e : \tau \mid \overline{\alpha}$ is derivable if and only if $\mathcal{T}[\![\Gamma]\!] \vdash^{cbv} \mathcal{T}[\![e]\!] : \mathcal{T}[\![\tau]\!] \mid \mathcal{T}[\![\overline{\alpha}]\!]$ is derivable.*

Combining the above theorem with the property of the call-by-value CPS translation, we obtain that the call-by-name CPS translation preserves typability.

Subject reduction property can be also derived easily.

Theorem 4.6 (Subject Reduction)**.** *If $\Gamma \vdash^{cbn} e_1 : \tau \mid \overline{\alpha}$ is derivable, and $e_1 \rightsquigarrow^* e_2$, then $\Gamma \vdash^{cbn} e_2 : \tau \mid \overline{\alpha}$ is derivable.*

Proof . *Suppose $\Gamma \vdash^{cbn} e_1 : \tau \mid \overline{\alpha}$ is derivable, and $e_1 \rightsquigarrow^* e_2$ in the call-by-name calculus. By Theorem 4.3, we have $\mathcal{T}[\![e_1]\!] \rightsquigarrow^* \mathcal{T}[\![e_2]\!]$. By the subject property of the call-by-value calculus and Theorem 4.5, we have $\mathcal{T}[\![\Gamma]\!] \vdash^{cbv} \mathcal{T}[\![e_2]\!] : \mathcal{T}[\![\tau]\!] \mid \mathcal{T}[\![\overline{\alpha}]\!]$. Again by Theorem 4.5, we have $\Gamma \vdash^{cbn} e_2 : \tau \mid \overline{\alpha}$.*

We also have the progress property for the call-by-name calculus.

Theorem 4.7 (Progress)**.** *If $\vdash^{cbn} \langle e_1 \rangle_n : \tau \mid \overline{\alpha}$ is derivable, there exists a term e_2 such that $\langle e_1 \rangle_n \rightsquigarrow e_2$. Moreover, if e_2 is not a value, it must be $\langle e_2' \rangle_n$ for some term e_2'.*

Proofs of these theorems are straightforward and omitted.

We have derived a type system for the call-by-name CPS hierarchy. Note that, by taking $n = 1$, we can reproduce Biernacka and Biernacki's call-by-name type system [6] modulo notational difference. A merit of our approach is that we do not have to directly consult the iterated CPS translation.

$$(\lambda x.e_1)\ e_2 = e_1\{x := e_2\} \qquad (\beta)$$
$$\langle E^i[\mathcal{S}_i k.e]\rangle_j = \langle e\{k \Leftarrow \langle E^i\rangle_i\}\rangle_j \ \text{ where } \ i \le j \qquad (\mathrm{rs}_n)$$
$$k' \hookleftarrow_j (E^i[\mathcal{S}_i k.e]) = \langle e\{k \Leftarrow (k' \hookleftarrow_j E^i)\}\rangle_j \ \text{ where } \ i \le j \text{ and } k \neq k' \qquad (\mathrm{ts})$$
$$\langle v\rangle_i = v \qquad (\mathrm{rv}_n)$$
$$\mathcal{S}_i k.(k \hookleftarrow_i \langle e\rangle_{i-1}) = \langle e\rangle_{i-1} \ \text{ where } \ k \notin \mathtt{FCV}(e) \qquad (\mathrm{se})$$
$$\mathcal{S}_i k.\langle e\rangle_i = \mathcal{S}_i k.e \qquad (\mathrm{sr})$$

Figure 4.12: Equational Theory for the Call-by-Name Calculus

4.5 Equational Theory

This section studies an equational theory for the call-by-name CPS hierarchy in the typed setting. Sabry and Felleisen [45] first established equational axiomatization of the calculus with control operators for first-class (unlimited) continuations. Regarding delimited-control operators, Kameyama and Hasegawa [26] and Kameyama [25], resp., axiomatized level-1 and higher-level, resp, `shift` and `reset` in the call-by-value calculi. For reference, Fig. 3.6 in the appendix lists the latter axiomatization.

We obtain an equational theory for the call-by-name calculus in the same spirit as the previous sections: we formulate them as a back image of the thunk translation and the call-by-value counterpart. The result is given in Fig. 4.12.

The call-by-name axioms (Fig. 4.12) and the call-by-value axioms (Fig. 3.6) have several differences. It should also be noted that, even if the axioms rs_v and rs_n have the same form, they have different meaning as the definitions of the evaluation contexts differ from each other.

We write $\vdash^{cbn} e_1 = e_2$ if the equation is derivable using the equations in Fig. 4.12. Similarly we write $\vdash^{cbv} e_1 = e_2$ for the equations in Fig. 3.6. It is easy to prove that the reduction semantics is subsumed by these our equations.

Theorem 4.8. *If $(\beta, rv_n, rs_n) \vdash e_1 \rightsquigarrow^* e_2$, then $\vdash^{cbn} e_1 = e_2$ is derivable.*

We can also prove that the call-by-name equations are sound with respect to the thunk translation, and hence the call-by-name CPS translation.

Theorem 4.9 (Soundness)**.** *If $\vdash^{cbn} e_1 = e_2$, then $\vdash^{cbv} \mathcal{T}[\![e_1]\!] = \mathcal{T}[\![e_2]\!]$.*

This theorem can be proved by simple calculations for each equation. It immediately implies soundness of call-by-name equations with respect to the call-by-name CPS translation in the appendix.

Corollary 1. *If $\vdash^{cbn} e_1 = e_2$, then $(\beta, \eta) \vdash \mathcal{C}^n[\![e_1]\!] = \mathcal{C}^n[\![e_2]\!]$.*

Finally, an interesting question is whether the equations are complete with respect to the thunk translation. Unfortunately, we have not succeeded in directly proving completeness using the thunk translation, since we cannot define the inverse of the thunk translation which preserves equality[1]. However, we can use our previous results to connect the call-by-name and call-by-value theories for the first level:

Theorem 4.10 (Correspondence in the First Level)**.** *Suppose the maximum level n is 1, and $e_1, e_2 \in \Lambda_{sr}$. Then we have $\vdash^{cbn} e_1 = e_2$ if and only if $\vdash^{cbv} \mathcal{T}[\![e_1]\!] = \mathcal{T}[\![e_2]\!]$.*

Proof . *Theorem 4.9 states soundness (the only-if direction). For completeness (the if direction), suppose $\vdash^{cbv} \mathcal{T}[\![e_1]\!] = \mathcal{T}[\![e_2]\!]$. By the soundness of the call-by-value equational theory, we have $(\beta, \eta) \vdash \mathcal{C}^v[\![\mathcal{T}[\![e_1]\!]]\!] = \mathcal{C}^v[\![\mathcal{T}[\![e_2]\!]]\!]$, and hence $(\beta, \eta) \vdash \mathcal{C}^n[\![e_1]\!] = \mathcal{C}^n[\![e_2]\!]$. By the completeness of the call-by-name equations for the first-level showed in the previous chapter [27], we have $\vdash^{cbn} e_1 = e_2$.*

[1] Note that η_v is admissible in the call-by-value calculus, while η-equality is not admissible in the call-by-name calculus.

4.6 Concluding Remarks

We have introduced the CPS hierarchy in call-by-name. Based on the thunk translation and factorization, we have derived a calculus and a type system, and proved several interesting properties of our system. The simplicity of the thunk translation makes it easy to treat a complex machinery such as the CPS hierarchy, and we do not have to directly consult the iterated CPS translations for most of the time.

This work builds on Danvy and Filinski's CPS Hierarchy, and is related to recent works on call-by-name delimited continuations. Among all, Saurin [46] proposed "Stream Hierarchy" as a call-by-name CPS Hierarchy, and developed a very interesting theory for this calculus, as it combines $\lambda\mu$-calculus in logic and streams in functional programming. He used a quite different CPS translation than ours, namely, η-equality is admissible in his theory, and thus his delimiter (`reset`) behaves quite differently from ours. Our theory does not admit η-equality since it badly interacts not only with the CPS translation, but also with the semantics of `reset`, since if we have full η-equality, we can convert every term to a value, which makes `reset` meaningless.[2]

As future work, we hope to relate the call-by-name calculi with the call-by-value one in the sense of duality, and also with classical logic.

[2]Using η-equality, we can derive $\langle e \rangle_i = \langle \lambda y.ey \rangle_i = \lambda y.ey = e$.

Chapter 5

The Semantics of Future with Delimited Control

We will explain our study about a calculus containing a parallelization operator [56] in this chapter.

5.1 Introduction

It is an important but a difficult problem how we parallelize programs using (delimited) control operators. As multi-core/many-core processors emerged as the mainstream hardware technology in recent years, the desire for parallel execution of existing programs are growing. When one wants to run programs with control operators in a parallel environment, he or she immediately encounters several problems: (1) what does a continuation mean when there are more than one thread, (2) how to manipulate a continuation in parallel environment, and (3) whether the notion agrees with the one in sequential execution. The original notion of continuations (and hence the meaning of control operators) assumes that a program is executed sequentially, and if we run a program with control operators in parallel without any restriction, the result of the computation varies from time to time, resulting in a non-deterministic computation. We think this is a undesirable situation, as we want to build testable, maintainable, and verifiable programs that will eventually lead to a dependable software.

Katz and Weise [28] and Moreau [37] proposed languages which contain both `call/cc` and `future`. The operator `future` was originally introduced in MultiLisp [21]. `future` operator is a language primitive for explicit parallelism. It takes one argument, and if the expression (`future` e) is executed, it immediately returns a dummy value to the surrounding expression, and the two computations (the computation for e and the surrounding one) are run in parallel. When the latter needs the actual value of (`future` e), it is suspended until the computation of e is finished. In this way, `future` introduces parallelism into a certain program point determined by the programmer.

Katz and Weise showed how to implement such `future` in Scheme (which means that the base language has `call/cc`) but they showed nothing about the property of control operators in the parallel environment. On the other hand, an important goal of Moreau's research is that `future` is transparent: for any programs containing `future`s, the program obtained by removing all `future`s form it has the same meaning as the given one. Moreau [38] defined a calculus with `call/cc` and `future` and proved the transparency of `future`.

Although he achieved his goal with a mathematically rigid proof of transparency, his calculus has a problem from the viewpoint of programming language designs: parallelism is severely destroyed whenever `call/cc` is executed.

It is because the control operator `call/cc` captures the entire rest of a computation, and to preserve the sequential semantics of `call/cc`, the other parts of a program must not be executed once `call/cc` is called. In other words, he succeeded in designing a transparent calculus with `call/cc` and `future`, but we cannot actually run a program with control operators in parallel. As our aim is to have true parallelism for control operators, we must conclude that Moreau's calculus is not very useful for our purpose.

In this research, we propose a parallel calculus with delimited-control operators: this calculus has `future` and `shift/reset`. For this calculus, we will define a rigorous semantics based on two abstract machines, one is for sequential semantics and the other is for parallel one, then we will prove the transparency of `future` by showing that the two abstract machines must have same semantics. Using `shift/reset` instead of `call/cc` has two advantages.

First, `shift/reset` can express more elaborate computational effects than that of `call/cc`. Second, programs can have more parallelism than ones with `call/cc` because `reset` decides the extent of a continuation, which means that programmers can delimit the extent of computational effects caused by `shift` (i.e., capturing the continuation).

As we said above, we proved the transparency rigorously. Our proving method is based on Takahashi's parallel reduction [52]. By using this method, we can construct clearer proof of the transparency than Moreau's in the sense that we can omit the troublesome discussion about the number of the reduction steps.

We will propose a semantics for our calculus λ_{sr}^{ft} in Section 5.3, which maximizes the possibility of parallel computations, while relating transparency of `future`. There are several alternative semantics for our calculus, and we will argue them with brief comparison with ours.

(1) The first way is to remove `shift/reset` by a program translation. By CPS translating a program with `shift/reset` [10], we can eliminate all occurrences of `shift/reset` from the program, and then the resulting program with `future` (which does not contain control operators) can be executed in parallel.

A problem in this approach is that, a CPS translation turns a program into a fully sequentialized one, which means that it removes the chances of parallelism in the source program. In particular, the expression $(\texttt{future}\ e)$ is sequentialized, against our purpose.

(2) The second way is to macro-define `shift/reset` in terms of `call/cc` and a mutable cell [15]. This is possible in Moreau's calculus [38], since it has both `call/cc` and mutable cells.

A problem in this approach is that both `shift` and `reset` are defined with `call/cc`. As we described above, `call/cc` always sequentializes its execution of a program even if it is executed in parallel, which means that our goal (using `shift/reset` in truly parallel environment) cannot be achieved in this approach.

(3) The third alternative semantics is to restrict the syntax of terms so that, for each subterm $(\texttt{future}\ M)$, M must be in the form $\langle M' \rangle$. This restriction makes sense theoretically, since the proof of transparency becomes trivial, as `shift` cannot capture a (delimited) continuation across the boundary of processes. However, as it severely restricts the possible occurrences of future to be immediately above a `reset`, it has a negative impact on the degree of parallelization; in particular, we cannot parallelize a program without control effects at all. We note that the current implementation of Racket follows this restriction.

(4) The last alternative semantics we discuss here is to allow pruning of some processes; for a term $\langle \texttt{flet}\ (p\ M)\ S \rangle$, if M is free from computational effects other than termination, and S does not contain p free, we can reduce it to S, discarding the process for computing M. While this strategy would give a better performance than ours, it is hard to implement in our setting, since freedom of any effects (purity) is not easily computable. One would be able to design a type and effect system to determine purity, while dynamic recomputation of effects severely penalizes the run-time performance.

There are other studies about `future` as follows:

- Niehren et al. presented calculi which have `future` and mutable cells [41, 40]. But they do not focus on the transparency: their goal is to show the expressive power of `future`. In fact, we can write a program using `futures` that may return different values for execution.
- Komiya and Yuasa proposed an implementation of `call/cc` in a parallel-computation environment.

The rest of this chapter consists as follows: in the next section, we will show examples and describe intuitive meaning of `future`. In the section 5.3, we will define a calculus and two abstract machines to define the semantics of our calculus. Then, in the section 5.4, we will prove the transparency of `future`.

5.2 Examples

In this section, we will show a simple example for `future` and explain its behavior intuitively. After that, we will show other examples such as A-normal form translation and tree search. The last two examples will show why we need transparency for the `future` operator.

5.2.1 `future`

`future` is an operator which executes two expressions in parallel; the one is an argument of `future` and the other is the surrounding expression (i.e., the evaluation context) of the expression $(\texttt{future}\ e)$. In this example, we use a notation $e_1 \mid e_2$ as a parallel execution of two expressions e_1 and e_2. Also we will assume that common expressions such as addition and let-expression can be used.

$$(\texttt{let}\ ((x\ (\texttt{future}\ (+\ 1\ 10))))\ (+\ 100\ 1000\ x)) \tag{5.1}$$
$$\rightsquigarrow(\texttt{let}\ ((x\ p))\ (+\ 100\ 1000\ x)) \mid p = (+\ 1\ 10) \tag{5.2}$$
$$\rightsquigarrow(+\ 1100\ p) \mid p = 11 \tag{5.3}$$
$$\rightsquigarrow(+\ 1100\ 11) \rightsquigarrow 1111 \tag{5.4}$$

When the expression $(\texttt{future}\ (+\ 1\ 10))$ is evaluated in the line (5.1), `future` splits the expression into two expressions running in parallel. The one is $(+\ 1\ 10)$ which is the argument of the `future` operator and the other is $(\texttt{let}\ ((x\ \square))\ (+\ 100\ 1000\ x))$ which is the evaluation context of the $(\texttt{future}\ e)$. The latter cannot begin to be evaluated without some value within the hole ($\square$) so it uses a special value p to fill its hole. As the result, two expressions are evaluated simultaneously as in the line (5.2). An important point is that a special variable p is not just a dummy value but a communication tool of the two expressions: when the expression $(+\ 1\ 10)$ is finished and returns a value 11 as in the line (5.3), variable p sends the value to the other expression. The other expression in the expression (5.3) is $(+\ 1100\ p)$ so it just needs the actual value of p. Then the special value p becomes to be a value 11 which means that the expression $(+\ 1100\ p)$ becomes $(+\ 1100\ 11)$ as in the last line.

5.2.2 A-Normal Form Translation

In this subsection, we will show a program for A-normal form translation. A-normal form translation is a kind of program translations: it gets a program and returns the program translated in A-normal form. The A-normal form is a form which contains no nested function-application expressions. In other words, all subexpressions are named and totally sequentialized to reflect its control flow. For example, A-normal form of an expression

```
(f (g x))
```

is as follows:

```
(let ((m1 (g x))) (let ((m2 ((f m1)))) m2))
```

There are two new variables `m1` and `m2` in the latter program and nested function-application expression `(g x)` is moved to the outside of the other application.

Asai [2] showed that an A-normal form translation can be defined by using `shift/reset`. In this subsection, our programs will be written in Scheme syntax and we will use quasi-quote mechanisms in the programs.

```
1  (define (a exp)
2    (cond [(symbol? exp) exp] ; for variable
3          [(eq? (car exp) 'lambda) ; for lambda abstraction
4           (let ((var (car (car (cdr exp))))
5                 (body (car (cdr (cdr exp)))))
6             `(lambda (,var) ,(reset (a body))))]
7          [else ; for application
8           (shift k
9             (let ((m (gensym 'm)) (n (gensym 'n)))
10              `(let ((,m ,(a (car exp)))
11                     (,n ,(a (car (cdr exp)))))
12                ,(k `(,m ,n)))))]))
```

The function `a` gets a lambda expression `exp` as a quoted list, and returns translated lambda-abstraction expression [1]. The translation is constructed with three parts.

(1) When the `exp` is a variable, it just returns the `exp`.

[1] For the sake of the ease, we assume that the let-expression is evaluated definitely up-to-down order

(2) When the `exp` is a lambda abstraction, it translates the body of the abstraction and returns the translated function. In this case, `reset` is put on the outside of the translation. The reason will be described below.

(3) When the `exp` is an application `(e1 e2)`, the essential behavior of the translation is simple: (1) it generates two fresh variables `m1` and `n1`, (2) it translates two expressions `e1` and `e2` respectively (and call them `a1`, `a2` respectively), and (3) returns the resulting expression `(let ((m1 a1) (n1 a2)) (m1 n1))`.

However, if `(f (g x))` is translated by only the way, its result will be like `(let ((m1 f) (n1 (let ((m2 g) (n2 x)) (m2 n2)))) (m1 n1))`, which is not in A-normal form. `shift` put on the translation of application is needed to resolve this problem. We will show how the expression is translated with `shift` below.

We show the steps of this translation which starts from the form `(reset (a '(f (g x))))`. Here we assume that `f`, `g` and `x` are variables and the translation will start with outer-most `reset`. As the term `(f (g x))` is an application, the first step results in the term as follows:

```
(reset (shift k `(let ((m1 ,(a 'f)) (n1 ,(a '(g x))))
                   ,(k `(m1 n1)))))
```

Then `shift` is executed. (As the continuation captured by `shift` is an empty one, we omit its application in the following program.)

```
(reset `(let ((m1 ,(a 'f)) (n1 ,(a '(g x)))) (m1 n1)))
```

Two translations `(a 'f)` and `(a '(g x))` are executed respectively.

```
(reset `(let ((m1 f)
              (n1 ,(shift k1 `(let ((m2 g) (n2 x))
                                ,(k1 `(m2 n2)))) ))
          (m1 n1)))
```

Note that, in the above, `` `(let ((m1 f) (n1 ...)) (m1 n1)) `` is the quoted expression but the expression with back-quote is unquoted so the expression `` `(let ((m2 g)) ...) `` will be executed. Then `shift` will capture the continuation and bind it to the variable `k1`.

Finally, the `k1` is applied to the quoted expression `` `(m2 n2) ``.

```
(reset `(let ((m2 g) (n2 x))
          (let ((m1 f) (n1 (m2 n2))) (m1 n1))))
```

The final result is the expression removed the quasi-quote and `reset`. [2]

5.2.2.1 Parallelizing A-Normal Form Translation

We will consider how we parallelize the A-normal form translation program. The first naive idea is translating two expressions in parallel when an application expression `(e1 e2)` is translated. By this way, however, if `e1` is in the form `(e11 e12)`, it immediately aborts the other translation process executing `(a e2)` because of `shift` in the part of application-expression translation.

To avoid this problem, we parallelize the part of the lambda-abstraction translation. As in the definition of the function `a`, the translation of the lambda-abstraction expression encloses the call of the function `a` by `reset`, so even if the translation of the function body calls `shift`, it does not affect other translation processes running in parallel outside of `reset`.

```
1| (define (a exp)                                                          ||
2|   (cond [...] ; same as in sequential                                    ||
3|         [(eq? (car exp) 'lambda) ; for lambda abstraction                ||
4|          (let ((var (car (car (cdr exp))))                              ||
5|                (body (car (cdr (cdr exp)))))                            ||
6|            `(lambda (,var) ,(future (reset (a body))))))]               ||
7|         [...] ; same as in sequential                                    ||
8|   ))
```

[2] In this example, we assume that variables are also expressions, which means that if the argument of this translation function is just a variable, it is also named. While this is not equal to the original definition of A-normal form translation, there are no any essential differences between them.

The only difference from the original definition is that the translation of the lambda-abstraction expression calls the function `a` within `future` operator, which means that, for example, if we translate an expression such as `((lambda(x) e1) e2)`, the two translation for `e1` and `e2` is executed in parallel.

An important point is that this kind of parallelization is difficult in Moreau's calculus because the extent of a continuation captured by `call/cc` cannot be delimited. In other words, as `shift` cannot affect over the nearest `reset`, if other processes are in the out of `reset`, they are not affected by `shift`.

To make the above description be clearer, we will introduce an idea which we make the translation function be runnable in parallel by using `call/cc` and a mutable cell.

As we described in previous section, we can define control operators which is the same as `shift/reset` with `call/cc` and a mutable cell [15], and we call them `SFT` and `RST`. Hence, by the two control operators, we can write A-normal form translation function with them on the face of it.

A problem of this way is that these pseudo `shift/reset` need a mutable cell in order to manage the outside of `reset` (`RST`) [3].

To show the problem, we will show an example of translation processes by a parallelized A-normal form translation function defined with `SFT/RST`. An intermediate result of the translation can be represented as follows:

```
(RST `(let ((m1 (lambda (x) ,(future (RST (a e1))))))
        (let ((m2 (lambda (y) ,(future (RST (a e2))))))
          (m1 m2))))
```

Then `future` in the first line makes a new process which translates `e1`, and also the other `future` makes another process which translates `e2`. While the two new processes are running in parallel, the latter process cannot access the mutable cell shared with the two process if the former process is running yet.

This means that the latter process cannot continue its translation while the former one is running and continuing to translate.

That is the reason why we cannot parallelize the A-normal form translation function in Moreau's calculus.

5.2.3 Binary search

We will introduce another example which traverses a tree, searches a target element and does something to the target. The example program is as follows:

```
1| (define (search tree pred proc)                                          ||
2|   (cond [(null? tree) '()]                                              ||
3|         [(pred tree) (proc tree)]                                        ||
4|         [else (begin (future (search (cadr tree) pred proc))            ||
5|                       (search (caddr tree) pred proc))]))                ||
```

The function `search` gets three arguments: a binary tree `tree`, a predicate `pred` and a procedure `proc`. The argument `pred` gets one argument and returns a boolean value. The argument `proc` is a function which is applied to a tree's element when the application of `pred` to the element returns `true`. The argument is a binary tree which is in the form `'(data left right)`: `data` is an arbitrary value and `left` and `right` are binary trees respectively. We used Scheme's primitive functions such as `cadr` and `caddr`, which get a list and return a second and third element of the list respectively. In the third line, this function examine the value of `(pred tree)`; if the value is `true` then it applies `proc` to the `tree`, and otherwise it searches the both child trees recursively.

This program is parallelized in the fourth line by enclosing the search of the left tree by `future`.

One of the simplest procedure which will be applied to the found target is a function printing the element (on the screen). Using `shift/reset`, we can define more useful procedures than such a simple one which we can control the behavior of the `search` function without rewriting the function directly.

```
(define (proc tr)
  (shift k (cons tr k)))
```

[3] Here we assume the implementation of `SFT/RST` which is in Gasbichler's literature [19]

$$
\begin{array}{rcl}
M & ::= & x \mid a \mid f \mid (\lambda x.M) \mid (M_1\ M_2) \mid \langle M \rangle \mid (\mathcal{S}k.M) \mid (\texttt{future}\ M) \\
a & ::= & 0 \mid 1 \mid 2 \cdots \\
f & ::= & \texttt{make} \mid \texttt{set!} \mid \texttt{deref} \\
x & ::= & x \mid y \mid z \mid k \mid v \cdots
\end{array}
$$

Figure 5.1: Syntax of λ_{sr}^{ft}

When this function gets an argument `tr`, it captures the delimited continuation and immediately returns a pair of values which contains the found (passed to this function) target `tr` and the captured continuation `k`.

The continuation bound to the variable `k` is just a continuation of the search so, for example, if the value contained in the returned value is not appropriate, we can search another target arbitrarily.

An interesting discussion is caused by this kind of behavior of `shift`; how does the processes contained in the continuation captured by `shift` behave? Readers may think that it is not realistic to halt or freeze other processes by the result of the behavior of control operators. We will show one of the answers to resolve the artificiality.

5.3 An Operational Semantics based on Abstract Machines

In this section, we will define our calculus λ_{sr}^{ft} with its syntax and the semantics. The semantics will be defined by two abstract machines, which we call AM-PFSR and AM-FSR respectively. The former processes `future` as a parallelization operator, and the latter does not processes it, which means the abstract machine AM-FSR ignores all `futures` in programs.

5.3.1 Syntax

The syntax of λ_{sr}^{ft} is defined in Figure 5.1. We will express the operator `shift` as $\mathcal{S}$ and the operator `reset` as $\langle - \rangle$. A term M consists of `shift`, `reset` and `future` along with common features such as variables, abstractions and applications. The functional constants `make`, `ref` and `set!` are used to generate, to read and to write mutable cells respectively. In the definition of terms, $(\lambda x.M)$ binds a free variable x in the term M, and also $(\mathcal{S}k.M)$ binds a free variable k in the term M. The bound and free variables are defined as usual, and we identify α-equivalent terms.

5.3.2 States of AM-PFSR

The definitions of states, contexts and stores of AM-PFSR are defined in Figure 5.2. We omit the definitions same as the ones in Figure 5.1.

A state S is defined as a pair of a store (θ) and a term (M). A constructor `flet` is a key term of our calculus, which appears when a term $(\texttt{future}\ M)$ is evaluated to denote a parallel execution.

There are new kinds of variables, p and b, in Figure 5.2. p is a communication variable which is generated when a term $(\texttt{future}\ M)$ is evaluated, and b is a box variable which is a reference cell, i.e., a mutable variable.

In the following, we ignore the difference between (ordinary) variables and the above variables unless we present any kinds of notions.

There are four kinds of contexts:

- C is an arbitrary contexts,
- E is a (general) evaluation contexts,
- F is a pure-evaluation contexts, and
- G is a truly pure-evaluation contexts.

State

$$
\begin{array}{rcl}
S & ::= & [\![\theta,\ M]\!] \\
M & ::= & V \mid (M_1\ M_2) \mid \langle M \rangle \mid (\mathcal{S}k.M) \\
 & & \mid (\texttt{future}\ M) \mid \langle \texttt{flet}\ (p\ M)\ S \rangle \\
V & ::= & c \mid x \mid b \mid p \mid (\lambda x.M) \mid (\texttt{set!}\ V) \\
b & ::= & b \mid b_1 \mid b_2 \cdots \\
A & ::= & c \mid \textbf{box} \mid \textbf{procedure} \mid \textbf{error}
\end{array}
\qquad
\begin{array}{rcl}
c & ::= & a \mid f \mid \textbf{void} \\
p & ::= & p \mid p_1 \mid p_2 \cdots
\end{array}
$$

Context

$$
\begin{array}{rcl}
C & ::= & \Box \mid (\lambda x.C) \mid (C\ M) \mid (M\ C) \mid \langle C \rangle \mid (\mathcal{S}k.C) \mid (\texttt{future}\ C) \\
 & & \mid \langle \texttt{flet}\ (p\ C)\ S \rangle \mid \langle \texttt{flet}\ (p\ M)\ [\![\theta,\ C]\!] \rangle \\
E & ::= & G \mid E[F] \\
F & ::= & \langle G \rangle \mid \langle \texttt{flet}\ (p\ G)\ S \rangle \\
G & ::= & \Box \mid (G\ M) \mid (V\ G)
\end{array}
$$

Store

$$\theta\ ::=\ \{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\} \text{ (each } b_i \text{ are unique variables)}$$

Figure 5.2: States, Contexts and Store of AM-PFSR

$C[M]$ stands for a hole-filling term, which is constructed by replacing the $\Box$ in the context C by the term M. The differences between E and F is whether the context can contain `reset` or not. An evaluation context E can contain an arbitrary number of `reset`s but a pure-evaluation contexts F can only have outer-most `reset`. A truly pure-evaluation context G cannot contain any `reset`.

A store θ is an environment of box variables: it is a set which consists of pairs, and the pair consists of a box variable and a value. We also define the domain of a state as follows:

$$\mathit{Dom}(\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\}) = \{b_1, \cdots, b_n\}$$

Free communication variables are bound in the state S of a term $\langle \texttt{flet}\ (p\ M)\ S \rangle$. Free box variables are bound in the state $[\![\theta,\ M]\!]$ when $b \in \mathit{Dom}(\theta)$. If p and b are not bound, we call them free. Two terms, contexts, stores and states are identified when the two have no differences except differences of the names of the variables in the ones.

We call a term *program* if the term is in the form $\langle M \rangle$ and M does not contain any free variables without box variables. We represent a programs by another meta variable P. Also we call a state *program state* if the term of the state is a program P, and $[\![\theta,\ M]\!]$ has no free variables.

5.3.3 Transition Rules of AM-PFSR

In this subsection, we will show the transition rules of AM-PFSR in two figures, Figure 5.3 and Figure 5.4: the former contains the rules for sequential transitions and the latter contains the ones for parallel transitions.

In Figure 5.3 (and also in Figure 5.4), we assume that the term contained in the state is a program because several rules cannot be applied to a state whose term has no `reset`.

(*app*) is rule for a function-application expression. $M\{x := V\}$ means an usual, capture-avoiding substitution.

(*rv*) is a rule to remove `reset` if a term is in the form $\langle V \rangle$. (*rs*) is a rule to capture the delimited-continuation F which is the continuation up to the nearest `reset`. The continuation becomes a function $(\lambda v.F[v])$ and it is bound to the variable k.

Box variables are generated by a rule (*make*). We assume that the name of a box variable is unique, which means the name of a generated variable must not be in the domain of the state. The value bound to a box variable is read by the rule (*deref*) and written by the rule (*set*).

$$\llbracket \theta,\ E[((\lambda x.M)\ V)] \rrbracket \to \llbracket \theta,\ E[M\{x := V\}] \rrbracket \quad (\textit{app})$$
$$\llbracket \theta,\ E[\langle V \rangle] \rrbracket \to \llbracket \theta,\ E[V] \rrbracket \quad (\textit{rv})$$
$$\llbracket \theta,\ E[F[(\mathcal{S}k.M)]] \rrbracket \to \llbracket \theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle] \rrbracket\ (v \text{ fresh}) \quad (\textit{rs})$$
$$\llbracket \theta,\ E[(\texttt{make}\ V)] \rrbracket \to \llbracket \theta \cup \{b \mapsto V\},\ E[b] \rrbracket\ (b \text{ fresh}) \quad (\textit{make})$$
$$\llbracket \theta \cup \{b \mapsto V_1\},\ E[((\texttt{set!}\ b)\ V)] \rrbracket \to \llbracket \theta \cup \{b \mapsto V\},\ E[\textbf{void}] \rrbracket \quad (\textit{set})$$
$$\llbracket \theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)] \rrbracket \to \llbracket \theta \cup \{b \mapsto V\},\ E[V] \rrbracket \quad (\textit{deref})$$

Figure 5.3: Sequential Transition Rules

$$\llbracket \theta,\ E[F[(\texttt{future}\ M)]] \rrbracket \to \llbracket \theta,\ E[\langle \texttt{flet}\ (p\ M)\ \llbracket \emptyset,\ F[p] \rrbracket \rangle] \rrbracket\ (p \text{ fresh}) \quad (\textit{fork})$$
$$\llbracket \theta,\ E[\langle \texttt{flet}\ (p\ V)\ \llbracket \theta_1,\ M_1 \rrbracket \rangle] \rrbracket \to \llbracket \theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}] \rrbracket \quad (\textit{join})$$
$$(\text{when } \mathit{Dom}(\theta) \cap \mathit{Dom}(\theta_1) = \emptyset)$$
$$\llbracket \theta,\ C[\langle \texttt{flet}\ (p\ M)\ S \rangle] \rrbracket \to \llbracket \theta,\ C[\langle \texttt{flet}\ (p\ M)\ S' \rangle] \rrbracket \text{ if } S \to S' \quad (\textit{spec})$$
$$\llbracket \theta \cup \{b \mapsto V\},\ M \rrbracket \to \llbracket \theta \cup \{b \mapsto V'\},\ M \rrbracket \text{ (where } \llbracket \emptyset,\ V \rrbracket \to \llbracket \emptyset,\ V' \rrbracket) \quad (\textit{store})$$

Figure 5.4: Parallel Transition Rules

We will define transition rules for parallel computation in Figure 5.4.

(*fork*) is a rule to process `future` as a parallelization operator. When a state S transits to another state S' by this rule, the state S' has a term containing a constructor `flet`. (We call a term in the form $\langle \texttt{flet}\ (p\ M)\ S \rangle$ flet-term.) Roughly speaking, a flet-term $\langle \texttt{flet}\ (p\ M)\ S \rangle$ denotes that the term M and the state S is running in parallel. M can be a target of a transition because, for an arbitrary evaluation contexts E, the context $E[\langle \texttt{flet}\ (p\ \Box)\ S \rangle]$ is also an evaluation context. And, S can transit to another states by the rule (*spec*), which we will describe below.

(*join*) works to merge two states. This rule can be applied only if a state has a flet-term like $\langle \texttt{flet}\ (p\ V)\ S \rangle$. When a state $\llbracket \theta,\ E[\langle \texttt{flet}\ (p\ V)\ \llbracket \theta_1,\ M_1 \rrbracket \rangle] \rrbracket$ transits to another state by (*join*) rule, the merged state is as follows:

- the store is $\theta \cup (\theta_1\{p := V\})$ which is a union of the two stores. (Here, we assume that the names of box variables in the both stores are renamed to avoid name collision automatically.),
- the term is $E[M_1\{p := V\}]$ which means that the evaluation context E surrounding the flet-term is filled by the term M_1 which the child state had held, and M_1 is substituted V for the variables p.

The rule (*spec*) is one of the most important, essential rules of our calculus. By this rule, an arbitrary inner state can transit to another state. An important point of this transition is that there are no needs that the context surrounding the state is an evaluation context, which means that, for example, even if a term $\langle \texttt{flet}\ (p\ M)\ S \rangle$ is in a body of a function, S can transit to another state S'. We note that such cases that a flet-term is in a body of a function is only caused when `shift` captures a context surrounded by `flet`.

The rule (*store*) causes a transition of the value in a store. In our calculus, a value can transit only when it contains a flet-term (as described above) and only by the rule (*spec*).

In the following, we call transitions by the rules (*spec*) and (*store*) *speculative transitions*, and the other transitions *mandatory transitions*. The word speculative means that the transition may not need to obtain the final result of its transition sequence.

We represent that a state $\llbracket \theta,\ M \rrbracket$ is `stuck` when M is not a value and there are no rules which can be applied to the state.

In particular, we define that a state is *`stuck` about mandatory transition* or *mandatory* `stuck` for short when there are no mandatory transition rules which can be applied to the state.

We will show an example of transitions in AM-PFSR. In this example, $(\texttt{let}\ ((x\ e_1))\ e_2)$ equals to $((\lambda x.e_2)\ e_1)$. Let the initial state S_0 be $[\![\emptyset,\ \langle M\rangle]\!]$ where M is

$$(\texttt{let}\ ((x\ (\texttt{make}\ 0)))\ (\texttt{let}\ ((y\ (\texttt{future}\ ((\texttt{set!}\ x)\ 10))))\\ (\texttt{let}\ ((z\ ((\texttt{set!}\ x)\ 20)))\ (\texttt{deref}\ x))))$$

This example will show the behaviors of processes which are trying to access to the same box variable.

$$\begin{aligned}
S_0 \rightarrow & [\![\{b_1 \mapsto 0\},\ \langle(\texttt{let}\ ((x\ b_1))\ \ (\texttt{let}\ ((y\ (\texttt{future}\ ((\texttt{set!}\ x)\ 10))))\\
& \qquad (\texttt{let}\ ((z\ ((\texttt{set!}\ x)\ 20)))\ (\texttt{deref}\ x)))))\rangle]\!] && (\textit{make})\\
\rightarrow & [\![\{b_1 \mapsto 0\},\ \langle(\texttt{let}\ ((y\ (\texttt{future}\ ((\texttt{set!}\ b_1)\ 10))))\\
& \qquad (\texttt{let}\ ((z\ ((\texttt{set!}\ b_1)\ 20)))\ (\texttt{deref}\ b_1))))\rangle]\!] && (\textit{app})\\
\rightarrow & [\![\{b_1 \mapsto 0\},\ \langle\texttt{flet}\ (p\ ((\texttt{set!}\ b_1)\ 10))\\
& \qquad [\![\emptyset,\ \langle(\texttt{let}\ ((y\ p))\ (\texttt{let}\ ((z\ ((\texttt{set!}\ b_1)\ 20)))\ (\texttt{deref}\ b_1))))\rangle]\!]\rangle]\!] && (\textit{fork})\\
\rightarrow & [\![\{b_1 \mapsto 0\},\ \langle\texttt{flet}\ (p\ ((\texttt{set!}\ b_1)\ 10))\\
& \qquad [\![\emptyset,\ \langle(\texttt{let}\ ((z\ ((\texttt{set!}\ b_1)\ 20)))\ (\texttt{deref}\ b_1)))\rangle]\!]\rangle]\!] && (\textit{spec})
\end{aligned}$$

In the above, the last transition is caused for the term in the `flet` so the applied rule is (*spec*). In the last state, while a term $((\texttt{set!}\ b_1)\ 20)$ is in the inner state, the term $((\texttt{set!}\ b_1)\ 10)$ is definitely the next target of transition because the store in the inner state does not have the mutable cell b_1 but the outer store has that.

$$\begin{aligned}
\rightarrow & [\![\{b_1 \mapsto 10\},\ \langle\texttt{flet}\ (p\ \textbf{void})\\
& \qquad [\![\emptyset,\ \langle(\texttt{let}\ ((z\ ((\texttt{set!}\ b_1)\ 20)))\ (\texttt{deref}\ b_1)))\rangle]\!]\rangle]\!] && (\textit{set})\\
\rightarrow & [\![\{b_1 \mapsto 10\},\ \langle(\texttt{let}\ ((z\ ((\texttt{set!}\ b_1)\ 20)))\ (\texttt{deref}\ b_1)))\rangle]\!] && (\textit{join})\\
\rightarrow & [\![\{b_1 \mapsto 20\},\ \langle(\texttt{let}\ ((z\ \textbf{void}))\ (\texttt{deref}\ b_1)))\rangle]\!] && (\textit{set})\\
\rightarrow & [\![\{b_1 \mapsto 20\},\ \langle(\texttt{deref}\ b_1)\rangle]\!] && (\textit{app})\\
\rightarrow & [\![\{b_1 \mapsto 20\},\ \langle 20\rangle]\!] \rightarrow [\![\{b_1 \mapsto 20\},\ 20]\!] && (\textit{deref}), (\textit{rv})
\end{aligned}$$

We designed the transition rules in Figure 5.3 and Figure 5.4 to be transparent. Actually, the rules work correctly as showed in the above example which shows that if there are more than one states and they tries to access the same box variable simultaneously, the result is equal to the one of a program obtained by removing all `futures` from the original one.

In AM-PFSR, the following property is satisfied.

Lemma 5.1. *For any term M of AM-PFSR, exactly one of the following clauses holds:*

(1) $M = V$,

(2) $M = E[(V_1\ V_2)]$ *(where V_1 is not* `set!`*)*,

(3) $M = E[\langle V\rangle]$,

(4) $M = E[(\mathcal{S}k.M_1)]$,

(5) $M = E[(\texttt{future}\ M_1)]$,

(6) $M = E[\langle\texttt{flet}\ (p\ V)\ S\rangle]$.

Moreover, the decomposition is unique, namely, the evaluation context E in the cases (2) - (6) is uniquely determined. □

Proof . *Proof by induction on the structure of M. In each case, the uniqueness of decomposition of its evaluation context will also be shown.*

$M \equiv (M_1\ M_2)$

By I.H., M_1 and M_2 are in one of the form (1) to (6).

M_1 **is in the form (1)**

Let M_1 be V_1. In this case, we need more case analysis on M_2.

M_2 **is in the form (1)**

Let M_2 be V_2. If M_1 is `set!`*, M is in the form (1). If M_1 is another value (one of $(\lambda x.M_3), a, b, p,$* ***void****,* `make`*,* `deref` *or $(\texttt{set!}\ V_3)$), M is in the form (2). In the latter, the evaluation context is $\Box$ which means that the uniqueness of evaluation context is recognized.*

M_2 **is in one of the form (2) to (6)**

Let M_2 be $E[R]$ where R is in one of the following: $(V_1\ V_2)$, $\langle V_1 \rangle$, $(\mathcal{S}k.M_3)$, $(\texttt{future}\ M_3)$ or $\langle \texttt{flet}\ (p\ V_1)\ S \rangle$. Then we can write $M = (V_1\ E[R])$. Here $(V_1\ E)$ is also an evaluation context so M can be put as $E'[R]$ where $E' = (V_1\ E)$. R is an arbitrary one of the above forms so we can say M is in the form (2) to (6). Moreover, we can find that E' is unique because E is unique by I.H..

M_1 **is in one of the form (2) to (6)**

We can prove as in the case above. Let M_1 be $E[R]$, then we can write $M = (E[R]\ M_2)$. $(E\ M_2)$ is an evaluation context so M can be put as $E'[R]$ where $E = (E\ M_2)$. Then we can find that M is in one of the form (2) to (6), and E' is unique.

$M \equiv \langle M_1 \rangle$

By I.H., M_1 is in one of the form (1) to (6).

M_1 **is in the form (1)**

In this case, M is clearly in the form (3).

M_1 **is in one of the form (2) to (6)**

M_1 can be put as $E[R]$ (where R is defined as in the first case). The context $\langle E \rangle$ is also an evaluation context so we can find that the term M is in one of the form of (2) to (6). Moreover, uniqueness of $\langle E \rangle$ is also clear.

$M \equiv (\mathcal{S}k.M_1)$

In this case, M is clearly in the form (4). The evaluation context is a hole ($\Box$) so it is also clearly unique.

$M \equiv (\textbf{future}\ M_1)$

In this case, M is clearly in the form (5). The evaluation context is a hole ($\Box$) so it is also clearly unique.

$M \equiv \langle \textbf{flet}\ (p\ M_1)\ S \rangle$

By I.H., M_1 is in one of the form (1) to (6).

M_1 **is in the form (1)**

In this case, M is in the form (6), and its evaluation context is unique ($\Box$).

M_1 **is in one of the form (2) to (6)**

By I.H., we can put M_1 as $E[R]$ with some E and R (R is defined in the case of $M \equiv (M_1\ M_2)$) so $M \equiv \langle \texttt{flet}\ (p\ E[R])\ S \rangle$. The context $\langle \texttt{flet}\ (p\ E)\ S \rangle$ is an evaluation context and it is unique (because E is unique). From the definition of R, we can find M is in one of the form (2) to (6).

■

Lemma 5.1 shows that if a term M is not a value, it can be decomposed to a pair of a unique evaluation context and a redex. However, if a term is in one of the form (2) to (6), it does not equal to the fact that the term can be a target of a transition. For instance, a term $(p\ V)$ is in the form (2) but it cannot be a target of (*app*) transition until some value is substituted for the communication variable p.

By the fact that an arbitrary term can be decomposed uniquely, we can find that there is at most one transition rule which can be applied to a state.

5.3.4 Evaluation Function of AM-PFSR

In Figure 5.5, we define three functions:

- *Unload* is the unloading function from a value to an answer,
- $eval_p$ is the evaluation function of AM-PFSR which get a term and return an answer, and

$\mathit{Unload} : \mathbb{V} \to \mathbb{A}$

$$
\begin{array}{rlrl}
\mathit{Unload}(c) & = c & \mathit{Unload}((\lambda x.M)) & = \textbf{procedure} \\
\mathit{Unload}(b) & = \textbf{box} & \mathit{Unload}((\texttt{set!}\ V)) & = \textbf{procedure} \\
\mathit{Unload}(p) & = \textbf{error} & \mathit{Unload}(x) & = \textbf{error}
\end{array}
$$

$\mathit{eval}_p : \mathbb{P} \to \text{Power}(\mathbb{A} \cup \{\bot\})$

$$
\begin{aligned}
\mathit{eval}_p(P) \ = & \{\mathit{Unload}(V) \mid [\![\emptyset,\ P]\!] \to^* [\![\theta,\ V]\!]\} \\
& \cup \begin{cases} \{\textbf{error}\} & \text{if there are any mandatory } \texttt{stuck}\text{transition sequences starting from } [\![\emptyset,\ P]\!] \\ \{\} & \text{otherwise} \end{cases} \\
& \cup \begin{cases} \{\bot\} & \text{there are any infinite transition sequences starting from } [\![\emptyset,\ P]\!] \\ & \text{containing infinitely many mandatory transitions} \\ \{\} & \text{otherwise} \end{cases}
\end{aligned}
$$

$\mathit{eval}_s : \mathbb{P} \to \text{Power}(\mathbb{A} \cup \{\bot\})$

$$
\mathit{eval}_s(P) \ = \begin{cases} \{\mathit{Unload}(V) \mid [\![\emptyset,\ P]\!] \to^* [\![\theta,\ V]\!]\} & \\ \{\textbf{error}\} & \text{if it is a } \texttt{stuck}\text{transition sequence} \\ \{\bot\} & \text{if it is an infinitely transition sequence starting from } [\![\emptyset,\ P]\!] \end{cases}
$$

Figure 5.5: Evaluation Functions

- eval_s is the evaluation function of AM-FSR. (We will describe this function in the next subsection.)

In Figure 5.5 and the following, we assume that $\mathbb{P}$, $\mathbb{V}$ and $\mathbb{A}$ denote the set of programs, values and answers respectively.

The function *Unload* translate a value V to an answer A. In this translation, it removes some details, for example, if it gets a function $(\lambda x.M)$, it always returns a symbol **procedure**.

eval_p is a function which gets a program P and returns a set which is a subset of $\mathbb{A} \cup \{\textbf{error}\} \cup \{\bot\}$. Here $\to^*$ means a reflective and transitive closure of $\to$. The resulting set of a program P is defined as follows: it begins transitions from an initial state $[\![\emptyset,\ P]\!]$ and

- when the state becomes a state like $[\![\theta,\ V]\!]$, the resulting set contains the answer $\mathit{Unload}(V)$,
- when this transition sequence becomes mandatory `stuck`, **error** will be contained in the resulting set, and
- when this transition sequence becomes an infinitely sequence and it contains infinitely many mandatory transitions, the resulting set contains $\bot$. Here, $\bot$ represents an infinite computation.

While the resulting set may contain several answers, the set must contain only an answer, which will be proved in the next section.

An important fact is that $\bot$ is obtained only when the transition sequence contains infinitely many mandatory transitions. This is because a state whose term is a value can cause infinitely many speculative transitions in AM-PFSR. This kind of infinite sequence should not be assumed an actual infinite transition sequence. For example, let Ω be $((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$, a state $[\![\theta,\ (\lambda x.\langle \texttt{flet}\ (p\ 0)\ [\![\theta',\ \Omega]\!]\rangle)]\!]$ can cause infinitely many speculative transitions.

5.3.5 The Definition of AM-FSR

In this subsection, we will define an abstract machine AM-FSR which compute a program completely sequentially. The terms, states and values of AM-FSR are defined almost same as the ones of AM-PFSR respectively. The only difference

is that communication variables (p) and flet-terms does not exist in AM-FSR.

The context C also does not contains flet-terms. The other contexts (evaluation contexts) E, F and G are defined as follows:

$$\begin{aligned} E &::= \Box \mid (E\ M) \mid (V\ E) \mid \langle E \rangle \mid (\texttt{future}\ E) \\ F &::= \langle G \rangle \\ G &::= \Box \mid (G\ M) \mid (V\ G) \mid (\texttt{future}\ G) \end{aligned}$$

AM-FSR has all transition rules in Figure 5.3 and a new rule (*fid*) defined as follows:

$$\langle\!\langle \theta,\ E[(\texttt{future}\ V)] \rangle\!\rangle \to \langle\!\langle \theta,\ E[V] \rangle\!\rangle \qquad (\textit{fid})$$

AM-FSR does not have rules in Figure 5.4.

The differences between AM-PFSR and AM-FSR appear when they evaluate a term like $E[F[(\texttt{future}\ M)]]$. In AM-PFSR, it begins parallel computation; generates a new communication variable p, generates a new state contains a term $F[p]$ and the term turns into a flet-term. On the other hand, in AM-FSR, it begins the evaluation of a term M (because it assumes $(\texttt{future}\ \Box)$ is a kind of evaluation context in AM-FSR), and, when the term M becomes a value V, it removes `future` and begins the evaluation of $E[F[V]]$. This means that `future`s do not have any effects in AM-FSR.

For the sequential machine AM-FSR, we can show the following property same as the one in AM-PFSR.

Lemma 5.2. *For any term M of AM-FSR, exactly one of the following clauses holds:*

(1) $M \equiv V$,

(2) $M \equiv E[(V_1\ V_2)]$ *(where* V_1 *is not* `set!`*),*

(3) $M \equiv E[\langle V \rangle]$,

(4) $M \equiv E[(\mathcal{S}k.M_1)]$,

(5) $M \equiv E[(\texttt{future}\ V)]$.

Moreover, the decomposition is unique, namely, the evaluation context E in the cases (2) - (6) is uniquely determined. □

Proof . *Lemma 5.2 can be proved in the same way as Lemma 5.1: by induction on the structure of M.*

$\underline{M \equiv V}$

In this case, M is clearly in the case (1).

$\underline{M \equiv (M_1\ M_2)}$

By I.H., M_1 and M_2 is in one of the form (1) to (5).

$\underline{M_1}$ **is in the form (1)**

Let M_1 be a value V_1. In this case, we must analyze on the form of M_2.

$\underline{M_2}$ **is in the form (1)**

*When M_1 is in one of the form $(\lambda x.M_3)$, a, x, b, p, **void**, `make`, `deref`, $(\texttt{set!}\ V_3)$, M is in the form (2). When M_1 is `set!`, M is in the form (1). In both cases, the evaluation context is $\Box$ so the uniqueness of the evaluation context is held.*

$\underline{M_2}$ **is in one of the form (2) to (5)**

By I.H., we can put M_2 as $E[R]$ where R is any one of $(V_{21}\ V_{22})$, $\langle V_2 \rangle$, $(\mathcal{S}k.M_3)$ or $(\texttt{future}\ V_2)$, then we can put M as $(V_1\ E[R])$. As R must be in one of the form (2) to (5) and the context $(V_1\ E)$ is an evaluation context, M is in one of the form (2) to (5). And, by the fact that E is unique, we can find that $(V_1\ E)$ is also unique.

$\underline{M_1}$ **is in the form (2) to (5)**

We can put M_1 as $E[R]$ (R is defined as also in the previous clause) so M can be put as $(E[R]\ M_2)$. As R must be in one of the form (2) to (5) and the context $(E\ M_2)$ is an evaluation context, M is in one of the form (2) to (5). And E is unique so $(V_1\ E)$ is also unique.

$M \equiv \langle M_1 \rangle$

By I.H., M_1 is in the form (1) to (5).

M_1 **is in the form (1)**

In this case, M is in the form (3).

M_1 **is in one of the form (2) to (5)**

We can proof this case same as the case that M is in the form $(M_1\ M_2)$. As we can put M_1 as $E[R]$ by I.H. and R is in one of the form (2) to (5), M is also in the form same as the one of R. The uniqueness of the evaluation context $\langle E \rangle$ is also found by the fact that E is unique.

$M \equiv (\mathcal{S}k.M_1)$

In this case, M is in the form (4).

$M \equiv (\texttt{future}\ M_1)$

This case is proved same as the case that M is $\langle M_1 \rangle$.

M_1 **is in the form (1)**

M is clearly in the case (5).

M_1 **is in one of the form (2) to (5)**

As M_1 can be put as $E[R]$ that R is in one of the form (2) to (5), $M = (\texttt{future}\ E[R])$ is in the same case of R. The uniqueness of the evaluation context is also clear.

■

By the above, we have the following theorem.

Theorem 5.1. *In AM-FSR, for any state S_0, there exists at most one state S_1 such that $S_0 \to S_1$.*

Proof . *It can be shown easily by Lemma 5.2 and the definition of the transition rules of AM-FSR.*

The evaluation function $eval_s$ is defined in Figure 5.5, and we can immediately find that the returned result of $eval_s$ is a set containing only one answer.

5.4 Transparency of `future`

The goal of this section is to prove the transparency of `future` stated as follows:

Theorem 5.2 (transparency)**.** *For an arbitrary program P, we have $eval_p(P) = eval_s(P)$.* □

This theorem says that

> the meaning for arbitrary programs in AM-PFSR is deterministic, and the semantics of AM-PFSR is equal to that of AM-FSR.

We have to prove three theorems before proving the main theorem. First is that a resulting set of AM-FSR is always included in that of AM-PFSR. Second is confluence of the transitions of AM-PFSR, which means that if an arbitrary state of AM-PFSR S_1 can transit to two different states S_2 and S_3, there is a state S_4 to which S_2 and S_3 can transit. The last is the uniqueness of the results of AM-PFSR, which means the resulting set of $eval_p(P)$ always contains only an answer.

5.4.1 Relationship between Sequential and Parallel Transition

In this subsection, we will prove the inclusion of the results of AM-FSR to that of AM-PFSR by showing theirs simulation.

Theorem 5.3. *For any program P, we have $eval_s(P) \subseteq eval_p(P)$.* □

To prove the above, we define new notations. In this subsection, we will write $(-)^s$ for an arbitrary element of AM-FSR and $(-)^p$ for that of AM-PFSR. For example, E^s means an evaluation context define in AM-FSR, and C^p is a context defined in AM-PFSR. We note that we cannot say E^s always can be E^p because E^s may contain an evaluation context $(\texttt{future}\ \square)$ while $(\texttt{future}\ \square)$ is not that in AM-PFSR

We will call an evaluation context which is surrounding by `reset` at its outermost a *program-evaluation context*.

We will define a binary relation $X^s \sim Y^s$ where X^s is one of state, term, store or context. This relation is define mutually inductively with the relations $F^s \approx_f C^p$ and $E^s \approx_e C^s$.

Def 5.1 ($X^s \sim Y^p$)**.** *Let X^s and Y^p be a term or a context, then*

$$\frac{(d = \square, x, b, c)}{d \sim d} \qquad \frac{X^s \sim Y^p}{(\lambda x.X^s) \sim (\lambda x.Y^p)} \qquad \frac{X_1^s \sim Y_1^p \quad X_2^s \sim Y_2^p}{(X_1^s\ X_2^s) \sim (Y_1^p\ Y_2^p)}$$

$$\frac{F^s \approx_f C^p \quad X^s \sim Y^p}{F^s[X^s] \sim C^p[Y^p]}\ (*) \qquad \frac{X^s \sim Y^p}{(\mathcal{S}k.X^s) \sim (\mathcal{S}k.Y^p)} \qquad \frac{X^s \sim Y^p}{(\texttt{future}\ X^s) \sim (\texttt{future}\ Y^p)}$$

The rule (*) is essential one in the proofs in this subsection. As described latter, when $M^s \sim M^p$, $\langle M^s \rangle \sim \langle M^p \rangle$ because we have $\langle \square \rangle \approx_f \langle \square \rangle$. Therefore, for any term M^s in AM-FSR, we have that $M^s \sim M^s$.

Def 5.2 ($X^s \sim Y^p$)**.** *Let X^s be a state or a store, then*

$$\frac{\theta^s \sim \theta^p \quad M^s \sim M^p}{[\![\theta^s,\ M^s]\!] \sim [\![\theta^p,\ M^p]\!]} \qquad \frac{V_i^s \sim V_i^p\ \ (\textit{for } i = 1, \cdots, n)}{\{b_1 \mapsto V_1^s, \cdots, b_n \mapsto V_n^s\} \sim \{b_1 \mapsto V_1^p, \cdots, b_n \mapsto V_n^p\}}$$

Next we will define a binary relation $\approx_f$. In this definition, we will use another definition of F^s:

$$F^s\ ::=\ \langle \square \rangle \mid F^s[(\square\ M^s)] \mid F^s[(V^s\ \square)] \mid F^s[(\texttt{future}\ \square)]$$

The reason is that, for several lemmas, this make their proof easy. It is easy to show that the two definitions are the same.

Def 5.3 ($F^s \approx_f C^p$)**.**

$$\frac{}{\langle \square \rangle \approx_f \langle \square \rangle} \qquad \frac{F^s \approx_f C^p}{F^s[(\texttt{future}\ \square)] \approx_f \langle \texttt{flet}\ (p\ \square)\ [\![\emptyset,\ C^p[p]]\!] \rangle\ \textit{(p fresh)}}$$

$$\frac{F^s \approx_f C^p \quad M^s \sim M^p}{F^s[(\square\ M^s)] \approx_f C^p[(\square\ M^p)]} \qquad \frac{F^s \approx_f C^p \quad V^s \sim V^p}{F^s[(V^s\ \square)] \approx_f C^p[(V^p\ \square)]}$$

Def 5.4 ($E^s \approx_e C^p$)**.**

$$\frac{G^s \sim C^p}{G^s \approx_e C^p} \qquad \frac{E^s \approx_e C_1^p \quad F^s \approx_f C_2^p}{E^s[F^s] \approx_e C_1^p[C_2^p]}$$

An intuitive meaning of $F^s \approx_f C^p$ is that we can obtain C^p by applying the (*fork*) rule to all `futures` in F^s. For example, when $F^s \equiv \langle((\texttt{future}\ ((\texttt{future}\ \square)\ x))\ y)\rangle$, we have $F^s \approx_f \langle \texttt{flet}\ (p_2\ \square)\ \langle \texttt{flet}\ (p_1\ (p_2\ x))\ \langle (p_1\ y) \rangle \rangle \rangle$.

An important point is that $F^s \sim C^p$ does not always imply that C^p is an pure evaluation context F^p since we have $\langle(\texttt{future}\ \square)\rangle \sim \langle(\texttt{future}\ \square)\rangle$. In our definition of AM-PFSR, $\langle(\texttt{future}\ \square)\rangle$ is not a pure evaluation context. If we have $F^s \approx_f C^p$, it implies that C^p is a pure evaluation context of AM-PFSR. This will be shown in Lemma 5.3.

Lemma 5.3. *We have the following properties about $\sim$ and $\approx_f$.*

(1) If $V^s \sim Y^p$ then Y^p is a value of AM-PFSR,

(2) If $M^s \sim Y^p$ then Y^p is a term of AM-PFSR,

(3) If $C^s \sim Y^p$ then Y^p is a context of AM-PFSR,

(4) If $S^s \sim Y^p$ then Y^p is a state of AM-PFSR,

(5) If $\theta^s \sim Y^p$ then Y^p is a store of AM-PFSR,

(6) If $F^s \approx_f C^p$ then C^p is a pure evaluation context of AM-PFSR,

(7) If $E^s \approx_e C^p$ and E^s is a program-evaluation context then C^p is an evaluation context of AM-PFSR. □

Proof . *We prove the clauses (1) to (7) by simultaneous induction on the derivations for the relations $\sim$, $\approx_f$ and $\approx_e$.*

(1) By the case analysis on the last rule used in the derivation of $V^s \sim Y^p$.

$\underline{V^s \equiv c, x, b \textbf{ and } Y^p \equiv V^s}$
Clearly Y^p is a value of AM-PFSR.

$\underline{V^s \equiv (\lambda x.M_1^s), Y^p \equiv (\lambda x.Y_1^p), M_1^s \sim Y_1^p}$
By I.H. on (2), Y_1^p is a term of AM-PFSR, which implies that Y^p is a value of AM-PFSR.

$\underline{V^s \equiv (\texttt{set!}\ V_2^s), Y^p \equiv (Y_1^p\ Y_2^p), \texttt{set!} \sim Y_1^p \textbf{ and } V_2^s \sim Y_2^p}$
From the definition of $\sim$, Y_1^p is `set!`, and by I.H. on (1), Y_2^p is a value of AM-PFSR, hence Y^p is a value in AM-PFSR.

(2) By the case analysis on the last rule used in the derivation of $M^s \sim Y^p$.

$\underline{M^s \textbf{ is a value of AM-FSR}}$
In these cases, Y^p is also a value of AM-PFSR which is proved in the clause (1) above, and a value is a kind of terms, hence Y^p is a term of AM-PFSR.

$\underline{M^s \equiv (M_1^s\ M_2^s), Y^p \equiv (Y_1^p\ Y_2^p), M_1^s \sim Y_1^p \textbf{ and } M_2^s \sim Y_2^p}$
By I.H. on (2), Y_1^p and Y_2^p are terms of AM-PFSR. Then, by the definition of terms of AM-PFSR, Y^p is a term of AM-PFSR.

$\underline{M^s \equiv F^s[M_1^s], Y^p \equiv C^p[Y_1^p], F^s \approx_f C^p \textbf{ and } M_1^s \sim Y_1^p}$
By I.H. on (2), Y_1^p is a term of AM-PFSR, and by I.H. on (6), C^p is a pure evaluation context of AM-PFSR. Then Y^p is a term of AM-PFSR.

$\underline{M^s \equiv (\mathcal{S}k.M_1^s), Y^p \equiv (\mathcal{S}k.Y_1^p) \textbf{ and } M_1^s \sim Y_1^p}$
By I.H. on (2), Y_1^p is a term of AM-PFSR. Then, by the definition of terms, Y^p is a term of AM-PFSR.

$\underline{M^s \equiv (\texttt{future}\ M_1^s), Y^p \equiv (\texttt{future}\ Y_1^p), M_1^s \sim Y_1^p}$
By I.H. on (2), Y_1^p is a term of AM-PFSR. Then, by the definition of terms, Y^p is a term of AM-PFSR.

(3) By the case analysis on the last rule used in the derivation of $C^s \sim Y^p$.

$\underline{C^s \equiv \Box \textbf{ and } Y^p \equiv C^s}$
Clearly Y^p is a context (a hole $\Box$) of AM-PFSR.

$\underline{C^s \equiv (\lambda x.C_1^s), Y^p \equiv (\lambda x.Y_1^p) \textbf{ and } C_1^s \sim Y_1^p}$
By I.H. on (3), Y^p is a context of AM-PFSR. Then $(\lambda x.Y_1^p)$ is also a context of AM-PFSR.

$\underline{C^s \equiv (C_1^s\ M_2^s), Y^p \equiv (Y_1^p\ Y_2^p), C_1^s \sim Y_1^p \textbf{ and } M_2^s \sim Y_2^p}$
By I.H. on (3), Y_1^p is a context of AM-PFSR, and by I.H. on (2), Y_2^p is a term of AM-PFSR. Then, by the definition of contexts, Y^p is also a context of AM-PFSR.

$\underline{C^s \equiv (M_1^s\ C_2^s), Y^p \equiv (Y_1^p\ Y_2^p), M_1^s \sim Y_1^p \textbf{ and } C_2^s \sim Y_2^p}$
By I.H. on (2), Y_1^p is a term of AM-PFSR, and by I.H. on (3), Y_2^p is a context of AM-PFSR. Then, by the definition of contexts, Y^p is also a context of AM-PFSR.

$\underline{C^s \equiv F_1^s[C_2^s], Y^p \equiv Y_1^p[Y_2^p], F_1^s \approx_f Y_1^p \textbf{ and } C_2^s \sim Y_2^p}$
By I.H. on (6), Y_1^p is a pure evaluation context of AM-PFSR, and by I.H. on (3), Y_2^p is a context of AM-PFSR. Then, by the definition of contexts, Y^p is also a context of AM-PFSR.

$\underline{C^s \equiv (\mathcal{S}k.C_1^s), Y^p \equiv (\mathcal{S}k.Y_1^p) \textbf{ and } C_1^s \sim Y_1^p}$
By I.H. on (3), Y_1^p is a context of AM-PFSR. Then, by the definition of contexts, Y^p is also a context of AM-PFSR.

$C^s \equiv (\texttt{future}\ C_1^s)$, $Y^p \equiv (\texttt{future}\ Y_1^p)$ **and** $C_1^s \sim Y_1^p$

By I.H. on (3), Y_1^p is a context of AM-PFSR. Then, by definition of contexts, Y^p is also a context of AM-PFSR.

(4) Let S^s be $[\![\theta^s,\ M^s]\!]$. Then there exist Y_1^p and Y_2^p such that $Y^p \equiv [\![Y_1^p,\ Y_2^p]\!]$, $\theta^s \sim Y^p$ and $M^s \sim Y_2^p$. By I.H. on (4), Y_1^p is a store of AM-PFSR and by I.H. on (2), Y_2^p is a term of AM-PFSR. By the definition of states of AM-PFSR, Y^p is a state of AM-PFSR.

(5) Let θ^s be $\{(b_1, V_1^s), ..., (b_n, V_n^s)\}$. By $\theta^s \sim \theta^p$, there exists $Y_i^p (1 \leq i \leq n)$ such that $\theta^p \equiv \{(b_1, Y_1^p), ..., (b_n, Y_n^p)\}$. Then, by $\theta^s \sim \theta^p$, $V_i^s \sim Y_i^p$ By I.H. on (1), each Y_i^p is a value of AM-PFSR. By the definition of the store of AM-PFSR, we have θ^p is a store of AM-PFSR.

(6) By the case analysis on the last rule used in the derivation of $F^s \approx_f Y^p$.

$F^s \equiv \langle\Box\rangle$, $Y^p \equiv \langle\Box\rangle$

Clearly Y^p is a pure evaluation context of AM-PFSR.

$F^s \equiv F_1^s(\Box\ M_2^s)$, $C^p \equiv Y_1^p(\Box\ Y_2^p)$, $F_1^s \approx_f Y_1^p$, $M_2^s \sim Y_2^p$

By I.H. on (6), Y_1^p is a pure evaluation context, and by I.H. on (2), Y_2^p is a term of AM-PFSR. Then C^p is a pure evaluation context of AM-PFSR.

$F^s \equiv F_1^s(V_2^s\ \Box)$, $C^p \equiv Y_1^p(Y_2^p\ \Box)$, $F_1^s \approx_f Y_1^p$, $V_2^s \sim Y_2^p$

By I.H. on (6), Y_1^p is a pure evaluation context, and by I.H. on (1), Y_2^p is a value of AM-PFSR. Then C^p is a pure evaluation context of AM-PFSR.

$F^s \equiv F_1^s(\texttt{future}\ \Box)$, $C^p \equiv \langle\texttt{flet}\ (p\ \Box)\ [\![\emptyset,\ Y_1^p[p]]\!]\rangle$, $F_1^s \approx_f Y_1^p$ **and** p **is fresh**

By I.H. on (6), Y_1^p is a pure evaluation context. Then C^p is a pure evaluation context of AM-PFSR.

(7) By the assumption that E^s is a program-evaluation context, the last rule used to derive $E^s \approx_e C^p$ is the second one of the definition of $\approx_e$. Therefore, there exist a program-evaluation context E_1^s and F_1^s such that $E^s \equiv E_1^s[F_1^s]$. Then, by the fact that $E^s \approx_e E^s$, there exist C_1^p and C_2^p such that $C^p \equiv C_1^p[C_2^p]$, $E_1^s \approx_e C_1^p$ and $F_1^s \approx_f C_2^p$. By I.H. on (7), C_1^p is an evaluation context and, by I.H. on (6), C_2^p is a pure evaluation context, which implies that C^p is an evaluation context. ■

Lemma 5.4. *(1) If $X^s \sim Y^p$ and $V^s \sim V^p$ then $X^s\{x := V^s\} \sim Y^p\{x := V^p\}$.*

(2) If $F^s \approx_f C^p$ and $V^s \sim V^p$ then $F^s\{x := V^s\} \approx_f C^p\{x := V^p\}$.

(3) We assume that X^s denotes one of terms or contexts. If $C^s \sim C^p$ and $X^s \sim X^p$ then $C^s[X^s] \sim C^p[X^p]$.

(4) If $F^s \approx_f C^p$ then $F^s \sim C^p$.

(5) If E^s is a program-evaluation context and $E^s \approx_e C^p$ then $E^s \sim C^p$. □

Proof . *(1) Proof by induction on the structure of X^s.*

$X^s \equiv c, b, \Box$

We will show only the case of $X^s = c$. The left-hand side is $X^s\{x := V^s\} = c$. By the assumption, $Y^p = c$, and then $Y^p\{x := V^p\} = c$. Other cases can be shown in the same manner.

$X^s \equiv x$

$x\{x := V^s\}$ equals to V^s. By the assumption, Y^p equals to x. Then $Y^p\{x := V^p\}$ equals to V^p.

$X^s \equiv y, y \neq x$

By the definition of substitution, $y\{x := V^s\}$ equals to y. By the assumption, Y^p equals to y. Then $Y^p\{x := V^p\}$ equals to V^p.

$X^s \equiv (\lambda x.X_1^s)$

By the definition of substitution, $(\lambda x.X_1^s)\{x := V^s\} = (\lambda x.X_1^s)$. And also, by definition of $\sim$, there exists Y_1^p such that $Y^p \equiv (\lambda x.Y_1^p)$ and $X_1^s \sim Y_1^p$. Since $Y^p\{x := V^p\} = (\lambda x.Y_1^p)$, we have $X^s\{x := V^s\} \sim Y^p\{x := V^p\}$.

$\underline{X^s \equiv (\lambda y.X_1^s), y \neq x}$

We can assume that y is not free in V^s. By the definition of substitution, $X^s\{x := V^s\} = (\lambda y.X_1^s)\{x := V^s\} = (\lambda y.X_1^s\{x := V^s\})$. By the assumption, there exists Y_1^p such that $Y^p \equiv (\lambda y.Y_1^p)$ and $X_1^s \sim Y_1^p$. Since we have the fact that $X_1^s \sim Y_1^p$ and $V^s \sim V^p$, $X_1^s\{x := V^s\} \sim Y_1^p\{x := V^p\}$ by I.H.. Therefore, by the definition of $\sim$, we have $X^s\{x := V^s\} \sim Y^p\{x := V^p\}$.

$\underline{X^s \equiv (X_1^s\ X_2^s)}$

By the definition of substitution, $X^s\{x := V^s\} = (X_1^s\{x := V^s\}\ X_2^s\{x := V^s\})$. By the assumption, there exist Y_1^p and Y_2^p such that $Y^p \equiv (Y_1^p\ Y_2^p)$, $X_1^s \sim Y_1^p$ and $X_2^s \sim Y_2^p$. Also, by the assumption, we have $X_1^s\{x := V^s\} \sim Y_1^p\{x := V^p\}$ and $X_2^s\{x := V^s\} \sim Y_2^p\{x := V^p\}$. By the definition of $\sim$, we have $(X_1^s\{x := V^s\}\ X_2^s\{x := V^s\}) \sim (Y_1^p\{x := V^p\}\ Y_2^p\{x := V^p\})$.

$\underline{X^s \equiv \langle X_1^s \rangle}$

By the definition of $\sim$, there exist F^s, C^p, X_1^s and Y_1^p such that $X^s \equiv F^s[X_1^s]$, $Y^p \equiv C^p[Y_1^p]$, $F^s \approx_f C^p$ and $X_1^s \sim Y_1^p$. By the definition of substitution,

$$X^s\{x := V^s\} \equiv F^s[X_1^s]\{x := V^s\} = F^s\{x := V^s\}[X_1^s\{x := V^s\}]$$

and

$$Y^p\{x := V^p\} \equiv C^p[Y_1^s]\{x := V^p\} = C^p\{x := V^p\}[X_1^p\{x := V^p\}]$$

By I.H. of (2), $F^s\{x := V^s\} \approx_f C^p\{x := V^p\}$, and by I.H. of (1), $X_1^s\{x := V^s\} \approx_f X_1^p\{x := V^p\}$. Then, by the definition of $\sim$,

$$F^s\{x := V^s\}[X_1^s\{x := V^s\}] \approx_f C^p\{x := V^p\}[X_1^p\{x := V^p\}]$$

$\underline{X^s \equiv (\mathcal{S}k.X_1^s), k = x}$

By the definition of substitution, $(\mathcal{S}k.X_1^s)\{x := V^s\} = (\mathcal{S}k.X_1^s)$. By the definition of $\sim$, there exists Y_1^p such that $Y^p \equiv (\mathcal{S}k.Y_1^p)$ and $X_1^s \sim Y_1^p$. Since $Y^p\{x := V^p\} = (\mathcal{S}k.x)Y_1^p$, we have $(\mathcal{S}k.X_1^s)\{x := V^s\} \sim (\mathcal{S}k.Y^p)\{x := V^p\}$.

$\underline{X^s \equiv (\mathcal{S}k.X_1^s), k \neq x}$

We can assume that y is not free in V^s. By the definition of substitution, $X^s\{x := V^s\} = (\mathcal{S}k.X_1^s)\{x := V^s\} = (\mathcal{S}k.X_1^s\{x := V^s\})$. By the assumption, there exists Y_1^p such that $Y^p \equiv (\mathcal{S}k.Y_1^p)$ and $X_1^s \sim Y_1^p$. Since we have that $X_1^s \sim Y_1^p$ and $V^s \sim V^p$, we have $X_1^s\{x := V^s\} \sim Y_1^p\{x := V^p\}$ by I.H.. Therefore, by definition of $\sim$, we have $(\mathcal{S}k.X_1^s)\{x := V^s\} \sim (\mathcal{S}k.Y_1^p)\{x := V^p\}$.

$\underline{X^s \equiv (\mathbf{future}\ X_1^s)}$

By the definition of substitution, $(\texttt{future}\ X_1^s)\{x := V^s\} = (\texttt{future}\ X_1^s\{x := V^s\})$. By the definition of $\sim$, there exists Y_1^p such that $Y^p \equiv (\texttt{future}\ Y_1^p)$ and $X_1^s \sim Y_1^p$. By I.H., $X_1^s\{x := V^s\} = Y_1^p\{x := V^p\}$. By the definition of $\sim$, we have $(\texttt{future}\ X_1^s\{x := V^s\}) \sim (\texttt{future}\ Y_1^p\{x := V^p\})$.

(2) By induction on the structure of F^s. Since this proof is almost the same as above, we show only the case $F^s \equiv F_1^s[(\texttt{future}\ \Box)]$.

By the assumption, there exists C_1^p such that $C^p \equiv \langle \texttt{flet}\ (p\ \Box)\ [\![\emptyset,\ C_1^p[p]]\!] \rangle$ and $F_1^s \approx_f C_1^p$ (p is fresh). We have

$$F^s\{x := V^s\} \equiv F_1^s\{x := V^s\}[(\texttt{future}\ \Box)]$$

and

$$C^p\{x := V^p\} \equiv \langle \texttt{flet}\ (p\ \Box)\ [\![\emptyset,\ C_1^p\{x := V^p\}[p]]\!] \rangle$$

By I.H., we have $F_1^s\{x := V^s\} \approx_f C_1^p\{x := V^p\}$, which implies $F^s \approx_f C^p$.

(3) Proof by induction on the structure of C.

$\underline{C^s \equiv \Box}$

By the definition of $\sim$, $C^p = \Box$. Then we have $C^s[X^s] = X^s \sim X^p = C^p[X^p]$.

$\underline{C^s \equiv (\lambda x.C_1^s)}$

By the definition of $\sim$, there exists some C_1^p such that $C^p \equiv (\lambda x.C_1^p)$ and $C_1^s \sim C_1^p$. By I.H., we have $C_1^s[X^s] \sim C_1^p[X^p]$. Then, by the definition of $\sim$, $(\lambda x.C_1^s[X^s]) \sim (\lambda x.C_1^p[X^p])$.

$\underline{C^s \equiv (C_1^s\ M_1^s)}$

By the definition of $\sim$, there exists some C_1^p and M_1^p such that $C^p \equiv (C_1^p\ M_1^p)$, $C_1^s \sim C_1^p$ and $M_1^s \sim M_1^p$. By I.H., we have $C_1^s[X^s] \sim C_1^p[X^p]$. Then, by the definition of $\sim$, $(C_1^s[X^s]\ M_1^s) \sim (C_1^p[X^p]\ M_1^p)$.

$\underline{C^s \equiv (M_1^s\ C_1^s)}$

By the definition of $\sim$, there exists some M_1^p and C_1^p such that $C^p \equiv (M_1^p\ C_1^p)$, $M_1^s \sim M_1^p$ and $C_1^s \sim C_1^p$. By I.H., we have $C_1^s[X^s] \sim C_1^p[X^p]$. Then, by the definition of $\sim$, $(M_1^s\ C_1^s[X^s]) \sim (M_1^p\ C_1^p[X^p])$.

$\underline{C^s \equiv \langle C_1^s \rangle}$

By the definition of $\sim$, there exists some F_1^s, C_1^s, C_1^p and C_2^p such that $C^s \equiv F_1^s[C_1^s]$, $C^p \equiv C_1^p[C_2^p]$, $F_1^s \approx_f C_1^p$ and $C_1^s \sim C_2^p$. By I.H., we have $C_1^s[X^s] \sim C_2^p[X^p]$. Then, by the definition of $\sim$, $F_1^s[C_1^s[X^s]] \sim C_1^p[C_2^p[X^p]]$.

$\underline{C^s \equiv (\mathcal{S}k.C_1^s)}$

By the definition of $\sim$, there exists some C_1^p such that $C^p \equiv (\mathcal{S}k.C_1^p)$ and $C_1^s \sim C_1^p$. By I.H., we have $C_1^s[X^s] \sim C_1^p[X^p]$. Then, by the definition of $\sim$, $(\mathcal{S}k.C_1^s[X^s]) \sim (\mathcal{S}k.C_1^p[X^p])$.

$\underline{C^s \equiv (\mathtt{future}\ C_1^s)}$

By the definition of $\sim$, there exists some C_1^p such that $C^p \equiv (\mathtt{future}\ C_1^p)$ and $C_1^s \sim C_1^p$. By I.H., we have $C_1^s[X^s] \sim C_1^p[X^p]$. Then, by the definition of $\sim$, $(\mathtt{future}\ C_1^s[X^s]) \sim (\mathtt{future}\ C_1^p[X^p])$.

(4) We have $F^s \approx_f C^p$ by the assumption. Then, if we assume $M^s \equiv M^p \equiv \Box$, we can derive $F^s \sim C^p$ immediately by the rule (*) which was defined in the definition of binary relation $\sim$.

(5) We will show by the induction on the structure of E^s. Since we have that E^s is a program-evaluation context by the hypothesis, we only have to show the case that $E^s \equiv E_1^s[F^s]$.

$\underline{E^s \equiv E_1^s[F^s]}$

By the definition of $\sim$, there exist some C_1^s and C_2^s such that $C^p \equiv C_1^p[C_2^p]$, $E_1^s \approx_e C_1^p$ and $F^s \approx_f C_2^p$. By I.H., $E_1^s \sim C_1^p$. By the clause (3) of this lemma, we also have $F^s \sim C_2^s$. By the clause (2) of this lemma, we have $E^s \equiv E_1^s[F^s] \sim C_1^p[C_2^p] \equiv C^p$. ■

In the following lemma ant its proof, an extended term (an e-term for short) is either term or a context.

Lemma 5.5 (Decomposition)**.** *(1) Let X^s be an e-term and E^s be a program-evaluation context. If $E^s[\langle X^s \rangle] \sim Y^p$ then there exist a context C^p and an e-term Z^p such that $Y^p \equiv C^p[\langle Z^p \rangle]$, $E^s \sim C^p$ and $X^s \sim Z^p$.*

(2) Let M^s be a value or a term in the form of $(V_1^s\ V_2^s)$. If $F^s[M^s] \sim N^p$ then there exist a pure evaluation context F^p and a term M^p such that $N^p \equiv F^p[M^p]$, $F^s \sim F^p$ and $M^s \sim M^p$.

(3) Let X^s be an e-term. If $G^s[X^s] \sim Y^p$ then there exist a context C^p and an e-term Z^p such that $Y^p \equiv C^p[Z^p]$, $G^s \sim C^p$ and $X^s \sim Z^p$. □

Proof . *(1) By induction on the program-evaluation context E^s where we use the alternative definition: $E^s ::= F^s \mid F^s[E^s]$.*

$\underline{E^s \equiv F^s}$

By $F^s[\langle X^s \rangle] \sim Y^p$ and the definition of $\sim$, there exist a pure evaluation context F_1^s, a context G_2^s, a context C_1^p and an e-term Y_1^p such that $F^s \equiv F_1^s[G_2^s]$, $Y^p \equiv C_1^p[Y_1^p]$, $F_1^s \approx_f C_1^p$ and $G_2^s[\langle X^s \rangle] \sim Y_1^p$.

From $G_2^s[\langle X^s \rangle] \sim Y_1^p$ and I.H. on (3), there exist a context C_2^p and an e-term Y_2^p such that $Y_1^p \equiv C_2^p[Y_2^p]$, $G_2^s \sim C_2^p$ and $\langle X^s \rangle \sim Y_2^p$.

From $F_1^s \approx_f C_1^p$ and $G_2^s \sim C_2^p$, we have $F_1^s[G_2^s] \sim C_1^p[C_2^p]$ using the rule (*), which means $E^s \sim C_1^p[C_2^p]$.

From $\langle X^s \rangle \sim Y_2^p$, there exists an e-term Z^p such that $X^s \sim Z^p$ and $Y_2^p \equiv \langle Z^p \rangle$.

Putting $C^p \equiv C_1^p[C_2^p]$, we have $E^s \approx_f C^p$, $X^s \sim Z^p$ and $Y^p \equiv C^p[\langle Z^p \rangle]$.

$\underline{E^s \equiv F_1^s[E_1^s]}$

By the same argument as the previous case, there exist a context C_0^p and an e-term Y_1^p such that $Y^p \equiv C_0^p[Y_1^p]$, $F_1^s \sim C_0^p$ and $E_1^s[\langle X^s \rangle] \sim Y_1^p$.

By I.H., there exist a context C_1^p and an e-term Z^p such that $Y_1^p \equiv C_1^p[\langle Z^p \rangle]$, $E_1^s \sim C_1^p$ and $X^s \sim Z^p$.

Putting $C^p \equiv C_0^p[C_1^p]$, this case is proved.

(2) By induction on the pure evaluation context F^s where we use the following alternative definition:

$$F^s \ ::= \ \langle\Box\rangle \mid F^s[(\Box\ M^s)] \mid F^s[(V^s\ \Box)] \mid F^s[(\texttt{future}\ \Box)].$$

We shall prove the case when $M^s \equiv (V_1^s\ V_2^s)$ only. By the definition of $\sim$, either one of the three cases holds.

$N^p \equiv F^p[M^p]$, $F^s \approx_f F^p$ **and** $M^s \sim M^p$
We immediately have the conclusion.

$N^p \equiv F_1^p[M_1^p]$, $F^s[(V_1^s\ \Box)] \approx_f F_1^p$ **and** $V_2^s \sim M_1^p$
We can prove, by induction on F^s, that there exists a pure evaluation context F^p and a term M_2^p such that $F_1^p \equiv F^p[(M_2^p\ \Box)]$, $F^s \approx_f F^p$ and $V_1^s \sim M_2^p$. We put $M^p \equiv (M_2^p\ M_1^p)$, then F^p and M^p satisfy the desired properties.

$N^p \equiv F_1^p[M_1^p]$, $F^s[(\Box\ V_2^s)] \approx_f F_1^p$ **and** $V_1^s \sim M_1^p$
This case is proved similarly to the above case.

(3) By the induction on the structure of G^s.

$G^s \equiv \Box$
By the assumption, we immediately find that $C^p \equiv \Box$ and $Z^p \equiv Y^p$.

$G^s \equiv (G_1^s\ M_1^s)$
$G^s[X^s] \equiv (G_1^s[X^s]\ M^s) \sim Y^p$. By the definition of $\sim$, there exist an e-term Z_1^p and a term Z_2^p such that $Y^p \equiv (Z_1^p\ Z_2^p)$, $G_1^s[X^s] \sim Z_1^p$ and $M^s \sim Z_2^p$. Then, by I.H., there exist a context C_1^p and an e-term Z_3^p such that $Z_1^p \equiv C_1^p[Z_3^p]$, $G_1^s \sim C_1^p$ and $X^s \sim Z_3^p$. Putting $C^p \equiv (C_1^p\ Z_2^p)$ and $Z^p \equiv Z_3^p$, we are done.

$G^s \equiv (V_1^s\ G_1^s)$
$G^s[X^s] \equiv (V_1^s\ G_1^s[M^s]) \sim Y^p$. By the definition of $\sim$, there exist a term Z_1^p and an e-term Z_2^p such that $Y^p \equiv (Z_1^p\ Z_2^p)$, $V_1^s \sim Z_1^p$ and $G_1^s[X^s] \sim Z_2^p$. Then, by I.H., there exist a context C_1^p and an e-term Z_3^p such that $Z_2^p \equiv C_1^p[Z_3^p]$, $G_1^s \sim C_1^p$ and $X^s \sim Z_3^p$. Putting $C^p \equiv (Z_1^p\ C_1^p)$ and $Z^p \equiv Z_3^p$, we are done.

$G^s \equiv (\texttt{future}\ G_1^s)$
$G^s[X^s] \equiv (\texttt{future}\ G_1^s[X^s]) \sim Y^p$. By the definition of $\sim$, there exist an e-term Z_1^p such that $Y^p \equiv (\texttt{future}\ Z_1^p)$ and $G_1^s[X^s] \sim Z_1^p$. Then, by I.H., there exist a context C_1^p and an e-term Z_2^p such that $Z_1^p \equiv C_1^p[Z_2^p]$, $G_1^s \sim C_1^p$ and $X^s \sim Z_2^p$. Putting $C^p \equiv (\texttt{future}\ C_1^p)$ and $Z^p \equiv Z_2^p$, we are done. ■

Here, we will define a new binary relation $\rightarrow_f^*$ which denotes transitions of contexts.

Def 5.5. *For any context C_i^p $(i = 1, 2)$ of AM-PFSR, we define $C_1^p \rightarrow_f^* C_2^p$ if and only if, for any term M^p, any store θ^p and any evaluation context E^p, $letrd\theta^p E^{@}[C_1^p[M^p]] \rightarrow^* \{\!|\theta^p,\ E^p[C_2^p[M^p]]|\!\}$ by applying only the $\texttt{fork}$ rule to the state.*

The next lemma (Lemma 5.6) is a key of this subsection.

Lemma 5.6. *(1) If $F^s \approx_f F^p$ and $G^s \sim C^p$, then there exists some F_1^p such that $F^s[G^s] \approx_f F_1^p$ and $F^p[C^p] \rightarrow_f^* F_1^p$.*

(2) If $F^s \sim C^p$ then there exists some F^p such that $F^s \approx_f F^p$ and $C^p \rightarrow_f^* F^p$.

(3) Let E^s be a program-evaluation context. If $E^s \sim C^p$ then there exists some E^p such that $E^s \approx_e E^p$ and $C^p \rightarrow_f^* E^p$. □

Proof . *(1) We first give an alternative inductive definition to a truly pure evaluation context G^s as follows:*

$$G^s \ ::= \ \Box \mid G^s[(\Box\ M^s)] \mid G^s[(V^s \Box\ ])\ \mid G^s[(\texttt{future}\ \Box)].$$

The equivalence of the original and alternative definitions can be shown easily.

$G^s \equiv \Box$
By the assumption $\Box \sim C^p$, we have $C^p \equiv \Box$. Then, $F^s[G^s] \equiv F^s \approx_f F^p \equiv F^p[C^p]$, and we only have to let $F_1^p \equiv F^p$.

$\underline{G^s \equiv G_1^s[(\texttt{future}\ \Box)]}$

By Lemma 5.5 (3), there exists some C_1^p such that $C^p \equiv C_1^p[(\texttt{future}\ \Box)]$. By I.H., there exists F_2^p such that $F^s[G_1^s] \approx_f F_2^p$ and $F^p[C_1^p] \to_f^* F_2^p$. If $F_1^p \equiv \langle \texttt{flet}\ (p\ \Box)\ \{\!|\emptyset,\ F_2^p[p]|\!\}\rangle$, we have $F^s[G^s] \equiv F^s[G_1^s[(\texttt{future}\ \Box)]] \approx_f F_2^p$ by the definition of $\approx_f$. Then, we have the following sequence of transition:

$$F^p[C^p] \equiv F^p[C_1^p[(\texttt{future}\ \Box)]] \to_f^* F_1^p[(\texttt{future}\ \Box)] \to_f \langle \texttt{flet}\ (p\ \Box)\ \{\!|\emptyset,\ F_1^p[p]|\!\}\rangle \equiv F_2^p.$$

$\underline{G^s \equiv G_1^s[(\Box\ M^s)]}$

By Lemma 5.5 (3), there exist some C_1^p and M^p such that $C^p \equiv C_1^p[(\Box\ M^p)]$, $G_1^s \sim C_1^p$ and $M^s \sim M^p$. By I.H., there exists F_2^p such that $F^s[G_1^s] \approx_f F_2^p$ and $F^p[C_1^p] \to_f^* F_2^p$. If $F_1^p \equiv F_2^p[(\Box\ M^p)]$, $F^s[G^s] \equiv F^s[G_1^s[(\Box\ M^s)]] \approx_f F_1^p$ by the definition of $\approx_f$. Then we have the following sequence of transitions:

$$F^p[C^p] \equiv F^p[C_1^p[(\Box\ M^p)]] \to_f^* F_2^p[(\Box\ M^p)] \equiv F_1^p.$$

$\underline{G^s \equiv G_1^s[(V^s\ \Box)]}$

By Lemma 5.5 (3), there exist some C_1^p and V^p such that $C^p \equiv C_1^p[(V^p\ \Box)]$, $G_1^s \sim C_1^p$ and $V^s \sim V^p$. By I.H., there exists F_2^p such that $F^s[G_1^s] \approx_f F_2^p$ and $F^p[C_1^p] \to_f^* F_2^p$. If $F_1^p \equiv F_2^p[(V^p\ \Box)]$, then $F^s[G^s] \equiv F^s[G_1^s[(V^s\ \Box)]] \approx_f F_1^p$ by the definition of $\approx_f$. Then we have the following sequence of transitions:

$$F^p[C^p] \equiv F^p[C_1^p[(V^p\ \Box)]] \to_f^* F_2^p[(V^p\ \Box)] \equiv F_1^p.$$

(2) Since the rule which is used to derive $F^s \sim C^p$ is the rule (*), there exist some F_1^s, G^s, F_1^p, C_1^p such that $F^s \equiv F^s[G^s]$, $C^p \equiv F_1^p[C_1^p]$, $F_1^s \approx_f F_1^p$ and $G^s \sim C_2^p$. Then, by the above lemma, we have some F^p such that $F_1^s[G_1^s] \approx_f F^p$ and $F_1^p[C_1^p] \to_f^* F^p$.

(3) We use the alternative inductive definition for program-evaluation contexts as $E^s ::= F^s \mid E^s[F^s]$. Then we prove this case by induction on the structure of program-evaluation context.

$\underline{E^s \equiv F^s}$

Since $E^s \equiv F^s \sim C^p$, there exists some F^p such that $F^s \approx_f F^p$ and $C^p \to_f^* F^p$ by (2).

$\underline{E^s \equiv E_1^s[F^s]}$

By Lemma 5.5, there exist some C_1^p and C_2^p such that $C^p \equiv C_1^p[C_2^p]$, $E_1^s \sim C_1^p$ and $F^s \sim C_2^p$. By the fact that $E_1^s \sim C_1^p$ and I.H., there exists some E_1^p such that $E_1^s \approx_e E_1^p$ and $C_1^p \to_f^* E_1^p$. Further, by the fact that $F^s \sim C_2^p$ and (2), there exists some F_1^p such that $F^s \approx_f F_1^p$ and $C_2^p \to_f^* F_1^p$. Here, E^s can be put as $E_1^p[F_1^p]$. Then, by the definition of $\approx_e$, $E^s \equiv E_1^s[F^s] \approx_e E_1^p[F_1^p] \equiv E^p$. Also we have that $C^p \equiv C_1^p[C_2^p] \to_f^* E_1^p[C_2^p] \to_f^* E_1^p[F_1^p] \equiv E^p$. ■

Theorem 5.4 (Simulation)**.** *Let S_1^s be a program state. If $S_1^s \sim S_1^p$ and $S_1^s \to S_2^s$, there exists S_2^p such that $S_2^s \sim S_2^p$ and $S_1^p \to^* S_2^p$. Moreover, the sequence $S_1^p \to^* S_2^p$ contains one or more mandatory transitions.* □

Proof . *By case analysis on the transition rule applied to $S_1^s \to S_2^s$.*

(app)

By the assumption, $S_1^s \equiv \{\!|\theta^s,\ E^s[((\lambda x.M^s)\ V^s)]|\!\}$ and $S_2^s \equiv \{\!|\theta^s,\ E^s[M^s\{x := V^s\}]|\!\}$. By Lemma 5.5, there exist θ^p, C^p, M^p, V^p such that $S_1^p \equiv \{\!|\theta^p,\ C^p[((\lambda x.M^p)\ V^p)]|\!\}$, $\theta^s \sim \theta^p$, $E^s \sim C^p$, $M^s \sim M^p$ and $V^s \sim V^p$. By Lemma 5.6, there exists E^p such that $E^s \approx_e E^p$ and $C^p \to_f^* E^p$. Since E^p is an evaluation context, $\{\!|\theta^p,\ E^p[((\lambda x.M^p)\ V^p)]|\!\} \to \{\!|\theta^p,\ E^p[M^p\{x := V^p\}]|\!\}$ by (app). If $S_2^p \equiv \{\!|\theta^p,\ E^p[M^p\{x := V^p\}]|\!\}$, we have $S_1^p \to^* S_2^p$, which contains a mandatory transition. And, by Lemma 5.4, $\{\!|\theta^s,\ E^s[M^s\{x := V^s\}]|\!\} \sim S_2^p$.

(rv)

By the assumption, $S_1^s \equiv \{\!|\theta^s,\ E^s[\langle V^s\rangle]|\!\}$ and $S_2^s \equiv \{\!|\theta^s,\ E^s[V^s]|\!\}$. By the fact that $S_1^s \sim S_1^p$, there exist some θ^p and N^p such that $S_1^p \equiv \{\!|\theta^p,\ N^p|\!\}$, $\theta^s \sim \theta^p$ and $E^s[\langle V^s\rangle] \sim N^p$. By Lemma 5.5 and $E^s[\langle V^s\rangle] \sim N^p$, there exist some C^p and V^p such that $S_1^p \equiv \{\!|\theta_p,\ C^p[\langle V^p\rangle]|\!\}$, $E^s \sim C^p$ and $V^s \sim V^p$. By Lemma 5.6, there exist some E^p such that $E^s \approx_e E^p$ and $C^p \to_f^* E^p$. By the definition of $\to_f^*$, we have $S_1^p \to^* \{\!|\theta^p,\ E^p[\langle V^p\rangle]|\!\}$. Since E^p is an evaluation context, $\{\!|\theta^p,\ E^p[\langle V^p\rangle]|\!\} \to \{\!|\theta^p,\ E^p[V^p]|\!\}$. Then, by the fact that S_2^p is assumed to $\{\!|\theta^p,\ E^p[V^p]|\!\}$, we have that $S_1^p \to^* S_2^p$ and this sequence contains a mandatory transition. Furthermore, by the definition of $\sim$, $S_2^s \sim S_2^p$.

(*rs*)

By the assumption, $S_1^s \equiv \langle\!\langle \theta^s,\ E^s[F^s[(\mathcal{S}k.M^s)]]\rangle\!\rangle$ and $S_2^s \equiv \langle\!\langle \theta^s,\ E^s[M^s\{k := (\lambda v.F^s[v])\}]\rangle\!\rangle$. By the definition of $\sim$ and Lemma 5.5, there exist some θ^p, C_1^p, C_2^p, M^p such that $S_1^p \equiv \langle\!\langle \theta^p,\ C_1^p[C_2^p[(\mathcal{S}k.M^p)]]\rangle\!\rangle$, $E^s \sim C_1^p$, $F^s \sim C_2^p$ and $M^s \sim M^p$. By Lemma 5.6, there exist E^p and F^p such that $E^s \approx_e E^p$, $C_1^p \to_f^* E^p$, $F^s \approx_f F^p$ and $C_2^p \to_f^* F^p$. By the definition of $\to_f^*$, $S_1^p \to^* \langle\!\langle \theta^p,\ E^p[F^p[(\mathcal{S}k.M^p)]]\rangle\!\rangle$. If S_2^p is assumed to $\langle\!\langle \theta^p,\ E^p[M^p\{k := (\lambda v.F^p[v])\}]\rangle\!\rangle$, $\langle\!\langle \theta^p,\ E^p[F^p[(\mathcal{S}k.M^p)]]\rangle\!\rangle \to S_2^p$. And this shows that the transition sequence $S_1^p \to^* S_2^p$ contains a mandatory transition.

(*fid*)

By the assumption, $S_1^s \equiv \langle\!\langle \theta^s,\ E^s[(\texttt{future}\ V^s)]\rangle\!\rangle$ and $S_2^s \equiv \langle\!\langle \theta^s,\ E^s[V^s]\rangle\!\rangle$. Since S_1^s is a program state, E^s is a program-evaluation context and we can put it as $E^s \equiv E_1^s[F^s]$ with some E_1^s and F^s. Then there exist some θ^p, C_1^p, C_2^p and V^p such that $S_1^p \equiv \langle\!\langle \theta^p,\ C_1^p[C_2^p[V^p]]\rangle\!\rangle$, $\theta^s \sim \theta^p$, $E_1^s \sim C_1^p$, $F^s[(\texttt{future}\ \square)] \sim C_2^p$ and $V^s \sim V^p$. By Lemma 5.6, there exists some E_1^p such that $E_1^s \approx_e E_1^p$ and $C_1^p \to_f^* E_1^p$. Here, we will divide this case into two cases based on the derivation of the relation $F^s[(\texttt{future}\ \square)] \sim C_2^p$. Namely, we have two subcases depending on whether $F^s[(\texttt{future}\ \square)] \approx_f C_2^p$.

$F^s[(\textbf{future}\ \square)] \approx_f C_2^p$

By the definition of $\approx_f$, there exists F^p such that $C_2^p \equiv \langle \texttt{flet}\ (p\ \square)\ F^p[p]\rangle$ (where p is a fresh communication variable) and $F^s \approx_f F^p$. If we assume that S_2^p is $\langle\!\langle \theta^p,\ E_1^p[F^p[V^p]]\rangle\!\rangle$, $S_2^s \sim S_2^p$. Further, we have the following:

$$\begin{aligned} S_1^p &\equiv \langle\!\langle \theta^p,\ C_1^p[\langle \texttt{flet}\ (p\ V^p)\ F^p[p]\rangle]\rangle\!\rangle \\ &\to^* \langle\!\langle \theta^p,\ E_1^p[\langle \texttt{flet}\ (p\ V^p)\ F^p[p]\rangle]\rangle\!\rangle \\ &\to \langle\!\langle \theta^p,\ E_1^p[F^p[V^p]]\rangle\!\rangle \equiv S_2^p \end{aligned}$$

the other case

In this case, we can put $F^s \equiv F_1^s[G^s]$ and $C_2^p \equiv F^p[C_3^p]$ with some F_1^s, G^s, F^p and C_3^p such that $F_1^s \approx_f F^p$, $G^s[(\texttt{future}\ \square)] \sim C_3^p$. By applying Lemma 5.5 to $G^s[(\texttt{future}\ \square)] \sim C_3^p$, there exists C_4^p such that $C_3^p \equiv C_r^p[(\texttt{future}\ \square)]$ and $G^s \sim C_4^p$. And also, by applying Lemma 5.6 (1) to $F_1^s \approx_f F^p$ and $G^s \sim C_4^p$, there exists F_2^p that $F_1^s[G^s] \approx_f F_2^p$ and $F^p[C_4^p] \to_f^* F_2^p$. By letting S_2^p be $\langle\!\langle \theta^p,\ E_1^p[F_2^p[V^p]]\rangle\!\rangle$, we have $S_2^s \sim S_2^p$, then we have the following:

$$\begin{aligned} S_1^p &\equiv \langle\!\langle \theta^p,\ C_1^p[F^p[C_4^p[(\texttt{future}\ V^p)]]]\rangle\!\rangle \to^* \langle\!\langle \theta^p,\ E_1^p[F^p[C_4^p[(\texttt{future}\ V^p)]]]\rangle\!\rangle \\ &\to^* \langle\!\langle \theta^p,\ E_1^p[F_2^p[(\texttt{future}\ V^p)]]\rangle\!\rangle \to \langle\!\langle \theta^p,\ E_1^p[\langle \texttt{flet}\ (p\ V^p)\ \langle\!\langle \emptyset,\ F_2^p[p]\rangle\!\rangle\rangle]\rangle\!\rangle \\ &\to \langle\!\langle \theta^p,\ E_1^p[F_2^p[V^p]]\rangle\!\rangle \equiv S_2^p \end{aligned}$$

The transitions contain a mandatory transition (fork).

(*make*)

By the assumption, $S_1^s \equiv \langle\!\langle \theta^s,\ E^s[(\texttt{make}\ V^s)]\rangle\!\rangle$ and $S_2^s \equiv \langle\!\langle \theta^s \cup \{b \mapsto V^s\},\ E^s[b]\rangle\!\rangle$ where b is a fresh variable. By Lemma 5.5, there exist some θ^p, C^p and V^p such that $S_1^p \equiv \langle\!\langle \theta^p,\ C^p[(\texttt{make}\ V^p)]\rangle\!\rangle$, $\theta^s \sim \theta^p$, $E^s \sim C^p$ and $V^s \sim V^p$. By Lemma 5.6, there exists some E^p such that $E^s \approx_e E^p$ and $C^p \to_f^* E^p$. By the definition of $\to_f^*$, $S_1^p \to \langle\!\langle \theta^p,\ E^p[(\texttt{make}\ V^p)]\rangle\!\rangle$. Since E^p is an evaluation context, $\langle\!\langle \theta^p,\ E^p[(\texttt{make}\ V^p)]\rangle\!\rangle \to \langle\!\langle \theta^p \cup \{b \mapsto V^p\},\ E^p[b]\rangle\!\rangle$. Here, we have $\theta^s \cup \{b \mapsto V^s\} \sim \theta^p \cup \{b \mapsto V^p\}$ because of the facts that $\theta^s \sim \theta^p$ and $V^s \sim V^p$. Since S_2^p can be assumed to the right-hand side of the last transition, we have $S_1^p \to^* S_2^p$ and $S_2^s \sim S_2^p$.

(*set*)

By the assumption, $S_1^s \equiv \langle\!\langle \theta^s \cup \{b \mapsto V_1^s\},\ E^s[((\texttt{set!}\ b)\ V^s)]\rangle\!\rangle$, $S_2^s \equiv \langle\!\langle \theta^s \cup \{b \mapsto V^s\},\ E^s[\textbf{\textit{void}}]\rangle\!\rangle$. By Lemma 5.5, there exist θ^p, V_1^p, C^p and V^p such that $S_1^p \equiv \langle\!\langle \theta^p \cup \{b \mapsto V_1^p\},\ C^p[((\texttt{set!}\ b)\ V^p)]\rangle\!\rangle$, $\theta^s \sim \theta^p$, $V_1^s \sim V_1^p$, $E^s \sim C^p$ and $V^s \sim V^p$. By Lemma 5.6, there exists some E^p such that $E^s \approx_e E^p$ and $C^p \to_f^* E^p$. Since S_2^p can be assumed that $\langle\!\langle \theta^p \cup \{b \mapsto V^p\},\ E^p[\textbf{\textit{void}}]\rangle\!\rangle$, we have that $S_2^s \sim S_2^p$ and the following transitions:

$$\begin{aligned} S_1^p &\equiv \langle\!\langle \theta^p \cup \{b \mapsto V_1^p\},\ C^p[((\texttt{set!}\ b)\ V^p)]\rangle\!\rangle \\ &\to^* \langle\!\langle \theta^p \cup \{b \mapsto V_1^p\},\ E^p[((\texttt{set!}\ b)\ V^p)]\rangle\!\rangle \\ &\to \langle\!\langle \theta^p \cup \{b \mapsto V^p\},\ E^p[\textbf{\textit{void}}]\rangle\!\rangle \equiv S_2^p \end{aligned}$$

(deref)

By the assumption, $S_1^s \equiv [\![\theta^s \cup \{b \mapsto V^s\},\ E^s[(\texttt{deref}\ b)]]\!]$, $S_2^s \equiv [\![\theta^s \cup \{b \mapsto V^s\},\ E^s[V^s]]\!]$. *By Lemma 5.5, there exist* θ^p, C^p *and* V^p *such that* $S_1^p \equiv [\![\theta^p \cup \{b \mapsto V^p\},\ C^p[(\texttt{deref}\ b)]]\!]$, $\theta^s \sim \theta^p$, $E^s \sim C^p$ *and* $V^s \sim V^p$. *By Lemma 5.6, there exists some* E^p *such that* $E^s \approx_e E^p$ *and* $C^p \to_f^* E^p$. *Since* S_2^p *can be assumed that* $[\![\theta^p \cup \{b \mapsto V^p\},\ E^p[V^p]]\!]$, *we have that* $S_2^s \sim S_2^p$ *and the following transitions:*

$$\begin{aligned} S_1^p &\equiv [\![\theta^p \cup \{b \mapsto V^p\},\ C^p[(\texttt{deref}\ b)]]\!] \\ &\to^* [\![\theta^p \cup \{b \mapsto V^p\},\ E^p[(\texttt{deref}\ b)]]\!] \\ &\to [\![\theta^p \cup \{b \mapsto V^p\},\ E^p[V^p]]\!] \equiv S_2^p \end{aligned}$$

■

At the end of this subsection, we will prove the theorem 5.3.

Proof . *Let* P *be an arbitrary program of AM-FSR.*

eval$_s(P) \neq \{\bot\}, \{\textbf{error}\}$

In the AM-FSR, we have $[\![\emptyset,\ P]\!] \to^* [\![\theta^s,\ V^s]\!]$. *By Theorem 5.4, there exist some* θ^p *and* V^p *such that* $[\![\emptyset,\ P]\!] \to^* [\![\theta^p,\ V^p]\!]$ *and* $V^s \sim V^p$. *Since* $V^s \sim V^p$ *derives the fact that Unload*$(V^s) =$ *Unload*(V^p), *we have* $eval_s(P) \subseteq eval_p(P)$.

eval$_s(P) = \{\textbf{error}\}$

Since $[\![\emptyset,\ P]\!] \to^* S_1^s$, *the state* S_1^s *is a* `stuck` *state. By Theorem 5.4, there exists some* S_1^p *such that* $[\![\emptyset,\ P]\!] \to^* S_1^p$ *and* $S_1^s \sim S_1^p$. *By the fact that the program state* S_1^s *is* `stuck` *and Lemma 5.2, there exist some* θ^s, E^s, V_1^s *and* V_2^s *such that* $S_1^s \equiv [\![\theta^s,\ E^s[(V_1^s\ V_2^s)]]\!]$. *By* $S_1^s \sim S_1^p$ *and Lemma 5.5, there exist* θ^p, C^p, V_1^p *and* V_2^p *such that* $S_1^p \equiv [\![\theta^p,\ C^p[(V_1^p\ V_2^p)]]\!]$, $\theta^s \sim \theta^p$, $E^s \sim C^p$, $V_1^s \sim V_1^p$ *and* $V_2^s \sim V_2^p$. *By Lemma 5.6, there exists some* E^p *such that* $C^p \to_f^* E^p$. *Let* S_2^p *be* $[\![\theta^p,\ E^p[(V_1^p\ V_2^p)]]\!]$, *then* $S_1^p \to^* S_2^p$.

We will prove that S_2^p *is a* `stuck` *state, by the case analysis on the cause that* S_1^s *is a* `stuck` *state. We prove only the case when* V_1^s *is a constant value other than* `make`, `set!` *or* `deref`. *By* $V_1^s \sim V_1^p$, V_1^p *is the same constant, and by Lemma 5.1, there exist no transitions starting with* S_2^p, *and consequently* S_2^p *is a* `stuck` *state. Other cases can be shown in a similar way. Finally, we have* $eval_s(P) = \{\textbf{\textit{error}}\} \subseteq eval_p(P)$.

eval$_s(P) = \{\bot\}$

There exists an infinite transition sequence starting with $[\![\emptyset,\ P]\!]$ *in AM-FSR. By Theorem 5.4, there exists an infinite transition sequence in AM-PFSR which starts with* $[\![\emptyset,\ P]\!]$ *and contains infinitely many mandatory transitions. Then* $eval_s(P) \subseteq eval_p(P)$. ■

5.4.2 Confluence of Transitions in AM-PFSR

The confluence of transitions is a key property to prove the transparency of our calculus, which is stated as follows:

Theorem 5.5 (Confluence)**.** *For any states* S_1, S_2 *and* S_3 *in AM-PFSR, if* $S_1 \to^* S_2$ *and* $S_1 \to^* S_3$ *then there exists some state* S_4 *such that* $S_2 \to^* S_4$ *and* $S_3 \to^* S_4$. □

In order to prove Theorem 5.5, we will extend the parallel reduction method. The original parallel reduction is proposed by Takahashi, and we extend the parallel reduction to our parallel transition, and $S_1 \Rightarrow S_2$ denotes our notion of the parallel transition. Also we will define a maximum parallel transition $(-)^*$.

Then we will prove a so called *diamond property* for the extended parallel reductions and maximum parallel reductions. In the following, we will simply call our extended parallel transition parallel transition.

Intuitively, $S_1 \Rightarrow S_2$ means that if the state S_1 has some redexes, namely, subterms which can be reduced, we can obtain S_2 by reducing zero or more redexes simultaneously. For example, let M_1 be $((\lambda x.x)\ 3)$ and M_2 be $((\lambda x.x)\ 5)$, and S_1, S_2 and S_3 are as follows.

$$\begin{aligned} S_1 &= [\![\emptyset,\ \langle \texttt{flet}\ (p\ M_1)\ [\![\emptyset,\ \langle \texttt{flet}\ (q\ M_2)\ [\![\emptyset,\ \langle (+\ p\ q)\rangle]\!]\rangle]\!]\rangle]\!] \\ S_2 &= [\![\emptyset,\ \langle \texttt{flet}\ (p\ 3)\ [\![\emptyset,\ \langle \texttt{flet}\ (q\ 5)\ [\![\emptyset,\ \langle (+\ p\ q)\rangle]\!]\rangle]\!]\rangle]\!] \\ S_3 &= [\![\emptyset,\ (+\ 3\ 5)]\!] \end{aligned}$$

At this time, we have that $S_1 \Rightarrow S_2$ and $S_2 \Rightarrow S_3$.

In the parallel transition $S_2 \Rightarrow S_3$, the two flet-terms are reduced simultaneously, namely, M_1 and M_2 are reduced in the same step. We note that we cannot reduce arbitrary redexes using our definitions; a subterm R of flet-terms can be reduced only if the whole term can be decomposed into the form $E[R]$. For example, we *cannot* have the following transition:

$$[\![\emptyset,\ (+\ M_1\ M_2)]\!] \Rightarrow [\![\emptyset,\ (+\ 3\ 5)]\!]$$

5.4.2.1 The Definition of Parallel Transitions

In this subsection, we will define three binary relations; parallel transition $\Rightarrow$ and its auxiliary transition $\Rightarrow_r$ and $\Rightarrow_s$. $\Rightarrow$ is defined on states in AM-PFSR, $\Rightarrow_r$ is defined on specific terms in AM-PFSR, and $\Rightarrow_s$ is define on terms, evaluation contexts and stores in AM-PFSR.

Def 5.6 ($\Rightarrow$: Parallel transition rules for states)**.**

$$\frac{R \Rightarrow_r R'\ \ E \Rightarrow_s E'\ \ \theta \Rightarrow_s \theta'}{[\![\theta,\ E[R]]\!] \Rightarrow [\![\theta',\ E'[R']]\!]}\ (\textit{Redex}) \qquad \frac{V \Rightarrow_s V'\ \ E \Rightarrow_s E'\ \ \theta \Rightarrow_s \theta'\ \ (b\ \textit{fresh})}{[\![\theta,\ E[(\texttt{make}\ V)]]\!] \Rightarrow [\![\theta' \cup \{b \mapsto V'\},\ E'[b]]\!]}\ (\textit{Make})$$

$$\frac{V \Rightarrow_s V'\ \ E \Rightarrow_s E'\ \ \theta \Rightarrow_s \theta'}{[\![\theta \cup \{b \mapsto V_1\},\ E[((\texttt{set!}\ b)\ V)]]\!] \Rightarrow [\![\theta' \cup \{b \mapsto V'\},\ E'[\textbf{\textit{void}}]]\!]}\ (\textit{Set})$$

$$\frac{V \Rightarrow_s V'\ \ E \Rightarrow_s E'\ \ \theta \Rightarrow_s \theta'}{[\![\theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]]\!] \Rightarrow [\![\theta' \cup \{b \mapsto V'\},\ E'[V']]\!]}\ (\textit{Deref})$$

$$\frac{\theta \Rightarrow_s \theta'\ \ E \Rightarrow_s E'\ \ V \Rightarrow_s V'\ \ [\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]\ \ \textit{Dom}(\theta') \cap \textit{Dom}(\theta_2) = \emptyset}{[\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!] \rangle]]\!] \Rightarrow [\![\theta' \cup (\theta_2\{p := V'\}),\ E'[M_2\{p := V'\}]]\!]}\ (\textit{Join})$$

$$\frac{M \Rightarrow_s M'\ \ \theta \Rightarrow_s \theta'}{[\![\theta,\ M]\!] \Rightarrow [\![\theta',\ M']\!]}\ (\textit{Spec})$$

Def 5.7 ($\Rightarrow_r$: Auxiliary parallel transition rules for specific terms)**.**

$$\frac{M \Rightarrow_s M'\ \ V \Rightarrow_s V'}{((\lambda x.M)\ V) \Rightarrow_r M'\{x := V'\}}\ (\textit{App})$$

$$\frac{M \Rightarrow_s M'\ \ F \Rightarrow_s F'\ \ (v\ \textit{fresh})}{F[(\mathcal{S}k.M)] \Rightarrow_r \langle M'\{k := (\lambda v.F'[v])\}\rangle}\ (\textit{RtSt}) \qquad \frac{V \Rightarrow_s V'}{\langle V \rangle \Rightarrow_r V'}\ (\textit{RtVl})$$

$$\frac{M \Rightarrow_s M'\ \ F \Rightarrow_s F'\ \ (p\ \textit{fresh})}{F[(\texttt{future}\ M)] \Rightarrow_r \langle \texttt{flet}\ (p\ M')\ [\![\emptyset,\ F'[p]]\!] \rangle}\ (\textit{Fork})$$

Def 5.8 ($\Rightarrow_s$: Auxiliary parallel transition rules for terms, evaluation contexts)**.**

$$\frac{v = x, c, b, p, \Box}{v \Rightarrow_s v}\ (\textit{S-Val}) \qquad \frac{M \Rightarrow_s M'}{(\lambda x.M) \Rightarrow_s (\lambda x.M')}\ (\textit{S-Lam})$$

$$\frac{M_1 \Rightarrow_s M_1'\ \ M_2 \Rightarrow_s M_2'}{(M_1\ M_2) \Rightarrow_s (M_1'\ M_2')}\ (\textit{S-App}) \qquad \frac{M \Rightarrow_s M'}{\langle M \rangle \Rightarrow_s \langle M' \rangle}\ (\textit{S-Rt})$$

$$\frac{M \Rightarrow_s M'}{(\mathcal{S}k.M) \Rightarrow_s (\mathcal{S}k.M')}\ (\textit{S-St}) \qquad \frac{M \Rightarrow_s M'}{(\texttt{future}\ M) \Rightarrow_s (\texttt{future}\ M')}\ (\textit{S-Ft})$$

$$\frac{M \Rightarrow_s M'\ \ S \Rightarrow S'}{\langle \texttt{flet}\ (p\ M)\ S \rangle \Rightarrow_s \langle \texttt{flet}\ (p\ M')\ S' \rangle}\ (\textit{S-Flet})$$

The binary relation $\Rightarrow_s$ is overloaded for terms and evaluation contexts; when the left-hand side is a term, so is the right-hand side, and $\Rightarrow_s$ becomes a binary relation over terms. When the left-hand side is an evaluation context (in particular, it contains one hole), so is the right-hand side, and it becomes a binary relation over evaluation contexts. While the definition allows other expressions (such as the one with two holes), we will not use (and be concerned with) such cases.

Note that the binary relation $\Rightarrow_s$ allows actual transitions occurring in flet-terms only, as indicated by the second assumption $S \Rightarrow S'$ of the rule (*S-Flet*).

Def 5.9 ($\Rightarrow_s$: Auxiliary parallel transition rules for stores)**.**

$$\frac{V_i \Rightarrow_s V_i' \ \ (1 \leq i \leq n)}{\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\} \Rightarrow_s \{b_1 \mapsto V_1', \cdots, b_n \mapsto V_n'\}} \ (\textit{S-Store})$$

If $[\![\theta,\ M]\!] \Rightarrow [\![\theta',\ M']\!]$ and the last rule which is used to derive the relation is other than (*Spec*), the transition is called *mandatory* and otherwise called *speculative*.

5.4.2.2 The Definition of Maximum Parallel Transitions

In this subsection, we will define three functions: a maximum parallel transition $(-)^\star$ on states of AM-PFSR, an auxiliary parallel transition $(-)^\#$ on terms, evaluation contexts and stores of AM-PFSR, and another auxiliary parallel transition $(-)^\circ$ on specific terms of AM-PFSR.

Wile several states or terms can be matched to one or more clauses in the definition of maximum parallel transition, we define that the upper clause is chosen preferentially.

Def 5.10 ($(-)^\star$: Maximum parallel reduction for states)**.**

$$\begin{aligned}
[\![\theta,\ E[R]]\!]^\star &= [\![\theta^\#,\ E^\#[R^\circ]]\!] \textit{ (if } R^\circ \textit{ is defined for the } R) \\
[\![\theta,\ E[(\texttt{make}\ V)]]\!]^\star &= [\![\theta^\# \cup \{b \mapsto V^\#\},\ E^\#[b]]\!] \textit{ (where } b \textit{ fresh)} \\
[\![\theta \cup \{b \mapsto V_1\},\ E[((\texttt{set!}\ b)\ V)]]\!]^\star &= [\![\theta^\# \cup \{b \mapsto V^\#\},\ E^\#[\textbf{\textit{void}}]]\!] \\
[\![\theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]]\!]^\star &= [\![\theta^\# \cup \{b \mapsto V^\#\},\ E^\#[V^\#]]\!] \\
[\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!]\rangle]]\!]^\star &= [\![\theta^\# \cup (\theta_2\{p := V^\#\}),\ E^\#[M_2\{p := V^\#\}]]\!] \\
[\![\theta_2,\ M_2]\!] = [\![\theta_1,\ M_1]\!]^\star &\textit{ and } \mathit{Dom}(\theta^\#) \cap \mathit{Dom}(\theta_2) = \emptyset \\
[\![\theta,\ M]\!]^\star &= [\![\theta^\#,\ M^\#]\!] \textit{ (otherwise)}
\end{aligned}$$

We note that we used $(-)^\star$ for the inner state at the definition of `flet` term.

Def 5.11 ($(-)^\circ$: Auxiliary parallel transition for specific terms)**.**

$$\begin{aligned}
((\lambda x.M)\ V)^\circ &= M^\#\{x := V^\#\} \\
\langle V \rangle^\circ &= V^\# \\
F[(\mathcal{S}k.M)]^\circ &= \langle M^\#\{k := (\lambda v.F^\#[v])\}\rangle \textit{ (} v \textit{ fresh)} \\
F[(\texttt{future}\ M)]^\circ &= \langle \texttt{flet}\ (p\ M^\#)\ [\![\emptyset,\ F^\#[p]]\!]\rangle \textit{ (} p \textit{ fresh)}
\end{aligned}$$

Def 5.12 ($(-)^\#$: Auxiliary parallel transition for terms and evaluation context)**.**

$$\begin{aligned}
v^\# &= v \textit{ (} v = x, c, b, p, \Box \textit{ の時)} \\
(\lambda x.M)^\# &= (\lambda x.M^\#) \\
(M\ N)^\# &= (M^\#\ N^\#) \\
\langle M \rangle^\# &= \langle M^\# \rangle \\
(\mathcal{S}k.M)^\# &= (\mathcal{S}k.M^\#) \\
\langle \texttt{flet}\ (p\ M)\ S \rangle^\# &= \langle \texttt{flet}\ (p\ M^\#)\ S^\star \rangle \\
(\texttt{future}\ M)^\# &= (\texttt{future}\ M^\#)
\end{aligned}$$

Def 5.13 ($(-)^{\#}$: Auxiliary transition for stores)**.**

$$\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\}^{\#} = \{b_1 \mapsto {V_1}^{\#}, \cdots, b_n \mapsto {V_n}^{\#}\}$$

$M^{\#}$ only affects to `flet` terms like the (original) (*spec*) transition. We note that $(-)^{\#}$ for a `flet` term uses $S^{\star}$ for the inner state other than $S^{\#}$.

5.4.2.3 The proof of Confluence

The key lemma to prove the confluence is Lemma 5.11. To prove that, we have to prove other auxiliary lemmas.

First we will show the key lemma.

Lemma 5.7. *We assume that* $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]$*, then we have the following.*

(1) If $E_1 \Rightarrow_s E_2$*,* $[\![\theta_1,\ E_1[M_1]]\!] \Rightarrow [\![\theta_2,\ E_2[M_2]]\!]$.

(2) Let x *denote both (usual) variables and communication variables. If* $V_1 \Rightarrow_s V_2$*,* $[\![\theta_1\{x := V_1\},\ M_1\{x := V_1\}]\!] \Rightarrow [\![\theta_2\{x := V_2\},\ M_2\{x := V_2\}]\!]$.

(3) If $\theta_1' \Rightarrow_s \theta_2'$ *and* $Dom(\theta_2) \cap Dom(\theta_2') = \emptyset$*,* $[\![\theta_1 \cup \theta_1',\ M_1]\!] \Rightarrow [\![\theta_2 \cup \theta_2',\ M_2]\!]$.

□

To prove Lemma 5.7 (1), we need the following Lemma 5.8.

Lemma 5.8. *We assume that* $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]$*. Then we have the following.*

(1) If $N_1 \Rightarrow_s N_2$*,* $[\![\theta_1,\ (M_1\ N_1)]\!] \Rightarrow [\![\theta_2,\ (M_2\ N_2)]\!]$.

(2) If $V_1 \Rightarrow_s V_2$*,* $[\![\theta_1,\ (V_1\ M_1)]\!] \Rightarrow [\![\theta_2,\ (V_2\ M_2)]\!]$.

(3) $[\![\theta_1,\ \langle M_1 \rangle]\!] \Rightarrow [\![\theta_2,\ \langle M_2 \rangle]\!]$.

(4) If $S_1 \Rightarrow S_2$*,* $[\![\theta_1,\ \langle \texttt{flet}\ (p\ M_1)\ S_1 \rangle]\!] \Rightarrow [\![\theta_2,\ \langle \texttt{flet}\ (p\ M_2)\ S_2 \rangle]\!]$.

□

Proof . *For all sub lemmas, we will prove by case analysis on the rule which derive the* $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]$.

(1) In the following, we will denote the $[\![\theta_1,\ M_1]\!]$ *as* S_1*, and* $[\![\theta_2,\ M_2]\!]$ *as* S_2.

(App)

By the assumption, $M_1 \equiv E_1[((\lambda x.M_1')\ V_1)]$ *and* $M_2 \equiv E_2[M_2'\{x := V_2\}]$ *where* $E_1 \Rightarrow_s E_2$*,* $M_1' \Rightarrow_s M_2'$ *and* $V_1 \Rightarrow_s V_2$*. By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$*. Then, by the definition of* $\Rightarrow$*, we have the following.*

$$[\![\theta_1,\ (M_1\ N_1)]\!] \equiv [\![\theta_1,\ (E_1[((\lambda x.M_1')\ V_1)]\ N_1)]\!] \Rightarrow [\![\theta_2,\ (E_2[M_2'\{x := V_2\}]\ N_2)]\!] \equiv [\![\theta_2,\ (M_2\ N_2)]\!]$$

(RtVl)

By the assumption, $M_1 \equiv E_1[\langle V_1 \rangle]$ *and* $M_2 \equiv E_2[V_2]$ *where* $E_1 \Rightarrow_s E_2$ *and* $V_1 \Rightarrow_s V_2$*. By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$*. Then, by definition of* $\Rightarrow$*, we have the following.*

$$[\![\theta_1,\ (M_1\ N_1)]\!] \equiv [\![\theta_1,\ (E_1[\langle V_1 \rangle]\ N_1)]\!] \Rightarrow [\![\theta_2,\ (E_2[V_2]\ N_2)]\!] \equiv [\![\theta_2,\ (M_2\ N_2)]\!]$$

(RtSt)

By the assumption, $M_1 \equiv E_1[F_1[(\mathcal{S}k.M_1')]]$ *and* $M_2 \equiv E_2[\langle M_2'\{k := (\lambda v.F_2[v])\}\rangle]$ *where* $E_1 \Rightarrow_s E_2$*,* $F_1 \Rightarrow_s F_2$ *and* $M_1' \Rightarrow_s M_2'$*. By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$*. Then, by the definition of* $\Rightarrow$*, we have the following.*

$$[\![\theta_1,\ (M_1\ N_1)]\!] \equiv [\![\theta_1,\ (E_1[F_1[(\mathcal{S}k.M_1')]]\ N_1)]\!] \Rightarrow [\![\theta_2,\ (E_2[\langle M_2'\{k := (\lambda v.F_2[v])\}\rangle]\ N_2)]\!] \equiv [\![\theta_2,\ (M_2\ N_2)]\!]$$

(*Make*)

By the assumption, $S_1 \equiv \langle\!\langle \theta_1,\ E_1[(\texttt{make}\ V_1)] \rangle\!\rangle$ *and* $S_2 \equiv \langle\!\langle \theta_2' \cup \{b \mapsto V_2\},\ E_2[b] \rangle\!\rangle$ *where* $\theta_1 \Rightarrow_s \theta_2'$, $E_1 \Rightarrow_s E_2$, $V_1 \Rightarrow_s V_2$ *and* b *is a fresh box variable. By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$. *Then, by the definition of* $\Rightarrow$, *we have the following.*

$$\langle\!\langle \theta_1,\ (M_1\ N_1) \rangle\!\rangle \equiv \langle\!\langle \theta_1,\ (E_1[(\texttt{make}\ V_1)]\ N_1) \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ (E_2[b]\ N_2) \rangle\!\rangle \equiv \langle\!\langle \theta_2,\ (M_2\ N_2) \rangle\!\rangle$$

(*Set*)

By the assumption, $S_1 \equiv \langle\!\langle \theta_1' \cup \{b \mapsto V_1'\},\ E_1[((\texttt{set!}\ b)\ V_1)] \rangle\!\rangle$ *and* $S_2 \equiv \langle\!\langle \theta_2' \cup \{b \mapsto V_2\},\ E_2[\textbf{\textit{void}}] \rangle\!\rangle$ *where* $\theta_1' \Rightarrow_s \theta_2'$, $E_1 \Rightarrow_s E_2$ *and* $V_1 \Rightarrow_s V_2$. *By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$. *Then, by definition of* $\Rightarrow$, *we have the following.*

$$\begin{aligned}\langle\!\langle \theta_1,\ (M_1\ N_1) \rangle\!\rangle &\equiv \langle\!\langle \theta_1' \cup \{b \mapsto V_1'\},\ (E_1[((\texttt{set!}\ b)\ V_1)]\ N_1) \rangle\!\rangle \\ &\Rightarrow \langle\!\langle \theta_2' \cup \{b \mapsto V_2\},\ (E_2[\textbf{\textit{void}}]\ N_2) \rangle\!\rangle \equiv \langle\!\langle \theta_2,\ (M_2\ N_2) \rangle\!\rangle\end{aligned}$$

(*Deref*)

By the assumption, $S_1 \equiv \langle\!\langle \theta_1' \cup \{b \mapsto V_1\},\ E_1[(\texttt{deref}\ b)] \rangle\!\rangle$ *and* $S_2 \equiv \langle\!\langle \theta_2' \cup \{b \mapsto V_2\},\ E_2[V_2] \rangle\!\rangle$ *where* $\theta_1' \Rightarrow_s \theta_2'$, $E_1 \Rightarrow_s E_2$ *and* $V_1 \Rightarrow_s V_2$. *By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$. *Then, by the definition of* $\Rightarrow$, *we have the following.*

$$\begin{aligned}\langle\!\langle \theta_1,\ (M_1\ N_1) \rangle\!\rangle &\equiv \langle\!\langle \theta_1' \cup \{b \mapsto V_1\},\ (E_1[(\texttt{deref}\ b)]\ N_1) \rangle\!\rangle \\ &\Rightarrow \langle\!\langle \theta_2' \cup \{b \mapsto V_2\},\ (E_2[V_2]\ N_2) \rangle\!\rangle \equiv \langle\!\langle \theta_2,\ (M_2\ N_2) \rangle\!\rangle\end{aligned}$$

(*Fork*)

By the assumption, $M_1 \equiv E_1[F_1[(\texttt{future}\ M_1')]]$ *and* $M_2 \equiv E_2[\langle \texttt{flet}\ (p\ M_2')\ \langle\!\langle \emptyset,\ F_2[p] \rangle\!\rangle \rangle]$ *where* $E_1 \Rightarrow_s E_2$, $F_1 \Rightarrow_s F_2$ *and* $M_1' \Rightarrow_s M_2'$. *By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$. *Then, by the definition of* $\Rightarrow$, *we have the following.*

$$\begin{aligned}\langle\!\langle \theta_1,\ (M_1\ N_1) \rangle\!\rangle &\equiv \langle\!\langle \theta_1,\ (E_1[F_1[(\texttt{future}\ M_1')]]\ N_1) \rangle\!\rangle \\ &\Rightarrow \langle\!\langle \theta_2,\ (E_2[\langle \texttt{flet}\ (p\ M_2')\ \langle\!\langle \emptyset,\ F_2[p] \rangle\!\rangle \rangle]\ N_2) \rangle\!\rangle \equiv \langle\!\langle \theta_2,\ (M_2\ N_2) \rangle\!\rangle\end{aligned}$$

(*Join*)

By the assumption, $S_1 \equiv \langle\!\langle \theta_1,\ E_1[\langle \texttt{flet}\ (p\ V_1)\ \langle\!\langle \theta_1',\ M_1' \rangle\!\rangle \rangle] \rangle\!\rangle$ *and* $S_2 \equiv \langle\!\langle \theta_2 \cup (\theta_2'\{p := V_2\}),\ E_2[M_2'\{p := V_2\}] \rangle\!\rangle$ *where* $E_1 \Rightarrow_s E_2$, $V_1 \Rightarrow_s V_2$, $\langle\!\langle \theta_1',\ M_1' \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2',\ M_2' \rangle\!\rangle$ *and* $Dom(\theta_1) \cap Dom(\theta_1') = \emptyset$. *By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$. *Then, by the definition of* $\Rightarrow$, *we have the following.*

$$\begin{aligned}\langle\!\langle \theta_1,\ (M_1\ N_1) \rangle\!\rangle &\equiv \langle\!\langle \theta_1,\ (E_1[\langle \texttt{flet}\ (p\ V_1)\ \langle\!\langle \theta_1',\ M_1' \rangle\!\rangle \rangle]\ N_1) \rangle\!\rangle \\ &\Rightarrow \langle\!\langle \theta_2 \cup (\theta_2'\{p := V_2\}),\ (E_2[M_2'\{p := V_2\}]\ N_2) \rangle\!\rangle \equiv \langle\!\langle \theta_2,\ (M_2\ N_2) \rangle\!\rangle\end{aligned}$$

(*Spec*)

Here we have $\theta_1 \Rightarrow_s \theta_2$ *and* $M_1 \Rightarrow_s M_2$. *By the assumption and the definition of* $\Rightarrow_s$ *for the evaluation contexts, we have that* $(E_1\ N_1) \Rightarrow_s (E_2\ N_2)$.

$$\langle\!\langle \theta_1,\ (M_1\ N_1) \rangle\!\rangle \Rightarrow \langle\!\langle \theta_2,\ (M_2\ N_2) \rangle\!\rangle$$

(2) Since we can prove this sublemma almost same as the above, we omit this proof.

(3) The proof of this sublemma is also same as the (1) so we omit.

(4) `flet` *context can be an evaluation context. Then this sublemma can be proved almost same as the (1).*

■

Proof (Lemma 5.7 (1))**.** *We prove Lemma 5.7 (1) by the induction on the structure of* E_1. *We use the other definition of* E *such as* $E ::= \Box \mid (E\ M) \mid (V\ E) \mid \langle E \rangle \mid \langle \texttt{flet}\ (p\ E)\ S \rangle$ *in order to simplify this proof. As we described in the previous section, the two definitions are same and it can be shown easily.*

$\underline{E_1 \equiv \Box}$

By the definition of $\Rightarrow$, we have $E_2 \equiv \Box$. Then we immediately obtain the following.

$$\begin{aligned} \llbracket \theta_1,\ E_1[M_1] \rrbracket &\equiv \llbracket \theta_1,\ M_1 \rrbracket \\ &\Rightarrow \llbracket \theta_2,\ M_2 \rrbracket && \textit{(by the assumption)} \\ &\equiv \llbracket \theta_2,\ E_2[M_2] \rrbracket \end{aligned}$$

$\underline{E_1 \equiv (E\ M)}$

By the assumption $E_1 \Rightarrow E_2$, there exist some E' and M' such that $E_2 \equiv (E'\ M')$, $E \Rightarrow_s E'$ and $M \Rightarrow_s M'$. By I.H., $\llbracket \theta_1,\ E[M_1] \rrbracket \Rightarrow \llbracket \theta_2,\ E'[M_2] \rrbracket$. Therefore, by Lemma 5.8 (1), we have the following.

$$\begin{aligned} \llbracket \theta_1,\ E_1[M_1] \rrbracket &\equiv \llbracket \theta_1,\ (E[M_1]\ M) \rrbracket \\ &\Rightarrow \llbracket \theta_2,\ (E'[M_2]\ M') \rrbracket \\ &\equiv \llbracket \theta_2,\ E_2[M_2] \rrbracket \end{aligned}$$

$\underline{E_1 \equiv (V\ E)}$

By the assumption $E_1 \Rightarrow E_2$, there exist some E' and V' such that $E_2 \equiv (V'\ E')$, $E \Rightarrow_s E'$ and $V \Rightarrow_s V'$. By I.H., $\llbracket \theta_1,\ E[M_1] \rrbracket \Rightarrow \llbracket \theta_2,\ E'[M_2] \rrbracket$. Therefore, by Lemma 5.8 (2), we have the following.

$$\begin{aligned} \llbracket \theta_1,\ E_1[M_1] \rrbracket &\equiv \llbracket \theta_1,\ (V\ E[M_1]) \rrbracket \\ &\Rightarrow \llbracket \theta_2,\ (V'\ E'[M_2]) \rrbracket \\ &\equiv \llbracket \theta_2,\ E_2[M_2] \rrbracket \end{aligned}$$

$\underline{E_1 \equiv \langle E \rangle}$

By the assumption $E_1 \Rightarrow E_2$, there exists some E' such that $E_2 \equiv \langle E' \rangle$ and $E \Rightarrow_s E'$. By I.H., $\llbracket \theta_1,\ E[M_1] \rrbracket \Rightarrow \llbracket \theta_2,\ E'[M_2] \rrbracket$. Therefore, by Lemma 5.8 (3), we have the following.

$$\begin{aligned} \llbracket \theta_1,\ E_1[M_1] \rrbracket &\equiv \llbracket \theta_1,\ \langle E[M_1] \rangle \rrbracket \\ &\Rightarrow \llbracket \theta_2,\ \langle E'[M_2] \rangle \rrbracket \\ &\equiv \llbracket \theta_2,\ E_2[M_2] \rrbracket \end{aligned}$$

$\underline{E_1 \equiv \langle \texttt{flet}\ (p\ E)\ S \rangle}$

By the assumption $E_1 \Rightarrow E_2$, there exist some E' and S' such that $E_2 \equiv \langle \texttt{flet}\ (p\ E')\ S' \rangle$, $E \Rightarrow_s E'$ and $S \Rightarrow S'$. By I.H., $\llbracket \theta_1,\ E[M_1] \rrbracket \Rightarrow \llbracket \theta_2,\ E'[M_2] \rrbracket$. Therefore, by Lemma 5.8 (4), we have the following.

$$\begin{aligned} \llbracket \theta_1,\ E_1[M_1] \rrbracket &\equiv \llbracket \theta_1,\ \langle \texttt{flet}\ (p\ E[M_1])\ S \rangle \rrbracket \\ &\Rightarrow \llbracket \theta_2,\ \langle \texttt{flet}\ (p\ E'[M_2])\ S' \rangle \rrbracket \\ &\equiv \llbracket \theta_2,\ E_2[M_2] \rrbracket \end{aligned}$$

■

Next we will prove Lemma 5.7. We can prove it by simultaneous induction with following Lemma 5.9.

Lemma 5.9. *When $V_1 \Rightarrow_s V_2$, we have*

(1) if $M_1 \Rightarrow_s M_2$ then $M_1\{x := V_1\} \Rightarrow_s M_2\{x := V_2\}$,

(2) if $E_1 \Rightarrow_s E_2$ then $E_1\{x := V_1\} \Rightarrow_s E_2\{x := V_2\}$,

(3) if $\theta_1 \Rightarrow_s \theta_2$ then $\theta_1\{x := V_1\} \Rightarrow_s \theta_2\{x := V_2\}$.

□

Proof (Lemma 5.7 (2) and Lemma 5.9)**.** *As we described above, we prove four lemmas simultaneously.*

Lemma 5.7 (2)

We will analyze on the rule which derived the relation $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]$.

(*App*)

By the assumption, we have that $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[((\lambda y.M)\ V)]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_2,\ E'[M'\{y := V'\}]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, $M \Rightarrow_s M'$ *and* $V \Rightarrow_s V'$. *And* $[\![\theta_1,\ M_1]\!]\{x := V_1\} \equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[((\lambda y.M)\{x := V_1\}\ V\{x := V_1\})]]\!]$. *Then, we will divide this case into two cases: whether the variable* y *is equal to* x *or not.*

$\underline{y = x}$

By the definition of substitution, $(\lambda y.M)\{x := V_1\} \equiv (\lambda y.M)$. *By I.H. of Lemma 5.9,* $\theta_1\{x := V_1\} \Rightarrow_s \theta_2\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$ *and* $V\{x := V_1\} \Rightarrow_s V'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned}
[\![\theta_1,\ M_1]\!]\{x := V_1\} &\equiv [\![\theta_1,\ E[((\lambda y.M)\ V)]]\!]\{x := V_1\} \\
&\equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[((\lambda y.M)\ V\{x := V_1\})]]\!] \\
&\Rightarrow [\![\theta_2\{x := V_2\},\ E\{x := V_2\}[M'\{y := V'\{x := V_2\}\}]]\!] \\
&\equiv [\![\theta_2,\ M'\{y := V'\}]\!]\{x := V_2\} \\
&\equiv [\![\theta_2,\ M_2]\!]\{x := V_2\}
\end{aligned}$$

$\underline{y \neq x}$

By I.H. of Lemma 5.9, $\theta_1\{x := V_1\} \Rightarrow_s \theta_2\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$, $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$ *and* $V\{x := V_1\} \Rightarrow_s V'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned}
[\![\theta_1,\ M_1]\!]\{x := V_1\} &\equiv [\![\theta_1,\ E[((\lambda y.M)\ V)]]\!]\{x := V_1\} \\
&\equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[((\lambda y.M\{x := V_1\})\ V\{x := V_1\})]]\!] \\
&\Rightarrow [\![\theta_2\{x := V_2\},\ E\{x := V_2\}[(M'\{x := V_2\})\{y := V'\{x := V_2\}\}]]\!] \\
&\equiv [\![\theta_2,\ M'\{y := V'\}]\!]\{x := V_2\} \\
&\equiv [\![\theta_2,\ M_2]\!]\{x := V_2\}
\end{aligned}$$

(*RtVl*)

By the assumption, we have that $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[\langle V\rangle]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_1,\ E[V]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, *and* $V \Rightarrow_s V'$. *And* $[\![\theta_1,\ M_1]\!]\{x := V_1\} \equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[\langle V\{x := V_1\}\rangle]]\!]$. *By I.H. of Lemma 5.9,* $\theta_1\{x := V_1\} \Rightarrow_s \theta_2\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$ *and* $V\{x := V_1\} \Rightarrow_s V'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned}
[\![\theta_1,\ M_1]\!]\{x := V_1\} &\equiv [\![\theta_1,\ E[\langle V\rangle]]\!]\{x := V_1\} \\
&\equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[\langle V\{x := V_1\}\rangle]]\!] \\
&\Rightarrow [\![\theta_2\{x := V_2\},\ E\{x := V_2\}[V\{x := V_2\}]]\!] \\
&\equiv [\![\theta_2,\ M']\!]\{x := V_2\} \\
&\equiv [\![\theta_2,\ M_2]\!]\{x := V_2\}
\end{aligned}$$

(*RtSt*)

By the assumption, we have that $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[F[(\mathcal{S}k.M)]]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_2,\ E'[\langle M'\{k := (\lambda v.F'[v])\}\rangle]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ *and* $M \Rightarrow_s M'$. *And* $[\![\theta_1,\ M_1]\!]\{x := V_1\} \equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[F\{x := V_1\}[(\mathcal{S}k.M)\{x := V_1\}]]]\!]$. *Then, we will divide this case into two cases: whether the variable* k *is equal to* x *or not.*

$\underline{k = x}$

By the definition of substitution, $(\mathcal{S}k.M)\{x := V_1\} \equiv (\mathcal{S}k.M)$. *By I.H. of Lemma 5.9,* $\theta_1\{x := V_1\} \Rightarrow_s \theta_2\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$, $F\{x := V_1\} \Rightarrow_s F'\{x := V_2\}$ *and* $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$. *Then we have the following.*

$$[\![\theta_1,\ M_1]\!]\{x := V_1\} \equiv [\![\theta_1,\ E[F[(\mathcal{S}k.M)]]]\!]\{x := V_1\}$$

$$
\begin{aligned}
&\equiv \langle\!\langle \theta_1\{x := V_1\},\ E\{x := V_1\}[F\{x := V_1\}[(\mathcal{S}k.M)]]\rangle\!\rangle \\
&\Rightarrow \langle\!\langle \theta_2\{x := V_2\},\ E\{x := V_2\}[\langle M'\{k := (\lambda v.F'\{x := V_2\}[v])\}\rangle]\rangle\!\rangle \\
&\equiv \langle\!\langle \theta_2,\ E'[\langle M'\{k := (\lambda v.F'[v])\}\rangle]\rangle\!\rangle\{x := V_2\} \\
&\equiv \langle\!\langle \theta_2,\ M_2\rangle\!\rangle\{x := V_2\}
\end{aligned}
$$

$\underline{k \neq x}$

By the definition of substitution, $(\mathcal{S}k.M)\{x := V_1\} \equiv (\mathcal{S}k.M\{x := V_1\})$. *By I.H. of Lemma 5.9,* $\theta_1\{x := V_1\} \Rightarrow_s \theta_2\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$, $F\{x := V_1\} \Rightarrow_s F'\{x := V_2\}$, $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$ *and* $V\{x := V_1\} \Rightarrow_s V'\{x := V_2\}$. *Then we have the following.*

$$
\begin{aligned}
\langle\!\langle \theta_1,\ M_1\rangle\!\rangle\{x := V_1\} &\equiv \langle\!\langle \theta_1,\ E[F[(\mathcal{S}k.M)]]\rangle\!\rangle\{x := V_1\} \\
&\equiv \langle\!\langle \theta_1\{x := V_1\},\ E\{x := V_1\}[F\{x := V_1\}[(\mathcal{S}k.M\{x := V_1\})]]\rangle\!\rangle \\
&\Rightarrow \langle\!\langle \theta_2\{x := V_2\},\ E\{x := V_2\}[\langle (M'\{x := V_2\})\{k := (\lambda v.F'\{x := V_2\}[v])\}\rangle]\rangle\!\rangle \\
&\equiv \langle\!\langle \theta_2,\ E'[\langle M'\{k := (\lambda v.F'\{x := V_2\}[v])\}\rangle]\rangle\!\rangle\{x := V_2\} \\
&\equiv \langle\!\langle \theta_2,\ M_2\rangle\!\rangle\{x := V_2\}
\end{aligned}
$$

(*Make*)

By the assumption, we have that $\langle\!\langle \theta_1,\ M_1\rangle\!\rangle \equiv \langle\!\langle \theta_1,\ E[(\texttt{make}\ V)]\rangle\!\rangle$ *and* $\langle\!\langle \theta_2,\ M_2\rangle\!\rangle \equiv \langle\!\langle \theta_2 \cup \{b \mapsto V'\},\ E'[b]\rangle\!\rangle$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, *and* $V \Rightarrow_s V'$ *and* b *is a fresh fresh box variable. And* $\langle\!\langle \theta_1,\ M_1\rangle\!\rangle\{x := V_1\} \equiv \langle\!\langle \theta_1\{x := V_1\},\ E\{x := V_1\}[(\texttt{make}\ V\{x := V_1\})]\rangle\!\rangle$. *By I.H. of Lemma 5.9,* $\theta_1\{x := V_1\} \Rightarrow_s \theta_2\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$ *and* $V\{x := V_1\} \Rightarrow_s V'\{x := V_2\}$. *Then we have the following.*

$$
\begin{aligned}
\langle\!\langle \theta_1,\ M_1\rangle\!\rangle\{x := V_1\} &\equiv \langle\!\langle \theta_1,\ E[(\texttt{make}\ V)]\rangle\!\rangle\{x := V_1\} \\
&\equiv \langle\!\langle \theta_1\{x := V_1\},\ E\{x := V_1\}[(\texttt{make}\ V\{x := V_1\})]\rangle\!\rangle \\
&\Rightarrow \langle\!\langle \theta_2\{x := V_2\} \cup \{b \mapsto (V'\{x := V_2\})\},\ E\{x := V_2\}[b]\rangle\!\rangle \\
&\equiv \langle\!\langle \theta_2 \cup \{b \mapsto V'\},\ E'[b]\rangle\!\rangle\{x := V_2\} \\
&\equiv \langle\!\langle \theta_2,\ M_2\rangle\!\rangle\{x := V_2\}
\end{aligned}
$$

(*Set*)

By the assumption, we have that $\langle\!\langle \theta_1,\ M_1\rangle\!\rangle \equiv \langle\!\langle \theta_1' \cup \{b \mapsto W\},\ E[((\texttt{set}\ !\ b)\ V)]\rangle\!\rangle$ *and* $\langle\!\langle \theta_2,\ M_2\rangle\!\rangle \equiv \langle\!\langle \theta_2' \cup \{b \mapsto V'\},\ E'[\textbf{\textit{void}}]\rangle\!\rangle$ *where* W *is a value and* $\theta_1' \Rightarrow_s \theta_2'$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *And* $\langle\!\langle \theta_1,\ M_1\rangle\!\rangle\{x := V_1\} \equiv \langle\!\langle \theta_1'\{x := V_1\} \cup \{b \mapsto W\{x := V_1\}\},\ E\{x := V_1\}[((\texttt{set}\ !\ b)\ V\{x := V_1\})]\rangle\!\rangle$. *By I.H. of Lemma 5.9,* $\theta_1'\{x := V_1\} \Rightarrow_s \theta_2'\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$ *and* $V\{x := V_1\} \Rightarrow_s V'\{x := V_2\}$. *We note that we do not have care about the value* W *since it is removed after the transition. Then we have the following.*

$$
\begin{aligned}
\langle\!\langle \theta_1,\ M_1\rangle\!\rangle\{x := V_1\} &\equiv \langle\!\langle \theta_1' \cup \{b \mapsto W\},\ E[((\texttt{set}\ !\ b)\ V)]\rangle\!\rangle\{x := V_1\} \\
&\equiv \langle\!\langle \theta_1'\{x := V_1\} \cup \{b \mapsto W\{x := V_1\}\},\ E\{x := V_1\}[((\texttt{set}\ !\ b)\ V\{x := V_1\})]\rangle\!\rangle \\
&\Rightarrow \langle\!\langle \theta_2'\{x := V_2\} \cup \{b \mapsto (V'\{x := V_2\})\},\ E\{x := V_2\}[\textbf{\textit{void}}]\rangle\!\rangle \\
&\equiv \langle\!\langle \theta_2' \cup \{b \mapsto V'\},\ E'[\textbf{\textit{void}}]\rangle\!\rangle\{x := V_2\} \\
&\equiv \langle\!\langle \theta_2,\ M_2\rangle\!\rangle\{x := V_2\}
\end{aligned}
$$

(*Deref*)

By the assumption, we have that $\langle\!\langle \theta_1,\ M_1\rangle\!\rangle \equiv \langle\!\langle \theta_1' \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]\rangle\!\rangle$ *and* $\langle\!\langle \theta_2,\ M_2\rangle\!\rangle \equiv \langle\!\langle \theta_2' \cup \{b \mapsto V'\},\ E'[V']\rangle\!\rangle$ *where* $\theta_1' \Rightarrow_s \theta_2'$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *And* $\langle\!\langle \theta_1,\ M_1\rangle\!\rangle\{x := V_1\} \equiv \langle\!\langle \theta_1'\{x := V_1\} \cup \{b \mapsto V\{x := V_1\}\},\ E\{x := V_1\}[(\texttt{deref}\ b)]\rangle\!\rangle$. *By I.H. of Lemma 5.9,* $\theta_1'\{x := V_1\} \Rightarrow_s \theta_2'\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$ *and* $V\{x := V_1\} \Rightarrow_s V'\{x := V_2\}$. *Then we have the following.*

$$
\begin{aligned}
\langle\!\langle \theta_1,\ M_1\rangle\!\rangle\{x := V_1\} &\equiv \langle\!\langle \theta_1' \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]\rangle\!\rangle\{x := V_1\} \\
&\equiv \langle\!\langle \theta_1'\{x := V_1\} \cup \{b \mapsto V\{x := V_1\}\},\ E\{x := V_1\}[(\texttt{deref}\ b)]\rangle\!\rangle \\
&\Rightarrow \langle\!\langle \theta_2'\{x := V_2\} \cup \{b \mapsto (V'\{x := V_2\})\},\ E\{x := V_2\}[V'\{x := V_2\}]\rangle\!\rangle \\
&\equiv \langle\!\langle \theta_2' \cup \{b \mapsto V'\},\ E'[V']\rangle\!\rangle\{x := V_2\} \\
&\equiv \langle\!\langle \theta_2,\ M_2\rangle\!\rangle\{x := V_2\}
\end{aligned}
$$

(*Fork*)

By the assumption, we have that $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[(\texttt{future}\ M)]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_2,\ E'[\langle \texttt{flet}\ (p\ M')\ [\![\emptyset,\ F'[p]]\!]\rangle]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$, $M \Rightarrow_s M'$. *And* $[\![\theta_1,\ M_1]\!]\{x := V_1\} \equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[F\{x := V_1\}[(\texttt{future}\ M\{x := V_1\})]]]\!]$. *By I.H. of Lemma 5.9,* $\theta_1\{x := V_1\} \Rightarrow_s \theta_2\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$, $F\{x := V_1\} \Rightarrow_s F'\{x := V_2\}$ *and* $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned}
[\![\theta_1,\ M_1]\!]\{x := V_1\} &\equiv [\![\theta_1,\ E[F[(\texttt{future}\ M)]]]\!]\{x := V_1\} \\
&\equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[F\{x := V_1\}[(\texttt{future}\ M\{x := V_1\})]]]\!] \\
&\Rightarrow [\![\theta_2\{x := V_2\},\ E\{x := V_2\}[\langle \texttt{flet}\ (p\ M'\{x := V_2\})\ [\![\emptyset,\ F'\{x := V_2\}[p]]\!]\rangle]]\!] \\
&\equiv [\![\theta_2,\ E'[\langle \texttt{flet}\ (p\ M')\ [\![\emptyset,\ F'[p]]\!]\rangle]]\!]\{x := V_2\} \\
&\equiv [\![\theta_2,\ M_2]\!]\{x := V_2\}
\end{aligned}$$

(*Join*)

By the assumption, we have that $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta,\ M]\!]\rangle]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta'_1 \cup (\theta'\{p := V'\}),\ E'[M'\{x := V'\}]]\!]$ *where* $\theta_1 \Rightarrow_s \theta'_1$, $E \Rightarrow_s E'$, $V \Rightarrow_s V'$ *and* $[\![\theta,\ M]\!] \Rightarrow [\![\theta',\ M']\!]$. *And* $[\![\theta_1,\ M_1]\!]\{x := V_1\} \equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[\langle \texttt{flet}\ (p\ V\{x := V_1\})\ [\![\theta,\ M]\!]\{x := V_1\}\rangle]]\!]$. *By I.H. of this lemma,* $[\![\theta,\ M]\!]\{x := V_1\} \Rightarrow [\![\theta',\ M']\!]\{x := V_1\}$, *and by I.H. of Lemma 5.9,* $\theta_1\{x := V_1\} \Rightarrow_s \theta'_1\{x := V_2\}$, $E\{x := V_1\} \Rightarrow_s E'\{x := V_2\}$ *and* $V\{x := V_1\} \Rightarrow_s V'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned}
[\![\theta_1,\ M_1]\!]\{x := V_1\} &\equiv [\![\theta_1,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta,\ M]\!]\rangle]]\!]\{x := V_1\} \\
&\equiv [\![\theta_1\{x := V_1\},\ E\{x := V_1\}[\langle \texttt{flet}\ (p\ V\{x := V_1\})\ [\![\theta,\ M]\!]\{x := V_1\}\rangle]]\!] \\
&\Rightarrow [\![\theta'_1\{x := V_2\} \cup ((\theta'\{x := V_2\})\{p := V'\{x := V_2\}\}), \\
&\qquad E\{x := V_2\}[(M'\{x := V_2\})\{p := V'\{x := V_2\}\}]]\!] \\
&\equiv [\![\theta'_1 \cup (\theta'\{p := V'\}),\ E'[M'\{p := V'\}]]\!]\{x := V_2\} \\
&\equiv [\![\theta_2,\ M_2]\!]\{x := V_2\}
\end{aligned}$$

(*Spec*)

By the assumption, we have that $\theta_1 \Rightarrow_s \theta_2$ *and* $M_1 \Rightarrow_s M_2$. *By I.H. of Lemma 5.9,* $\theta_1\{x := V_1\} \Rightarrow_s \theta'_1\{x := V_2\}$ *and* $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned}
[\![\theta_1,\ M_1]\!]\{x := V_1\} &\equiv [\![\theta_1\{x := V_1\},\ M_1\{x := V_1\}]\!] \\
&\Rightarrow [\![\theta_2\{x := V_2\},\ M_2\{x := V_2\}]\!] \\
&\equiv [\![\theta_2,\ M_2]\!]\{x := V_2\}
\end{aligned}$$

Lemma 5.9 (1)

We will prove by case analysis on the rule which derived the relation $M_1 \Rightarrow_s M_2$.

(*S-Val*)

The case that M_1 *is a constant value and box variable is trivial. Then we will only consider the case that* M_1 *is a variable, and we denote the variable* y. *(We note that the case that* M_1 *is a communication variable can be proved almost same as that of the variable.) We must divide that case if* y *is equal to* x *or not.*

$y = x$

We have the following.

$$\begin{aligned}
M_1\{x := V_1\} &\equiv x\{x := V_1\} \\
&\equiv V_1 \\
&\Rightarrow_s V_2
\end{aligned}$$

$M_2\{x := V_2\}$ *is* V_2 *so we have* $M_1\{x := V_1\} \Rightarrow_s M_2\{x := V_2\}$.

$y \neq x$

We have the following.

$$M_1\{x := V_1\} \equiv y\{x := V_1\}$$

$$
\begin{aligned}
&\equiv y \\
&\equiv y\{x := V_2\} \\
&\equiv M_2\{x := V_2\}
\end{aligned}
$$

(*S-Lam*)

By the assumption, we have $M_1 \equiv (\lambda y.M)$ and $M_2 \equiv (\lambda y.M')$ where $M \Rightarrow_s M'$. Then we divide this case into two sub cases: whether y equals to x or not.

$y = x$

By the definition of substitution, $(\lambda y.M)\{x := V_1\} \equiv (\lambda y.M)$. Then we have the following.

$$
\begin{aligned}
M_1\{x := V_1\} &\equiv (\lambda x.M)\{x := V_1\} \\
&\equiv (\lambda x.M) \\
&\Rightarrow_s (\lambda x.M') \\
&\equiv (\lambda x.M')\{x := V_2\} \\
&\equiv M_2\{x := V_2\}
\end{aligned}
$$

$y \neq x$

By I.H. of this lemma, $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$. Then we have the following.

$$
\begin{aligned}
M_1\{x := V_1\} &\equiv (\lambda y.M)\{x := V_1\} \\
&\equiv (\lambda y.M\{x := V_1\}) \\
&\Rightarrow_s (\lambda y.M'\{x := V_2\}) \\
&\equiv (\lambda y.M')\{x := V_2\} \\
&\equiv M_2\{x := V_2\}
\end{aligned}
$$

(*S-App*)

By the assumption, we have $M_1 \equiv (N_1\ N_2)$ and $M_2 \equiv (N_1'\ N_2')$ where $N_1 \Rightarrow_s N_1'$ and $N_2 \Rightarrow_s N_2'$. By I.H. of this lemma, $N_1\{x := V_1\} \Rightarrow_s N_1'\{x := V_2\}$ and $N_2\{x := V_1\} \Rightarrow_s N_2'\{x := V_2\}$. Then we have the following.

$$
\begin{aligned}
M_1\{x := V_1\} &\equiv (N_1\ N_2)\{x := V_1\} \\
&\equiv (N_1\{x := V_1\}\ N_2\{x := V_1\}) \\
&\Rightarrow_s (N_1'\{x := V_2\}\ N_2'\{x := V_2\}) \\
&\equiv (N_1'\ N_2')\{x := V_2\} \\
&\equiv M_2\{x := V_2\}
\end{aligned}
$$

(*S-Rt*)

By the assumption, we have $M_1 \equiv \langle M \rangle$ and $M_2 \equiv \langle M' \rangle$ where $M \Rightarrow_s M'$. By I.H. of this lemma, $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$. Then we have the following.

$$
\begin{aligned}
M_1\{x := V_1\} &\equiv \langle M \rangle\{x := V_1\} \\
&\equiv \langle M\{x := V_1\} \rangle \\
&\Rightarrow_s \langle M'\{x := V_2\} \rangle \\
&\equiv \langle M' \rangle\{x := V_2\} \\
&\equiv M_2\{x := V_2\}
\end{aligned}
$$

(*S-St*)

By the assumption, we have $M_1 \equiv (\mathcal{S}k.M)$ and $M_2 \equiv (\mathcal{S}k.M')$ where $M \Rightarrow_s M'$. Then we divide this case into two cases: whether k is equal to x or not.

$\underline{k = x}$

By the definition of substitution, $(\mathcal{S}k.M)\{x := V_1\} \equiv (\mathcal{S}k.M)$. *Then we have the following.*

$$\begin{aligned} M_1\{x := V_1\} &\equiv (\mathcal{S}k.M)\{x := V_1\} \\ &\equiv (\mathcal{S}k.M) \\ &\Rightarrow_s (\mathcal{S}k.M') \\ &\equiv (\mathcal{S}k.M')\{x := V_2\} \\ &\equiv M_2\{x := V_2\} \end{aligned}$$

$\underline{k \neq x}$

By I.H. of this lemma, $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned} M_1\{x := V_1\} &\equiv (\mathcal{S}k.M)\{x := V_1\} \\ &\equiv (\mathcal{S}k.M\{x := V_1\}) \\ &\Rightarrow_s (\mathcal{S}k.M'\{x := V_2\}) \\ &\equiv (\mathcal{S}k.M')\{x := V_2\} \\ &\equiv M_2\{x := V_2\} \end{aligned}$$

(*S-Ft*)

By the assumption, we have $M_1 \equiv (\texttt{future}\ M)$ *and* $M_2 \equiv (\texttt{future}\ M')$ *where* $M \Rightarrow_s M'$. *By I.H. of this lemma,* $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned} M_1\{x := V_1\} &\equiv (\texttt{future}\ M)\{x := V_1\} \\ &\equiv (\texttt{future}\ M\{x := V_1\}) \\ &\Rightarrow_s (\texttt{future}\ M'\{x := V_2\}) \\ &\equiv (\texttt{future}\ M')\{x := V_2\} \\ &\equiv M_2\{x := V_2\} \end{aligned}$$

(*S-Flet*)

By the assumption, we have $M_1 \equiv \langle \texttt{flet}\ (p\ M)\ S\rangle$ *and* $M_2 \equiv \langle \texttt{flet}\ (p\ M')\ S'\rangle$ *where* $M \Rightarrow_s M'$ *and* $S \Rightarrow S'$. *Then we divide this case into two sub cases: whether* p *is equal to* x *or not.*

$\underline{p = x}$

By the definition of substitution, $\langle \texttt{flet}\ (p\ M)\ S\rangle\{x := V_1\} \equiv \langle \texttt{flet}\ (p\ M\{x := V_1\})\ S\rangle$. *Then we have the following.*

$$\begin{aligned} M_1\{x := V_1\} &\equiv \langle \texttt{flet}\ (p\ M)\ S\rangle\{x := V_1\} \\ &\equiv \langle \texttt{flet}\ (p\ M\{x := V_1\})\ S\rangle \\ &\Rightarrow_s \langle \texttt{flet}\ (p\ M'\{x := V_2\})\ S'\rangle \\ &\equiv \langle \texttt{flet}\ (p\ M')\ S'\rangle\{x := V_2\} \\ &\equiv M_2\{x := V_2\} \end{aligned}$$

$\underline{p \neq x}$

By I.H. of this lemma, $M\{x := V_1\} \Rightarrow_s M'\{x := V_2\}$ *and by I.H. of Lemma 5.7 (2),* $S\{x := V_1\} \Rightarrow S'\{x := V_2\}$. *Then we have the following.*

$$\begin{aligned} M_1\{x := V_1\} &\equiv \langle \texttt{flet}\ (p\ M)\ S\rangle\{x := V_1\} \\ &\equiv \langle \texttt{flet}\ (p\ M\{x := V_1\})\ S\{x := V_1\}\rangle \\ &\Rightarrow_s \langle \texttt{flet}\ (p\ M'\{x := V_2\})\ S'\{x := V_2\}\rangle \\ &\equiv \langle \texttt{flet}\ (p\ M')\ S'\rangle\{x := V_2\} \\ &\equiv M_2\{x := V_2\} \end{aligned}$$

Lemma 5.9 (2)

Since this can be proved almost same as the above one, we omit this proof.

Lemma 5.9 (3)

Let θ_1 be $\{b_1 \mapsto W_1, \cdots, b_n \mapsto W_n\}$ and θ_2 be $\{b_1 \mapsto W_1', \cdots, b_n \mapsto W_n'\}$ where each W_i is a value and $W_i \Rightarrow_s W_i' (1 \leq i \leq n)$. Then we have that $\theta_1\{x := V_1\} \equiv \{b_1 \mapsto W_1\{x := V_1\}, \cdots, b_n \mapsto W_n\{x := V_1\}\}$. By I.H. of Lemma 5.9 (1), we have $W_i\{x := V_1\} \Rightarrow_s W_i'\{x := V_2\}$ for each i. Then we have the following.

$$\begin{aligned}
\theta_1\{x := V_1\} &\equiv \{b_1 \mapsto W_1, \cdots, b_n \mapsto W_n\}\{x := V_1\} \\
&\equiv \{b_1 \mapsto W_1\{x := V_1\}, \cdots, b_n \mapsto W_n\{x := V_1\}\} \\
&\Rightarrow_s \{b_1 \mapsto W_1'\{x := V_2\}, \cdots, b_n \mapsto W_n'\{x := V_2\}\} \\
&\equiv \{b_1 \mapsto W_1', \cdots, b_n \mapsto W_n'\}\{x := V_2\} \\
&\equiv M_2\{x := V_2\}
\end{aligned}$$

■

We have to prove the next Lemma 5.10 in order to prove Lemma 5.7 (3).

Lemma 5.10. *If $\theta_1 \Rightarrow_s \theta_1'$ and $\theta 2 \Rightarrow_s \theta_2'$ then $\theta_1 \cup \theta_2 \Rightarrow_s \theta_1' \cup \theta_2'$.* □

Proof . *Let θ_2 be $\{b_{2,1} \mapsto V_{2,1}, \cdots, b_{2,n} \mapsto V_{2,n}\}$ and θ_2' be $\{b_{2,1} \mapsto V_{2,1}', \cdots, b_{2,n} \mapsto V_{2,n}'\}$. Then, by the definition of $\Rightarrow_s$, we have*

$$\theta_1 \cup \{b_{2,1} \mapsto V_{2,1}, \cdots, b_{2,n} \mapsto V_{2,n}\} \Rightarrow_s \theta_1' \cup \{b_{2,1} \mapsto V_{2,1}', \cdots, b_{2,n} \mapsto V_{2,n}'\}$$

■

Proof (Lemma 5.7 (3))**.** *We will prove this Lemma 5.7 (3) by induction: proof by case analysis on the rule which is applied to derive the relation of $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]$.*

(***App***)

By the assumption, we have $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[((\lambda x.M)\ V)]]\!]$ and $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_2,\ E'[M'\{x := V'\}]]\!]$ where $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, $M \Rightarrow_s M'$ and $V \Rightarrow_s V'$. By Lemma 5.10, $\theta_1 \cup \theta_1' \Rightarrow_s \theta_2 \cup \theta_2'$. Then we have the following.

$$\begin{aligned}
[\![\theta_1 \cup \theta_1',\ M_1]\!] &\equiv [\![\theta_1 \cup \theta_1',\ E[((\lambda x.M)\ V)]]\!] \\
&\Rightarrow [\![\theta_2 \cup \theta_2',\ E'[M'\{x := V'\}]]\!] \\
&\equiv [\![\theta_2 \cup \theta_2',\ M_2]\!]
\end{aligned}$$

(***RtVl***)

By the assumption, we have $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[\langle V\rangle]]\!]$ and $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_2,\ E'[V']]\!]$ where $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$ and $V \Rightarrow_s V'$. By Lemma 5.10, $\theta_1 \cup \theta_1' \Rightarrow_s \theta_2 \cup \theta_2'$. Then we have the following.

$$\begin{aligned}
[\![\theta_1 \cup \theta_1',\ M_1]\!] &\equiv [\![\theta_1 \cup \theta_1',\ E[\langle V\rangle]]\!] \\
&\Rightarrow [\![\theta_2 \cup \theta_2',\ E'[V']]\!] \\
&\equiv [\![\theta_2 \cup \theta_2',\ M_2]\!]
\end{aligned}$$

(***RtSt***)

By the assumption, we have $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[F[(\mathcal{S}k.M)]]]\!]$ and $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_2,\ E'[\langle M'\{k := (\lambda v.F'[v])\}\rangle]]\!]$ where $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ and $M \Rightarrow_s M'$. By Lemma 5.10, $\theta_1 \cup \theta_1' \Rightarrow_s \theta_2 \cup \theta_2'$. Then we have the following.

$$\begin{aligned}
[\![\theta_1 \cup \theta_1',\ M_1]\!] &\equiv [\![\theta_1 \cup \theta_1',\ E[F[(\mathcal{S}k.M)]]]\!] \\
&\Rightarrow [\![\theta_2 \cup \theta_2',\ E'[\langle M'\{k := (\lambda v.F'[v])\}\rangle]]\!] \\
&\equiv [\![\theta_2 \cup \theta_2',\ M_2]\!]
\end{aligned}$$

(***Make***)

By the assumption, we have $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[(\texttt{make}\ V)]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_3 \cup \{b \mapsto V'\},\ E'[b]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_3$, $E \Rightarrow_s E'$, $V \Rightarrow_s V'$ *and* b *is a fresh box variable. By Lemma 5.10,* $\theta_1 \cup \theta'_1 \Rightarrow_s \theta_3 \cup \theta'_2$. *Then we have the following.*

$$\begin{aligned}[\![\theta_1 \cup \theta'_1,\ M_1]\!] &\equiv [\![\theta_1 \cup \theta'_1,\ E[(\texttt{make}\ V)]]\!] \\ &\Rightarrow [\![\theta_2 \cup \theta'_2 \cup \{b \mapsto V'\},\ E'[b]]\!] \\ &\equiv [\![\theta_2 \cup \theta'_2,\ M_2]\!]\end{aligned}$$

(***Set***)

By the assumption, we have $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_3 \cup \{b \mapsto W\},\ E[((\texttt{set!}\ b)\ V)]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_4 \cup \{b \mapsto V'\},\ E'[\textbf{\textit{void}}]]\!]$ *where* $\theta_3 \Rightarrow_s \theta_4$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *By Lemma 5.10,* $\theta_3 \cup \theta'_1 \Rightarrow_s \theta_4 \cup \theta'_2$. *Then we have the following.*

$$\begin{aligned}[\![\theta_1 \cup \theta'_1,\ M_1]\!] &\equiv [\![\theta_3 \cup \{b \mapsto W\} \cup \theta'_1,\ E[((\texttt{set!}\ b)\ V)]]\!] \\ &\Rightarrow [\![\theta_4 \cup \{b \mapsto V'\} \cup \theta'_2,\ E'[\textbf{\textit{void}}]]\!] \\ &\equiv [\![\theta_2 \cup \theta'_2,\ M_2]\!]\end{aligned}$$

(***Deref***)

By the assumption, we have $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_3 \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_4 \cup \{b \mapsto V'\},\ E'[V']]\!]$ *where* $\theta_3 \Rightarrow_s \theta_4$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *By Lemma 5.10,* $\theta_3 \cup \theta'_1 \Rightarrow_s \theta_4 \cup \theta'_2$. *Then we have the following.*

$$\begin{aligned}[\![\theta_1 \cup \theta'_1,\ M_1]\!] &\equiv [\![\theta_3 \cup \{b \mapsto V\} \cup \theta'_1,\ E[(\texttt{deref}\ b)]]\!] \\ &\Rightarrow [\![\theta_4 \cup \{b \mapsto V'\} \cup \theta'_2,\ E'[V']]\!] \\ &\equiv [\![\theta_2 \cup \theta'_2,\ M_2]\!]\end{aligned}$$

(***Fork***)

By the assumption, we have $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[F[(\texttt{future}\ M)]]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_2,\ E'[\langle \texttt{flet}\ (p\ M')\ [\![\emptyset,\ F'[p]]\!]\rangle]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ *and* $M \Rightarrow_s M'$. *By Lemma 5.10,* $\theta_1 \cup \theta'_1 \Rightarrow_s \theta_1 \cup \theta'_2$. *Then we have the following.*

$$\begin{aligned}[\![\theta_1 \cup \theta'_1,\ M_1]\!] &\equiv [\![\theta_1 \cup \theta'_1,\ E[F[(\texttt{future}\ M)]]]\!] \\ &\Rightarrow [\![\theta_2 \cup \theta'_2,\ E'[\langle \texttt{flet}\ (p\ M')\ [\![\emptyset,\ F'[p]]\!]\rangle]]\!] \\ &\equiv [\![\theta_2 \cup \theta'_2,\ M_2]\!]\end{aligned}$$

(***Join***)

By the assumption, we have $[\![\theta_1,\ M_1]\!] \equiv [\![\theta_1,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta,\ M]\!]\rangle]]\!]$ *and* $[\![\theta_2,\ M_2]\!] \equiv [\![\theta_2 \cup (\theta'\{p := V'\}),\ E'[M'\{p := V'\}]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E \Rightarrow_s E'$, $V \Rightarrow_s V'$ *and* $[\![\theta,\ M]\!] \Rightarrow [\![\theta',\ M']\!]$. *By Lemma 5.10,* $\theta_1 \cup \theta'_1 \Rightarrow_s \theta_2 \cup \theta'_2$. *Then we have the following.*

$$\begin{aligned}[\![\theta_1 \cup \theta'_1,\ M_1]\!] &\equiv [\![\theta_1 \cup \theta'_1,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta,\ M]\!]\rangle]]\!] \\ &\Rightarrow \theta_2 \cup (\theta'\{p := V'\}) \cup \theta'_2 E'[M'\{p := V'\}] \\ &\equiv [\![\theta_2 \cup \theta'_2,\ M_2]\!]\end{aligned}$$

(***Spec***)

By the assumption, $\theta_1 \Rightarrow_s \theta_2$ *and* $M_1 \Rightarrow_s M_2$. *By Lemma 5.10,* $\theta_1 \cup \theta'_1 \Rightarrow_s \theta_2 \cup \theta'_2$. *Then we have the following.*

$$[\![\theta_1 \cup \theta'_1,\ M_1]\!] \Rightarrow \theta_2 \cup \theta'_2 M_2$$

■

From here, we will show and prove the key, Lemma 5.11, to prove our confluence. Before we show the auxiliary lemmas to prove Lemma 5.11, we will show Lemma 5.11.

Lemma 5.11. *We have the following properties about states S_i, stores θ_i, terms M_i and evaluation contexts E_i (where $i = 1, 2$). In the following, we will use a metavariable X_i to denote one of a store, a term or an evaluation context.*

(0) $S_1 \Rightarrow S_1$,

(0') $X_1 \Rightarrow_s X_1$,

(1) if $S_1 \to S_2$ *then* $S_1 \Rightarrow S_2$,

(2) if $S_1 \Rightarrow S_2$ *then* $S_1 \to^* S_2$,

(2') if $\theta_1 \Rightarrow_s \theta_2$ *and* $E_1 \Rightarrow_s E_2$, $M_1 \Rightarrow_s M_2$ *then* $\{\!|\theta_1,\ E_1[M_1]|\!\} \to^* \{\!|\theta_2,\ E_2[M_2]|\!\}$,

(3) $S_1 \Rightarrow {S_1}^\star$,

(3') $X_1 \Rightarrow_s {X_1}^\#$,

(4) if $S_1 \Rightarrow S_2$ *then* $S_2 \Rightarrow {S_1}^\star$,

(4') if $X_1 \Rightarrow_s X_2$ *then* $X_2 \Rightarrow_s {X_1}^\#$,

(5) if $S_1 \Rightarrow S_2$ *is a mandatory other than (Join) transition then* $S_2 \Rightarrow {S_1}^\star$ *is a speculative transition,*

(5') if $S_1 \Rightarrow S_2$ *is a speculative transition and* $S_1 \Rightarrow {S_1}^\star$ *is a mandatory one then* $S_2 \Rightarrow {S_1}^\star$ *is a mandatory transition.*
□

□

First, we will show Lemma 5.11 (0) and (0').

Proof (Lemma 5.11 (0), (0'))**.** *Proof by simultaneous induction on the structure of S_1 or X_1.*

Lemma 5.11 (0)
Let S_1 be $\{\!|\theta,\ M|\!\}$. By I.H. of (0'), we have $\theta \Rightarrow_s \theta$ and $M \Rightarrow_s M$. Then by the rule (Spec), we have $S_1 \Rightarrow S_2$.

Lemma 5.11 (0')
By case analysis on the structure of X_1. Since the case that X_1 is a store can be proved easily and the case that X_1 is an evaluation context can be proved almost same as the case that X_1 is a term, we will prove only the case that X_1 is a term.

$X_1 \equiv x, b, p, c$
These cases are trivial so we omit this case.

$X_1 \equiv (\lambda x.M)$
By I.H., $M \Rightarrow_s M$. Then, by the rule (S-Lam), $(\lambda x.M) \Rightarrow_s (\lambda x.M)$.

$X_1 \equiv (M_1\ M_2)$
By I.H., $M_1 \Rightarrow_s M_1$ and $M_2 \Rightarrow_s M_2$. Then, by the rule (S-App), $(M_1\ M_2) \Rightarrow_s (M_1\ M_2)$.

$X_1 \equiv \langle M \rangle$
By I.H., $M \Rightarrow_s M$. Then, by the rule (S-Rt), $\langle M \rangle \Rightarrow_s \langle M \rangle$.

$X_1 \equiv (\mathcal{S}k.M)$
By I.H., $M \Rightarrow_s M$. Then, by the rule (S-St), $(\mathcal{S}k.M) \Rightarrow_s (\mathcal{S}k.M)$.

$X_1 \equiv (\texttt{future}\ M)$
By I.H., $M \Rightarrow_s M$. Then, by the rule (S-Ft), $(\texttt{future}\ M) \Rightarrow_s (\texttt{future}\ M)$.

$X_1 \equiv \langle \texttt{flet}\ (p\ M)\ S \rangle$
By I.H., $M \Rightarrow_s M$, and by I.H. of the (0), $S \Rightarrow S$. Then, by the rule (S-Flet), $\langle \texttt{flet}\ (p\ M)\ S \rangle \Rightarrow_s \langle \texttt{flet}\ (p\ M)\ S \rangle$.

■

Lemma 5.12. *If $M_1 \Rightarrow_s M_2$ and $E_1 \Rightarrow_s E_2$ then $E_1[M_1] \Rightarrow_s E_2[M_2]$.* □

Proof . *Prove by induction on the structure of E. We will use the other definition of E here (defined in the previous section).*

$\underline{E_1 \equiv \Box}$
$E_1[M_1] \equiv M_1$ and $E_2[M_2] \equiv M_2$. Then $E_1[M_1] \equiv M_1 \Rightarrow_s M_2 \equiv E_2[M_2]$.

$\underline{E_1 \equiv (E\ M)}$
By the definition of $\Rightarrow_s$, there exist some E' and M' such that $E_2 \equiv (E'\ M')$, $E \Rightarrow_s E'$ and $M \Rightarrow_s M'$. By I.H., $E[M_1] \Rightarrow_s E'[M_2]$. Then, by the definition of the rule (S-App), $(E[M_1]\ M) \Rightarrow_s (E'[M_2]\ M')$.

$\underline{E_1 \equiv (V\ E)}$
By the definition of $\Rightarrow_s$, there exist some E' and V' such that $E_2 \equiv (V'\ E')$, $E \Rightarrow_s E'$ and $V \Rightarrow_s V'$. By I.H., $E[M_1] \Rightarrow_s E'[M_2]$. Then, by the definition of the rule (S-App), $(V\ E[M_1]) \Rightarrow_s (V'\ E'[M_2])$.

$\underline{E_1 \equiv \langle E \rangle}$
By the definition of $\Rightarrow_s$, there exists some E' such that $E_2 \equiv \langle E' \rangle$ and $E \Rightarrow_s E'$. By I.H., $E[M_1] \Rightarrow_s E'[M_2]$. Then, by the definition of the rule (S-Rt), $\langle E[M_1] \rangle \Rightarrow_s \langle E'[M_2] \rangle$.

$\underline{E_1 \equiv \langle \texttt{flet}\ (p\ E)\ S \rangle}$
By the definition of $\Rightarrow_s$, there exist some E' and S' such that $E_2 \equiv \langle E' \rangle$, $E \Rightarrow_s E'$ and $S \Rightarrow S'$. By I.H., $E[M_1] \Rightarrow_s E'[M_2]$. Then, by the definition of the rule (S-Flet), $\langle \texttt{flet}\ (p\ E[M_1])\ S \rangle \Rightarrow_s \langle \texttt{flet}\ (p\ E'[M_2])\ S' \rangle$.

■

Lemma 5.13. *If $S_1 \Rightarrow S_2$ then $C[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \Rightarrow_s C[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle]$.* □

Proof . *Proof by induction on the structure of C.*

$\underline{C \equiv \Box}$
By the rule (S-Flet), it is clear $\langle \texttt{flet}\ (p\ M)\ S_1 \rangle \Rightarrow_s \langle \texttt{flet}\ (p\ M)\ S_2 \rangle$.

$\underline{C \equiv (\lambda x.C_1)}$
By I.H., $C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \Rightarrow_s C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle]$. Then, by the rule (S-Lam), $(\lambda x.C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle]) \Rightarrow_s (\lambda x.C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle])$.

$\underline{C \equiv (C_1\ M_1)}$
By I.H., $C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \Rightarrow_s C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle]$. By Lemma 5.11 (0'), $M_1 \Rightarrow_s M_1$. Then, by the rule (S-App), $(C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle]\ M_1) \Rightarrow_s (\lambda x.C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle])$.

$\underline{C \equiv (M_1\ C_1)}$
By I.H., $C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \Rightarrow_s C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle]$. By Lemma 5.11 (0'), $M_1 \Rightarrow_s M_1$. Then, by the rule (S-App), $(M_1\ C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle]) \Rightarrow_s (M_1\ C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle])$.

$\underline{C \equiv \langle C_1 \rangle}$
By I.H., $C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \Rightarrow_s C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle]$. Then, by the rule (S-Rt), $\langle C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \rangle \Rightarrow_s \langle C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle] \rangle$.

$\underline{C \equiv (\mathcal{S}k.C_1)}$
By I.H., $C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \Rightarrow_s C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle]$. Then, by the rule (S-St), $(\mathcal{S}k.C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle]) \Rightarrow_s (\mathcal{S}k.C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle])$.

$\underline{C \equiv (\texttt{future}\ C_1)}$
By I.H., $C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \Rightarrow_s C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle]$. Then, by the rule (S-Ft), $(\texttt{future}\ C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle]) \Rightarrow_s (\texttt{future}\ C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle])$.

$\underline{C \equiv \langle \texttt{flet}\ (p\ C_1)\ S \rangle}$
By I.H., $C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle] \Rightarrow_s C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle]$. Then, by the rule (S-Flet), $(\texttt{future}\ C_1[\langle \texttt{flet}\ (p\ M)\ S_1 \rangle]) \Rightarrow_s (\texttt{future}\ C_1[\langle \texttt{flet}\ (p\ M)\ S_2 \rangle])$.

$C \equiv \langle \texttt{flet}\ (p\ M_1)\ [\![\theta,\ C_1]\!]\rangle$

By I.H., $C_1[\langle \texttt{flet}\ (p\ M)\ S_1\rangle] \Rightarrow_s C_1[\langle \texttt{flet}\ (p\ M)\ S_2\rangle]$. *By the definition of* $\Rightarrow$, $[\![\theta,\ C_1[\langle \texttt{flet}\ (p\ M)\ S_1\rangle]]\!] \Rightarrow [\![\theta,\ C_1[\langle \texttt{flet}\ (p\ M)\ S_2\rangle]]\!]$. *Then, by the rule (S-Flet),* $\langle \texttt{flet}\ (p\ M_1)\ [\![\theta,\ C_1[\langle \texttt{flet}\ (p\ M)\ S_1\rangle]]\!]\rangle \Rightarrow \langle \texttt{flet}\ (p\ M_1)\ [\![\theta,\ C_1[\langle \texttt{flet}\ (p\ M)\ S_2\rangle]]\!]\rangle$.

■

By the above two lemmas, we can prove Lemma 5.11 (1).

Proof (Lemma 5.11 (1))**.** *Proof by induction and we will show that by case analysis on the rule which is used to derived the relation* $S_1 \to S_2$.

(*app*)

By the assumption, we have $S_1 \equiv [\![\theta,\ E[((\lambda x.M)\ V)]]\!]$ *and* $S_2 \equiv [\![\theta,\ E[M\{x := V\}]]\!]$. *By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$, $E \Rightarrow_s E$, $M \Rightarrow_s M$ *and* $V \Rightarrow_s V$. *Then, by the rule (App),* $[\![\theta,\ E[((\lambda x.M)\ V)]]\!] \Rightarrow [\![\theta,\ E[M\{x := V\}]]\!]$.

(*rv*)

By the assumption, we have $S_1 \equiv [\![\theta,\ E[\langle V\rangle]]\!]$ *and* $S_2 \equiv [\![\theta,\ E[V]]\!]$. *By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$, $E \Rightarrow_s E$ *and* $V \Rightarrow_s V$. *Then, by the rule (RtVl),* $[\![\theta,\ E[\langle V\rangle]]\!] \Rightarrow [\![\theta,\ E[V]]\!]$.

(*rs*)

By the assumption, we have $S_1 \equiv [\![\theta,\ E[F[(\mathcal{S}k.M)]]]\!]$ *and* $S_2 \equiv [\![\theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle]]\!]$. *By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$, $E \Rightarrow_s E$, $F \Rightarrow_s F$ *and* $M \Rightarrow_s M$. *Then, by the rule (RtSt),* $[\![\theta,\ E[F[(\mathcal{S}k.M)]]]\!] \Rightarrow [\![\theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle]]\!]$.

(*make*)

By the assumption, we have $S_1 \equiv [\![\theta,\ E[(\texttt{make}\ V)]]\!]$ *and* $S_2 \equiv [\![\theta \cup \{b \mapsto V\},\ E[b]]\!]$ *where* b *is a fresh box variable. By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$, $E \Rightarrow_s E$ *and* $V \Rightarrow_s V$. *Then, by the rule (Make),* $[\![\theta,\ E[(\texttt{make}\ V)]]\!] \Rightarrow [\![\theta \cup \{b \mapsto V\},\ E[b]]\!]$.

(*set*)

By the assumption, we have $S_1 \equiv [\![\theta \cup \{b \mapsto V_1\},\ E[((\texttt{set!}\ b)\ V)]]\!]$ *and* $S_2 \equiv [\![\theta \cup \{b \mapsto V\},\ E[\textbf{\textit{void}}]]\!]$. *By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$, $E \Rightarrow_s E$ *and* $V \Rightarrow_s V$. *Then, by the rule (Set),* $[\![\theta \cup \{b \mapsto V_1\},\ E[((\texttt{set!}\ b)\ V)]]\!] \Rightarrow [\![\theta \cup \{b \mapsto V\},\ E[\textbf{\textit{void}}]]\!]$.

(*deref*)

By the assumption, we have $S_1 \equiv [\![\theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]]\!]$ *and* $S_2 \equiv [\![\theta \cup \{b \mapsto V\},\ E[V]]\!]$. *By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$, $E \Rightarrow_s E$ *and* $V \Rightarrow_s V$. *Then, by the rule (Deref),* $[\![\theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]]\!] \Rightarrow [\![\theta \cup \{b \mapsto V\},\ E[V]]\!]$.

(*fork*)

By the assumption, we have $S_1 \equiv [\![\theta,\ E[F[(\texttt{future}\ M)]]]\!]$ *and* $S_2 \equiv [\![\theta,\ E[\langle \texttt{flet}\ (p\ M)\ [\![\emptyset,\ F[p]]\!]\rangle]]\!]$ *where* p *is a fresh communication variable. By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$, $E \Rightarrow_s E$, $F \Rightarrow_s F$ *and* $M \Rightarrow_s M$. *Then, by the rule (Fork),* $[\![\theta,\ E[F[(\texttt{future}\ M)]]]\!] \Rightarrow [\![\theta,\ E[\langle \texttt{flet}\ (p\ M)\ [\![\emptyset,\ F[p]]\!]\rangle]]\!]$.

(*join*)

By the assumption, we have $S_1 \equiv [\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!]\rangle]]\!]$ *and* $S_2 \equiv [\![\theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}]]\!]$. *By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$, $E \Rightarrow_s E$, *and by Lemma 5.11 (0),* $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_1,\ M_1]\!]$. *Then, by the rule (Join),* $[\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!]\rangle]]\!] \Rightarrow [\![\theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}]]\!]$.

(*spec*)

By the assumption, we have $S_1 \equiv [\![\theta,\ C[\langle \texttt{flet}\ (p\ M)\ [\![\theta_1,\ M_1]\!]\rangle]]\!]$ *and* $S_2 \equiv [\![\theta,\ C[\langle \texttt{flet}\ (p\ M)\ [\![\theta_2,\ M_2]\!]\rangle]]\!]$ *where* $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]$. *By Lemma 5.11 (0'),* $\theta \Rightarrow_s \theta$ *and by Lemma 5.13,* $C[\langle \texttt{flet}\ (p\ M)\ [\![\theta_1,\ M_1]\!]\rangle] \Rightarrow_s C[\langle \texttt{flet}\ (p\ M)\ [\![\theta_2,\ M_2]\!]\rangle]$. *Then, by the rule (Spec),* $[\![\theta,\ C[\langle \texttt{flet}\ (p\ M)\ [\![\theta_1,\ M_1]\!]\rangle]]\!] \Rightarrow [\![\theta,\ C[\langle \texttt{flet}\ (p\ M)\ [\![\theta_2,\ M_2]\!]\rangle]]\!]$.

(*store*)

Let θ *be* $\theta' \cup \{b \mapsto V\}$. *Then, by the assumption, we have* $S_1 \equiv [\![\theta' \cup \{b \mapsto V\},\ M]\!]$ *and* $S_2 \equiv [\![\theta' \cup \{b \mapsto V'\},\ M]\!]$ *where* $[\![\emptyset,\ V]\!] \to [\![\emptyset,\ V']\!]$. *By Lemma 5.11 (0'), we have* $\theta' \Rightarrow_s \theta'$ *and* $M \Rightarrow_s M$. *By I.H.,* $[\![\emptyset,\ V]\!] \Rightarrow [\![\emptyset,\ V']\!]$. *Since* V *is a value, we can find that* $[\![\emptyset,\ V]\!] \Rightarrow [\![\emptyset,\ V']\!]$ *is a speculative transition, which means that* $V \Rightarrow_s V'$. *Therefore, we have that* $\theta' \cup \{b \mapsto V\} \Rightarrow_s \theta' \cup \{b \mapsto V'\}$ *by the rule (S-Store). Then we have* $[\![\theta' \cup \{b \mapsto V\},\ M]\!] \Rightarrow [\![\theta' \cup \{b \mapsto V'\},\ M]\!]$.

In order to prove Lemma 5.11 (2) and (2'), we need the next lemma, Lemma 5.14. We can prove the Lemma 5.11 (2) by simultaneous induction with the next lemma 5.14, and the Lemma 5.11 (2') can be immediately derived from them.

Lemma 5.14. *(1) If* $\theta_1 \Rightarrow_s \theta_2$ *then* $[\![\theta_1,\ M]\!] \rightarrow^* [\![\theta_2,\ M]\!]$ *for arbitrary* M.

(2) If $M_1 \Rightarrow_s M_2$ *then* $[\![\theta,\ C[M_1]]\!] \rightarrow^* [\![\theta,\ C[M_2]]\!]$ *for arbitrary* θ *and* C.

(3) If $E_1 \Rightarrow_s E_2$ *then* $[\![\theta,\ E_1[M]]\!] \rightarrow^* [\![\theta,\ E_2[M]]\!]$ *for arbitrary* θ *and* M.

Proof . *Proof by simultaneous induction with the three lemmas.*

Lemma 5.11 (2)

Proof by induction, and we will show the proof by case analysis on the rule which is used to derive the relation $S_1 \Rightarrow S_2$.

(App)

By the assumption, $S_1 \equiv [\![\theta_1,\ E_1[((\lambda x.M_1)\ V_1)]]\!]$ *and* $S_2 \equiv [\![\theta_2,\ E_2[M_2\{x := V_2\}]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E_1 \Rightarrow_s E_2$, $M_1 \Rightarrow_s M_2$ *and* $V_1 \Rightarrow_s V_2$. *Then, by I.H. of Lemma 5.14 (1),* $[\![\theta_1,\ M]\!] \rightarrow^* [\![\theta_2,\ M]\!]$ *for arbitrary* M. *Also, by I.H. of Lemma 5.14 (2),* $[\![\theta,\ E_1[((\lambda x.M_1)\ V_1)]]\!] \rightarrow^* [\![\theta,\ E_2[((\lambda x.M_2)\ V_2)]]\!]$ *for any* θ. *Then we have the following.*

$$
\begin{array}{lll}
S_1 \equiv [\![\theta_1,\ E_1[((\lambda x.M_1)\ V_1)]]\!] & & \\
\quad \rightarrow^* [\![\theta_2,\ E_1[((\lambda x.M_1)\ V_1)]]\!] & \textit{by I.H. of Lemma 5.14 (1)} \\
\quad \rightarrow^* [\![\theta_2,\ E_2[((\lambda x.M_2)\ V_2)]]\!] & \textit{by I.H. of Lemma 5.14 (2)} \\
\quad \rightarrow [\![\theta_2,\ E_2[M_2\{x := V_2\}]]\!] & \textit{by rule (app)} \\
\quad \equiv S_2 &
\end{array}
$$

(RtVl)

By the assumption, $S_1 \equiv [\![\theta_1,\ E_1[\langle V_1\rangle]]\!]$ *and* $S_2 \equiv [\![\theta_2,\ E_2[V_2]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E_1 \Rightarrow_s E_2$ *and* $V_1 \Rightarrow_s V_2$. *Then, by I.H. of Lemma 5.14 (1),* $[\![\theta_1,\ M]\!] \rightarrow^* [\![\theta_2,\ M]\!]$ *for any* M. *Also, by I.H. of Lemma 5.14 (2),* $[\![\theta,\ E_1[\langle V_1\rangle]]\!] \rightarrow^* [\![\theta,\ E_2[\langle V_2\rangle]]\!]$ *for arbitrary* θ. *Then we have the following.*

$$
\begin{array}{lll}
S_1 \equiv [\![\theta_1,\ E_1[\langle V_1\rangle]]\!] & & \\
\quad \rightarrow^* [\![\theta_2,\ E_1[\langle V_1\rangle]]\!] & \textit{by I.H. of Lemma 5.14 (1)} \\
\quad \rightarrow^* [\![\theta_2,\ E_2[\langle V_2\rangle]]\!] & \textit{by I.H. of Lemma 5.14 (2)} \\
\quad \rightarrow [\![\theta_2,\ E_2[V_2]]\!] & \textit{by rule (rv)} \\
\quad \equiv S_2 &
\end{array}
$$

(RtSt)

By the assumption, $S_1 \equiv [\![\theta_1,\ E_1[F_1[(\mathcal{S}k.M_1)]]]\!]$ *and* $S_2 \equiv [\![\theta_2,\ E_2[\langle M_2\{k := (\lambda v.F_2[v])\}\rangle]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E_1 \Rightarrow_s E_2$, $F_1 \Rightarrow_s F_2$ *and* $M_1 \Rightarrow_s M_2$. *Then, by I.H. of Lemma 5.14 (1),* $[\![\theta_1,\ M]\!] \rightarrow^* [\![\theta_2,\ M]\!]$ *for any* M. *Also, by I.H. of Lemma 5.14 (2),* $[\![\theta,\ E_1[F[(\mathcal{S}k.M_1)]]]\!] \rightarrow^* [\![\theta,\ E_2[F_2[(\mathcal{S}k.M_2)]]]\!]$ *for arbitrary* θ. *Then we have the following.*

$$
\begin{array}{lll}
S_1 \equiv [\![\theta_1,\ E_1[F_1[(\mathcal{S}k.M_1)]]]\!] & & \\
\quad \rightarrow^* [\![\theta_2,\ E_1[F_1[(\mathcal{S}k.M_1)]]]\!] & \textit{by I.H. of Lemma 5.14 (1)} \\
\quad \rightarrow^* [\![\theta_2,\ E_2[F_2[(\mathcal{S}k.M_2)]]]\!] & \textit{by I.H. of Lemma 5.14 (2)} \\
\quad \rightarrow [\![\theta_2,\ E_2[\langle M_2\{k := (\lambda v.F_2[v])\}\rangle]]\!] & \textit{by rule (rs)} \\
\quad \equiv S_2 &
\end{array}
$$

(Make)

By the assumption, $S_1 \equiv [\![\theta_1,\ E_1[(\texttt{make}\ V_1)]]\!]$ *and* $S_2 \equiv [\![\theta_2 \cup \{b \mapsto V_2\},\ E_2[b]]\!]$ *where* $\theta_1 \Rightarrow_s \theta_2$, $E_1 \Rightarrow_s E_2$, $V_1 \Rightarrow_s V_2$ *and* b *is a fresh box variable. Then, by I.H. of Lemma 5.14 (1),* $[\![\theta_1,\ M]\!] \rightarrow^* [\![\theta_2,\ M]\!]$

for any M. Also, by I.H. of Lemma 5.14 (2), $\langle\!\langle \theta,\ E_1[(\texttt{make}\ V_1)]\rangle\!\rangle \to^* \langle\!\langle \theta,\ E_2[(\texttt{make}\ V_2)]\rangle\!\rangle$ for any θ. Then we have the following.

$$
\begin{array}{ll}
S_1 \equiv \langle\!\langle \theta_1,\ E_1[(\texttt{make}\ V_1)]\rangle\!\rangle & \\
\quad \to^* \langle\!\langle \theta_2,\ E_1[(\texttt{make}\ V_1)]\rangle\!\rangle & \textit{by I.H. of Lemma 5.14 (1)} \\
\quad \to^* \langle\!\langle \theta_2,\ E_2[(\texttt{make}\ V_2)]\rangle\!\rangle & \textit{by I.H. of Lemma 5.14 (2)} \\
\quad \to \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ E_2[\textbf{\textit{void}}]\rangle\!\rangle & \textit{by rule (make)} \\
\quad \equiv S_2 &
\end{array}
$$

(*Set*)

By the assumption, $S_1 \equiv \langle\!\langle \theta_1 \cup \{b \mapsto V\},\ E_1[((\texttt{set!}\ b)\ V_1)]\rangle\!\rangle$ and $S_2 \equiv \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ E_2[\textbf{\textit{void}}]\rangle\!\rangle$ where $\theta_1 \Rightarrow_s \theta_2$, $E_1 \Rightarrow_s E_2$, $V_1 \Rightarrow_s V_2$. Then, by I.H. of Lemma 5.14 (1), $\langle\!\langle \theta_1 \cup \{b \mapsto V\},\ M\rangle\!\rangle \to^* \langle\!\langle \theta_2 \cup \{b \mapsto V\},\ M\rangle\!\rangle$ for any M. Also, by I.H. of Lemma 5.14 (2), $\langle\!\langle \theta,\ E_1[((\texttt{set!}\ b)\ V_1)]\rangle\!\rangle \to^* \langle\!\langle \theta,\ E_2[((\texttt{set!}\ b)\ V_2)]\rangle\!\rangle$ for arbitrary θ. Then we have the following.

$$
\begin{array}{ll}
S_1 \equiv \langle\!\langle \theta_1 \cup \{b \mapsto V\},\ E_1[((\texttt{set!}\ b)\ V_1)]\rangle\!\rangle & \\
\quad \to^* \langle\!\langle \theta_2 \cup \{b \mapsto V\},\ E_1[((\texttt{set!}\ b)\ V_1)]\rangle\!\rangle & \textit{by I.H. of Lemma 5.14 (1)} \\
\quad \to^* \langle\!\langle \theta_2 \cup \{b \mapsto V\},\ E_2[((\texttt{set!}\ b)\ V_2)]\rangle\!\rangle & \textit{by I.H. of Lemma 5.14 (2)} \\
\quad \to \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ E_2[\textbf{\textit{void}}]\rangle\!\rangle & \textit{by rule (set)} \\
\quad \equiv S_2 &
\end{array}
$$

(*Deref*)

By the assumption, $S_1 \equiv \langle\!\langle \theta_1 \cup \{b \mapsto V_1\},\ E_1[(\texttt{deref}\ b)]\rangle\!\rangle$ and $S_2 \equiv \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ E_2[V_2]\rangle\!\rangle$ where $\theta_1 \Rightarrow_s \theta_2$, $E_1 \Rightarrow_s E_2$, $V_1 \Rightarrow_s V_2$. Then, by I.H. of Lemma 5.14 (1), $\langle\!\langle \theta_1 \cup \{b \mapsto V_1\},\ M\rangle\!\rangle \to^* \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ M\rangle\!\rangle$ for arbitrary M. Also, by I.H. of Lemma 5.14 (2), $\langle\!\langle \theta,\ E_1[(\texttt{deref}\ b)]\rangle\!\rangle \to^* \langle\!\langle \theta,\ E_2[(\texttt{deref}\ b)]\rangle\!\rangle$ for arbitrary θ. Then we have the following.

$$
\begin{array}{ll}
S_1 \equiv \langle\!\langle \theta_1 \cup \{b \mapsto V_1\},\ E_1[(\texttt{deref}\ b)]\rangle\!\rangle & \\
\quad \to^* \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ E_1[(\texttt{deref}\ b)]\rangle\!\rangle & \textit{by I.H. of Lemma 5.14 (1)} \\
\quad \to^* \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ E_2[(\texttt{deref}\ b)]\rangle\!\rangle & \textit{by I.H. of Lemma 5.14 (2)} \\
\quad \to \langle\!\langle \theta_2 \cup \{b \mapsto V_2\},\ E_2[V_2]\rangle\!\rangle & \textit{by rule (deref)} \\
\quad \equiv S_2 &
\end{array}
$$

(*Fork*)

By the assumption, $S_1 \equiv \langle\!\langle \theta_1,\ E_1[F_1[(\texttt{future}\ M_1)]]\rangle\!\rangle$ and $S_2 \equiv \langle\!\langle \theta_2,\ E_2[\langle \texttt{flet}\ (p\ M_2)\ \langle\!\langle \emptyset,\ F_2[p]\rangle\!\rangle\rangle]\rangle\!\rangle$ where $\theta_1 \Rightarrow_s \theta_2$, $E_1 \Rightarrow_s E_2$, $F_1 \Rightarrow_s F_2$ and $M_1 \Rightarrow_s M_2$. Then, by I.H. of Lemma 5.14 (1), $\langle\!\langle \theta_1,\ M\rangle\!\rangle \to^* \langle\!\langle \theta_2,\ M\rangle\!\rangle$ for any M. Also, by I.H. of Lemma 5.14 (2), $\langle\!\langle \theta,\ E_1[F_1[(\texttt{future}\ M_1)]]\rangle\!\rangle \to^* \langle\!\langle \theta,\ E_2[F_2[(\texttt{future}\ M_2)]]\rangle\!\rangle$ for arbitrary θ. Then we have the following.

$$
\begin{array}{ll}
S_1 \equiv \langle\!\langle \theta_1,\ E_1[F_1[(\texttt{future}\ M_1)]]\rangle\!\rangle & \\
\quad \to^* \langle\!\langle \theta_2,\ E_1[F_1[(\texttt{future}\ M_1)]]\rangle\!\rangle & \textit{by I.H. of Lemma 5.14 (1)} \\
\quad \to^* \langle\!\langle \theta_2,\ E_2[F_2[(\texttt{future}\ M_2)]]\rangle\!\rangle & \textit{by I.H. of Lemma 5.14 (2)} \\
\quad \to \langle\!\langle \theta_2,\ E_2[\langle \texttt{flet}\ (p\ M_2)\ \langle\!\langle \emptyset,\ F_2[p]\rangle\!\rangle\rangle]\rangle\!\rangle & \textit{by rule (fork)} \\
\quad \equiv S_2 &
\end{array}
$$

(*Join*)

By the assumption, $S_1 \equiv \langle\!\langle \theta_1,\ E_1[\langle \texttt{flet}\ (p\ V_1)\ \langle\!\langle \theta,\ M\rangle\!\rangle\rangle]\rangle\!\rangle$ and $S_2 \equiv \langle\!\langle \theta_2 \cup \theta'\{p := V_2\},\ E_2[M'\{p := V_1\}]\rangle\!\rangle$ where $\theta_1 \Rightarrow_s \theta_2$, $E_1 \Rightarrow_s E_2$, $V_1 \Rightarrow_s V_2$ and $\langle\!\langle \theta,\ M\rangle\!\rangle \Rightarrow \langle\!\langle \theta',\ M'\rangle\!\rangle$. Then, by I.H. of Lemma 5.14 (1), $\langle\!\langle \theta_1,\ M\rangle\!\rangle \to^* \langle\!\langle \theta_2,\ M\rangle\!\rangle$ for any M. Also, by I.H. of Lemma 5.14 (2), $\langle\!\langle \theta,\ E_1[\langle \texttt{flet}\ (p\ V_1)\ \langle\!\langle \theta,\ M\rangle\!\rangle\rangle]\rangle\!\rangle \to^* \langle\!\langle \theta,\ E_2[\langle \texttt{flet}\ (p\ V_2)\ \langle\!\langle \theta,\ M\rangle\!\rangle\rangle]\rangle\!\rangle$ for arbitrary θ. Furthermore, by I.H. of this lemma, $\langle\!\langle \theta,\ M\rangle\!\rangle \to^* \langle\!\langle \theta',\ M'\rangle\!\rangle$. Then we have the following.

$$S_1 \equiv \langle\!\langle \theta_1,\ E_1[\langle \texttt{flet}\ (p\ V_1)\ \langle\!\langle \theta,\ M\rangle\!\rangle\rangle]\rangle\!\rangle$$

$\rightarrow^* [\![\theta_2,\ E_1[\langle \texttt{flet}\ (p\ V_1)\ [\![\theta,\ M]\!]\rangle]]\!]$ *by I.H. of Lemma 5.14 (1)*

$\rightarrow^* [\![\theta_2,\ E_2[\langle \texttt{flet}\ (p\ V_2)\ [\![\theta,\ M]\!]\rangle]]\!]$ *by I.H. of Lemma 5.14 (2)*

$\rightarrow^* [\![\theta_2,\ E_2[\langle \texttt{flet}\ (p\ V_2)\ [\![\theta',\ M']\!]\rangle]]\!]$ *by the rule (spec)*

$\rightarrow [\![\theta_2 \cup \theta'\{p := V_2\},\ E_2[M'\{p := V_1\}]]\!]$ *by rule (join)*

$\equiv S_2$

(*Spec*)

By the assumption, $S_1 \equiv [\![\theta_1,\ M_1]\!]$ and $S_2 \equiv [\![\theta_2,\ M_2]\!]$ where $\theta_1 \Rightarrow_s \theta_2$ and $M_1 \Rightarrow_s M_2$. Then, we have the following.

$S_1 \equiv [\![\theta_1,\ M_1]\!]$

$\rightarrow^* [\![\theta_2,\ M_1]\!]$ *by I.H. of Lemma 5.14 (1)*

$\rightarrow^* [\![\theta_2,\ M_2]\!]$ *by I.H. of Lemma 5.14 (2)*

$\equiv S_2$

Lemma 5.14 (1)

Let θ_1 be $\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\}$. By the assumption, $\theta_2 \equiv \{b_1 \mapsto V_1', \cdots, b_n \mapsto V_n'\}$ where $V_i \Rightarrow_s V_i'$ $(1 \leq i \leq n)$. By I.H. of Lemma 5.14 (2), $[\![\emptyset,\ V_i]\!] \rightarrow^* [\![\emptyset,\ V_i']\!]$ for each V_i. Then, by applying the rule (store), we have the following.

$[\![\theta_1,\ M]\!] \equiv [\![\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\},\ M]\!]$

$\rightarrow^* [\![\{b_1 \mapsto V_1', \cdots, b_n \mapsto V_n\},\ M]\!]$

$\cdots$

$\rightarrow^* [\![\{b_1 \mapsto V_1', \cdots, b_n \mapsto V_n'\},\ M]\!]$

$\equiv [\![\theta_2,\ M]\!]$

Lemma 5.14 (2)

Proof by induction on the structure of M_1. We note that C denotes an arbitrary context, which may be a context that is not an evaluation context.

$M_1 \equiv c, x, b, p$

In this case, since M_2 is equivalent of M_1, it is clear that $[\![\theta,\ C[M_1]]\!] \rightarrow^0 [\![\theta,\ C[M_2]]\!]$.

$M_1 \equiv (\lambda x.M)$

By the assumption, $M_2 \equiv (\lambda x.M')$ where $M \Rightarrow_s M'$. Then, by I.H., we have $[\![\theta,\ C[(\lambda x.M)]]\!] \rightarrow^* [\![\theta,\ C[(\lambda x.M')]]\!]$.

$M_1 \equiv (N_1\ N_2)$

By the assumption, $M_2 \equiv (N_1'\ N_2')$ where $N_1 \Rightarrow_s N_1'$ and $N_2 \Rightarrow_s N_2'$. Then, by I.H., we have $[\![\theta,\ C[(N_1\ N_2)]]\!] \rightarrow^* [\![\theta,\ C[(N_1'\ N_2)]]\!] \rightarrow^* [\![\theta,\ C[(N_1'\ N_2')]]\!]$.

$M_1 \equiv \langle M \rangle$

By the assumption, $M_2 \equiv \langle M' \rangle$ where $M \Rightarrow_s M'$. Then, by I.H., we have $[\![\theta,\ C[\langle M \rangle]]\!] \rightarrow^* [\![\theta,\ C[\langle M' \rangle]]\!]$.

$M_1 \equiv (\mathcal{S}k.M)$

By the assumption, $M_2 \equiv (\mathcal{S}k.M')$ where $M \Rightarrow_s M'$. Then, by I.H., we have $[\![\theta,\ C[(\mathcal{S}k.M)]]\!] \rightarrow^* [\![\theta,\ C[(\mathcal{S}k.M')]]\!]$.

$M_1 \equiv (\textbf{future}\ M)$

By the assumption, $M_2 \equiv (\texttt{future}\ M')$ where $M \Rightarrow_s M'$. Then, by I.H., we have $[\![\theta,\ C[(\mathcal{S}k.M)]]\!] \rightarrow^* [\![\theta,\ C[(\mathcal{S}k.M')]]\!]$.

$M_1 \equiv \langle \textbf{flet}\ (p\ M)\ S \rangle$

By the assumption, $M_2 \equiv \langle \texttt{flet}\ (p\ M')\ S' \rangle$ where $M \Rightarrow_s M'$ and $S \Rightarrow S'$. By I.H. of Lemma 5.11 (2), $S_1 \rightarrow^* S_2$. Then, by I.H., we have $[\![\theta,\ C[\langle \texttt{flet}\ (p\ M)\ S \rangle]]\!] \rightarrow^* [\![\theta,\ C[\langle \texttt{flet}\ (p\ M')\ S \rangle]]\!] \rightarrow^* [\![\theta,\ C[\langle \texttt{flet}\ (p\ M')\ S' \rangle]]\!]$.

From here, we will show the proofs of Lemma 5.11 (3), (3'), (4), (4'), (5) and (5').

Proof (Lemma 5.11 (3),(3'))**.**

Lemma 5.11 (3)

Proof by induction on the structure of S_1.

$S_1 \equiv \langle\!\langle \theta,\ E[((\lambda x.M)\ V)] \rangle\!\rangle$

By the assumption, $S_1{}^\star \equiv \langle\!\langle \theta^\#,\ E^\#[M^\#\{x := V^\#\}] \rangle\!\rangle$. *By I.H. of Lemma 5.11 (3'),* $\theta \Rightarrow_s \theta^\#$, $E \Rightarrow_s E^\#$, $M \Rightarrow_s M^\#$ *and* $V \Rightarrow_s V^\#$. *Then, by the rule (App), we have*

$$\langle\!\langle \theta,\ E[((\lambda x.M)\ V)] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\#,\ E^\#[M^\#\{x := V^\#\}] \rangle\!\rangle$$

$S_1 \equiv \langle\!\langle \theta,\ E[\langle V \rangle] \rangle\!\rangle$

By the assumption, $S_1{}^\star \equiv \langle\!\langle \theta^\#,\ E^\#[V^\#] \rangle\!\rangle$. *By I.H. of Lemma 5.11 (3'),* $\theta \Rightarrow_s \theta^\#$, $E \Rightarrow_s E^\#$ *and* $V \Rightarrow_s V^\#$. *Then, by the (RtVl), we have*

$$\langle\!\langle \theta,\ E[\langle V \rangle] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\#,\ E^\#[V^\#] \rangle\!\rangle$$

$S_1 \equiv \langle\!\langle \theta,\ E[F[(\mathcal{S}k.M)]] \rangle\!\rangle$

By the assumption, $S_1{}^\star \equiv \langle\!\langle \theta^\#,\ E^\#[F^\#[M^\#]] \rangle\!\rangle$. *By I.H. of Lemma 5.11 (3'),* $\theta \Rightarrow_s \theta^\#$, $E \Rightarrow_s E^\#$, $F \Rightarrow_s F^\#$ *and* $V \Rightarrow_s V^\#$. *Then, by the (RtSt), we have*

$$\langle\!\langle \theta,\ E[F[(\mathcal{S}k.M)]] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\#,\ E^\#[F^\#[M^\#]] \rangle\!\rangle$$

$S_1 \equiv \langle\!\langle \theta,\ E[(\mathbf{make}\ V)] \rangle\!\rangle$

By the assumption, $S_1{}^\star \equiv \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[b] \rangle\!\rangle$ *where* b *is a fresh box variable. By I.H. of Lemma 5.11 (3'),* $\theta \Rightarrow_s \theta^\#$, $E \Rightarrow_s E^\#$ *and* $V \Rightarrow_s V^\#$. *Then, by the (Make), we have*

$$\langle\!\langle \theta,\ E[(\texttt{make}\ V)] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[b] \rangle\!\rangle$$

$S_1 \equiv \langle\!\langle \theta \cup \{b \mapsto V_1\},\ E[((\mathbf{set!}\ V)\ ])] \rangle\!\rangle$

By the assumption, $S_1{}^\star \equiv \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[\textbf{\textit{void}}] \rangle\!\rangle$. *By I.H. of Lemma 5.11 (3'),* $\theta \Rightarrow_s \theta^\#$, $E \Rightarrow_s E^\#$ *and* $V \Rightarrow_s V^\#$. *Then, by the (Set), we have*

$$\langle\!\langle \theta \cup \{b \mapsto V_1\},\ E[((\texttt{set!}\ V)\ ])] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[\textbf{\textit{void}}] \rangle\!\rangle$$

$S_1 \equiv \langle\!\langle \theta \cup \{b \mapsto V\},\ E[(\mathbf{deref}\ b)] \rangle\!\rangle$

By the assumption, $S_1{}^\star \equiv \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[V^\#] \rangle\!\rangle$. *By I.H. of Lemma 5.11 (3'),* $\theta \Rightarrow_s \theta^\#$, $E \Rightarrow_s E^\#$ *and* $V \Rightarrow_s V^\#$. *Then, by the (Deref), we have*

$$\langle\!\langle \theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[V^\#] \rangle\!\rangle$$

$S_1 \equiv \langle\!\langle \theta,\ E[F[(\mathbf{future}\ M)]] \rangle\!\rangle$

By the assumption, $S_1{}^\star \equiv \langle\!\langle \theta^\#,\ E^\#[\langle \texttt{flet}\ (p\ M^\#)\ \langle\!\langle \emptyset,\ F^\#[p] \rangle\!\rangle \rangle] \rangle\!\rangle$ *where* p *is a fresh communication variable. By I.H. of Lemma 5.11 (3'),* $\theta \Rightarrow_s \theta^\#$, $E \Rightarrow_s E^\#$, $F \Rightarrow_s F^\#$ *and* $M \Rightarrow_s M^\#$. *Then, by the (Fork), we have*

$$\langle\!\langle \theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[V^\#] \rangle\!\rangle$$

$S_1 \equiv \langle\!\langle \theta,\ E[\langle \mathbf{flet}\ (p\ V)\ \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle \rangle] \rangle\!\rangle$

By the assumption, $S_1{}^\star \equiv \langle\!\langle \theta^\# \cup ({\theta_2}^\#\{p := V^\#\}),\ E^\#[M^\#\{p := V^\#\}] \rangle\!\rangle$ *where* $\langle\!\langle \theta_2,\ M_2 \rangle\!\rangle \equiv \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle^\star$. *By I.H. of Lemma 5.11 (3'),* $\theta \Rightarrow_s \theta^\#$, $E \Rightarrow_s E^\#$ *and* $M \Rightarrow_s M^\#$. *Also by I.H. of this lemma,* $\langle\!\langle \theta_1,\ M_1 \rangle\!\rangle \Rightarrow \langle\!\langle \theta_1,\ M_1 \rangle\!\rangle^\star \equiv \langle\!\langle \theta_2,\ M_2 \rangle\!\rangle$. *Then, by the (Fork), we have*

$$\langle\!\langle \theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)] \rangle\!\rangle \Rightarrow \langle\!\langle \theta^\# \cup \{b \mapsto V^\#\},\ E^\#[V^\#] \rangle\!\rangle$$

otherwise

In this case, $S_1 \equiv [\![\theta,\ M]\!]$ and $S_1{}^\star \equiv [\![\theta^\#,\ M^\#]\!]$. By I.H. of Lemma 5.11 (3'), $\theta \Rightarrow_s \theta^\#$ and $M \Rightarrow_s M^\#$. Then, by the (Spec), we have

$$[\![\theta,\ M]\!] \Rightarrow [\![\theta^\#,\ M^\#]\!]$$

Lemma 5.11 (3')

We note that the proof for evaluation contexts is almost same as that for terms so we will prove them simultaneously.

X is a term or an evaluation context

Proof by induction. In this proof, the meta-variable M denotes a term or an evaluation context.

$X \equiv c, x, b, p, \Box$

These cases are trivial.

$X \equiv (\lambda x.M)$

By the assumption, $X^\# \equiv (\lambda x.M^\#)$. By I.H. of this lemma, $M \Rightarrow_s M^\#$. Then, by the rule (S-Lam), we have $(\lambda x.M) \Rightarrow_s (\lambda x.M^\#)$.

$X \equiv (M_1\ M_2)$

By the assumption, $X^\# \equiv ({M_1}^\#\ {M_2}^\#)$. By I.H. of this lemma, $M_1 \Rightarrow_s {M_1}^\#$ and $M_2 \Rightarrow_s {M_2}^\#$. Then, by the rule (S-App), we have $(M_1\ M_2) \Rightarrow_s ({M_1}^\#\ {M_2}^\#)$.

$X \equiv \langle M \rangle$

By the assumption, $X^\# \equiv \langle M^\# \rangle$. By I.H. of this lemma, $M \Rightarrow_s M^\#$. Then, by the rule (S-Rt), we have $\langle M \rangle \Rightarrow_s \langle M^\# \rangle$.

$X \equiv (\mathcal{S}k.M)$

By the assumption, $X^\# \equiv (\mathcal{S}k.M^\#)$. By I.H. of this lemma, $M \Rightarrow_s M^\#$. Then, by the rule (S-St), we have $(\mathcal{S}k.M) \Rightarrow_s (\mathcal{S}k.M^\#)$.

$X \equiv (\texttt{future}\ M)$

By the assumption, $X^\# \equiv (\texttt{future}\ M^\#)$. By I.H. of this lemma, $M \Rightarrow_s M^\#$. Then, by the rule (S-Ft), we have $(\texttt{future}\ M) \Rightarrow_s (\texttt{future}\ M^\#)$.

$X \equiv \langle \texttt{flet}\ (p\ M)\ S \rangle$

By the assumption, $X^\# \equiv \langle \texttt{flet}\ (p\ M^\#)\ S^\star \rangle$. By I.H. of this lemma, $M \Rightarrow_s M^\#$ and by I.H. of Lemma 5.11 (3), $S \Rightarrow S^\star$. Then, by the rule (S-Flet), we have $\langle \texttt{flet}\ (p\ M)\ S \rangle \Rightarrow_s \langle \texttt{flet}\ (p\ M^\#)\ S^\star \rangle$.

X is a store

Let X be $\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\}$. By I.H. of this lemma, $V_i \Rightarrow_s {V_i}^\#$ for each i. Then, by the rule (S-Store), we have $\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\} \Rightarrow_s \{b_1 \mapsto {V_1}^\#, \cdots, b_n \mapsto {V_n}^\#\}$.

Proof (Lemma 5.11 (4) and (4'))**.** *We will prove the two lemmas by simultaneous induction.*

Lemma 5.11 (4)

We will prove the two by induction on the size of the derivation of the relation $\Rightarrow$ and $\Rightarrow_s$. We will divide the proof by the last rule used to derive the relation $S_1 \Rightarrow S_2$.

(App)

By the assumption, $S_1 \equiv [\![\theta,\ E[((\lambda x.M)\ V)]]\!]$ and $S_2 \equiv [\![\theta',\ E'[M'\{x := V'\}]]\!]$ where $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $M \Rightarrow_s M'$ and $V \Rightarrow_s V'$. By I.H. of Lemma 5.11 (4'), $\theta' \Rightarrow_s \theta^\#$, $E' \Rightarrow_s E^\#$, $M' \Rightarrow_s M^\#$ and $V' \Rightarrow_s V^\#$. Since $M'\{x := V'\} \Rightarrow_s M^\#\{x := V^\#\}$ by Lemma 5.9, $E'[M'\{x := V'\}] \Rightarrow_s E^\#[M^\#\{x := V^\#\}]$ by Lemma 5.7. Then we have $S_2 \Rightarrow [\![\theta^\#,\ E^\#[M^\#\{x := V^\#\}]]\!] \equiv S_1{}^\star$.

(RtVl)

By the assumption, $S_1 \equiv [\![\theta,\ E[\langle V \rangle]]\!]$ and $S_2 \equiv [\![\theta',\ E'[V']]\!]$ where $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$ and $V \Rightarrow_s V'$. By I.H. of Lemma 5.11 (4'), $\theta' \Rightarrow_s \theta^\#$, $E' \Rightarrow_s E^\#$, $M' \Rightarrow_s M^\#$ and $V' \Rightarrow_s V^\#$. By Lemma 5.7, $E'[V'] \Rightarrow_s E^\#[V^\#]$. Then we have $S_2 \Rightarrow [\![\theta^\#,\ E^\#[V^\#]]\!] \equiv S_1{}^\star$.

(RtSt)

By the assumption, $S_1 \equiv [\![\theta,\ E[F[(\mathcal{S}k.M)]]]\!]$ and $S_2 \equiv [\![\theta',\ E'[\langle M'\{k := (\lambda v.F'[v])\}\rangle]]\!]$ where $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ and $M \Rightarrow_s M'$. By I.H. of (4'), $\theta' \Rightarrow_s \theta^\#$, $E' \Rightarrow_s E^\#$, $F' \Rightarrow_s F^\#$ and $M' \Rightarrow_s M^\#$. Then, by the fact that $S_1{}^\star \equiv [\![\theta^\#,\ E^\#[\langle M^\#\{k := (\lambda v.F^\#[v])\}\rangle]]\!]$ and Lemma 5.7, $S_2 \Rightarrow {S_1}^\#$.

(*Make*)

By the assumption, $S_1 \equiv [\![\theta,\ E[(\texttt{make}\ V)]]\!]$ *and* $S_2 \equiv [\![\theta' \cup \{b \mapsto V'\},\ E'[b]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$ *and* b *is a fresh box variable. By I.H. of (4'),* $\theta' \Rightarrow_s \theta^{\#}$, $E' \Rightarrow_s E^{\#}$ *and* $V' \Rightarrow_s V^{\#}$. *Then, by Lemma 5.7, we have the following.*

$$S_2 \equiv [\![\theta' \cup \{b \mapsto V'\},\ E'[b]]\!] \Rightarrow [\![\theta^{\#} \cup \{b \mapsto V^{\#}\},\ E^{\#}[b]]\!] \equiv {S_1}^{\star}$$

(*Set*)

By the assumption, $S_1 \equiv [\![\theta \cup \{b \mapsto V_1\},\ E[((\texttt{set!}\ b)\ V)]]\!]$ *and* $S_2 \equiv [\![\theta' \cup \{b \mapsto V'\},\ E'[\textbf{\textit{void}}]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *By I.H. of (4'),* $\theta' \Rightarrow_s \theta^{\#}$, $E' \Rightarrow_s E^{\#}$ *and* $V' \Rightarrow_s V^{\#}$. *Then, by Lemma 5.7, we have the following.*

$$S_2 \equiv [\![\theta' \cup \{b \mapsto V'\},\ E'[\textbf{\textit{void}}]]\!] \Rightarrow [\![\theta^{\#} \cup \{b \mapsto V^{\#}\},\ E^{\#}[\textbf{\textit{void}}]]\!] \equiv {S_1}^{\star}$$

(*Deref*)

By the assumption, $S_1 \equiv [\![\theta \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]]\!]$ *and* $S_2 \equiv [\![\theta' \cup \{b \mapsto V'\},\ E'[V']]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *By I.H. of (4'),* $\theta' \Rightarrow_s \theta^{\#}$, $E' \Rightarrow_s E^{\#}$ *and* $V' \Rightarrow_s V^{\#}$. *Then, by Lemma 5.7, we have the following.*

$$S_2 \equiv [\![\theta' \cup \{b \mapsto V'\},\ E'[V']]\!] \Rightarrow [\![\theta^{\#} \cup \{b \mapsto V^{\#}\},\ E^{\#}[V^{\#}]]\!] \equiv {S_1}^{\star}$$

(*Fork*)

By the assumption, $S_1 \equiv [\![\theta,\ E[F[(\texttt{future}\ M)]]]\!]$ *and* $S_2 \equiv [\![\theta',\ E'[\langle \texttt{flet}\ (p\ M')\ [\![\emptyset,\ F'[p]]\!]\rangle]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ *and* $M \Rightarrow_s M'$. *By I.H. of (4'),* $\theta' \Rightarrow_s \theta^{\#}$, $E' \Rightarrow_s E^{\#}$, $F' \Rightarrow_s F^{\#}$ *and* $V' \Rightarrow_s V^{\#}$. *Then, by Lemma 5.7, we have the following.*

$$S_2 \equiv [\![\theta,\ E[F[(\texttt{future}\ M)]]]\!] \Rightarrow [\![\theta^{\#},\ E^{\#}[\langle \texttt{flet}\ (p\ M^{\#})\ [\![\emptyset,\ F^{\#}[p]]\!]\rangle]]\!] \equiv {S_1}^{\star}$$

(*Join*)

By the assumption, $S_1 = [\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!]\rangle]]\!]$ *and* $S_2 = [\![\theta' \cup \theta_2\{p := V'\},\ E'[M_2\{p := V'\}]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $V \Rightarrow_s V'$ *and* $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]$. *By I.H. of this lemma,* $[\![\theta_2,\ M_2]\!] \Rightarrow [\![\theta_1,\ M_1]\!]^{\star}$ *and we will denote* $[\![\theta_1,\ M_1]\!]^{\star}$ *as* $[\![\theta_3,\ M_3]\!]$. *Also, by I.H. of (4'),* $\theta' \Rightarrow_s \theta^{\#}$, $E' \Rightarrow_s E^{\#}$ *and* $V' \Rightarrow_s V^{\#}$. *Then, by Lemma 5.7 (3),* $[\![\theta_2\{p := V'\},\ M_2\{p := V'\}]\!] \Rightarrow [\![\theta_3\{p := V^{\#}\},\ M_3\{p := V^{\#}\}]\!]$. *Also by Lemma 5.7 (4) and (1),* $[\![\theta' \cup \theta_2\{p := V'\},\ E'[M_2\{p := V'\}]]\!] \Rightarrow [\![\theta^{\#} \cup \theta_3\{p := V^{\#}\},\ E^{\#}[M_3\{p := V^{\#}\}]]\!]$.

(*Spec*)

By case analysis on the structure of S_1.

$\underline{S_1 \equiv [\![\theta,\ E[((\lambda x.M)\ V)]]\!]}$

By the fact that $S_1 \Rightarrow S_2$ *is speculative,* $S_2 \equiv [\![\theta,\ E[((\lambda x.M)\ V)]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $M \Rightarrow_s M'$ *and* $V \Rightarrow_s V'$. *Then, as the case (App), we have* $S_2 \Rightarrow {S_1}^{\star}$.

$\underline{S_1 \equiv [\![\theta,\ E[\langle V\rangle]]\!]}$

By the fact that $S_1 \Rightarrow S_2$ *is speculative,* $S_2 \equiv [\![\theta',\ E'[\langle V'\rangle]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $V \Rightarrow_s V'$. *Then, as the case (RtVl), we have* $S_2 \Rightarrow {S_1}^{\star}$.

$\underline{S_1 \equiv [\![\theta,\ E[F[(\mathcal{S}k.M)]]]\!]}$

By the fact that $S_1 \Rightarrow S_2$ *is speculative,* $S_2 \equiv [\![\theta',\ E'[F'[(\mathcal{S}k.M')]]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ *and* $M \Rightarrow_s M'$. *Then, as the case (RtSt), we have* $S_2 \Rightarrow {S_1}^{\star}$.

$\underline{S_1 \equiv [\![\theta,\ E[(\textbf{make}\ V)]]\!]}$

By the fact that $S_1 \Rightarrow S_2$ *is speculative,* $S_2 \equiv [\![\theta',\ E'[(\texttt{make}\ V')]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *Then, as the case (Make), we have* $S_2 \Rightarrow {S_1}^{\star}$.

$\underline{S_1 \equiv [\![\theta \cup \{b \mapsto V_1\},\ E[((\textbf{set!}\ b)\ V)]]\!]}$

By the fact that $S_1 \Rightarrow S_2$ *is speculative,* $S_2 \equiv [\![\theta' \cup \{b \mapsto V_1'\},\ E'[((\texttt{set!}\ b)\ V')]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *Then, as the case (Set), we have* $S_2 \Rightarrow {S_1}^{\star}$.

$\underline{S_1 \equiv [\![\theta \cup \{b \mapsto V\},\ E[(\textbf{deref}\ b)]]\!]}$

By the fact that $S_1 \Rightarrow S_2$ *is speculative,* $S_2 \equiv [\![\theta' \cup \{b \mapsto V\},\ E[(\texttt{deref}\ b)]]\!]$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$ *and* $V \Rightarrow_s V'$. *Then, as the case (Deref), we have* $S_2 \Rightarrow {S_1}^{\star}$.

$S_1 \equiv [\![\theta,\ E[F[(\mathbf{future}\ M)]]]\!]$

By the fact that $S_1 \Rightarrow S_2$ is speculative, $S_2 \equiv [\![\theta',\ E'[F'[(\texttt{future}\ M')]]]\!]$ where $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ and $M \Rightarrow_s M'$. Then, as the case (Fork), we have $S_2 \Rightarrow {S_1}^\star$.

$S_1 \equiv [\![\theta,\ E[\langle \mathbf{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!]\rangle]]\!]$

By the fact that $S_1 \Rightarrow S_2$ is speculative, $S_2 \equiv [\![\theta',\ E'[\langle \texttt{flet}\ (p\ V')\ [\![\theta_2,\ M_2]\!]\rangle]]\!]$ where $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $\Rightarrow_s V'$ and $[\![\theta_1,\ M_1]\!] \Rightarrow [\![\theta_2,\ M_2]\!]$. Then, as the case (Fork), we have $S_2 \Rightarrow {S_1}^\star$.

otherwise

By the definition of $\Rightarrow$, $S_1 \equiv [\![\theta,\ M]\!]$ and $S_2 \equiv [\![\theta',\ M']\!]$ where $\theta \Rightarrow_s \theta'$ and $M \Rightarrow_s M'$. By I.H. of (4'), $\theta' \Rightarrow_s \theta^\#$ and $M' \Rightarrow_s M^\#$. Then, $S_2 \Rightarrow [\![\theta^\#,\ M^\#]\!] \equiv {S_1}^\star$.

Lemma 5.11 (4')

We prove that X_1 is a term only because the case that X_1 is a context can be proved in a similar way, and the case that X_1 is a store can be shown from the case that X_1 is a term.

We will show the case analysis on the structure of X_1, namely, a term.

$X_1 \equiv x, c, b, p$

${X_1}^\#$ is just that so $X_1 = X_2 = {X_1}^\#$.

$X_1 \equiv (\lambda x.M_1)$

From $X_1 \Rightarrow_s X_2$, $X_2 \equiv (\lambda x.M_1')$ where $M_1 \Rightarrow_s M_1'$. By I.H., $M_1' \Rightarrow_s {M_1}^\#$, which implies $(\lambda x.M_1') \Rightarrow_s (\lambda x.{M_1}^\#)$.

$X_1 \equiv (M_1\ M_2)$

From $X_1 \Rightarrow_s X_2$, $X_2 \equiv (M_1'\ M_2')$ where $M_1 \Rightarrow_s M_1'$ and $M_2 \Rightarrow_s M_2'$. By I.H., $M_1' \Rightarrow_s {M_1}^\#$ and $M_2' \Rightarrow_s {M_2}^\#$, which implies $(M_1'\ M_2') \Rightarrow_s ({M_1}^\#\ {M_2}^\#)$.

$X_1 \equiv \langle M_1 \rangle$

From $X_1 \Rightarrow_s X_2$, $X_2 \equiv \langle M_1' \rangle$ where $M_1 \Rightarrow_s M_1'$. By I.H., $M_1' \Rightarrow_s {M_1}^\#$, which implies $\langle M_1' \rangle \Rightarrow_s \langle {M_1}^\# \rangle$.

$X_1 \equiv (\mathcal{S}k.M_1)$

From $X_1 \Rightarrow_s X_2$, $X_2 \equiv (\mathcal{S}k.M_1')$ where $M_1 \Rightarrow_s M_1'$. By I.H., $M_1' \Rightarrow_s {M_1}^\#$, which implies $(\mathcal{S}k.M_1') \Rightarrow_s (\mathcal{S}k.{M_1}^\#)$.

$X_1 \equiv (\mathbf{future}\ M_1)$

From $X_1 \Rightarrow_s X_2$, $X_2 \equiv (\texttt{future}\ M_1')$ where $M_1 \Rightarrow_s M_1'$. By I.H., $M_1' \Rightarrow_s {M_1}^\#$, which implies $(\texttt{future}\ M_1') \Rightarrow_s (\texttt{future}\ {M_1}^\#)$.

■

Proof (Lemma 5.11 (5), (5'))**.** *These can be proved by revising the proof of (4) and (4') precisely so we show several cases of (5) and (5') only.*

Lemma 5.11 (5)

(*App*)

By the assumption, $S_1 \equiv [\![\theta,\ E[((\lambda x.M)\ V)]]\!]$ and $S_2 \equiv [\![\theta',\ E'[M'\{x := V'\}]]\!]$ where $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $M \Rightarrow_s M'$ and $V \Rightarrow_s V'$, and $S_2 \Rightarrow {S_1}^\star \equiv [\![\theta^\#,\ E^\#[M^\#\{x := V^\#\}]]\!]$. By this fact, we can find that $S_2 \Rightarrow {S_1}^\star$ is speculative because this parallel transition is by (Spec).

(*Spec*)

By the assumption, $S_1 \equiv [\![\theta,\ E[F[(\texttt{future}\ M)]]]\!]$ and $S_2 \equiv [\![\theta',\ E'[\langle \texttt{flet}\ (p\ M')\ [\![\emptyset,\ F'[p]]\!]\rangle]]\!]$ where $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ and $M \Rightarrow_s M'$, and

$$S_2 \equiv [\![\theta,\ E[F[(\texttt{future}\ M)]]]\!] \Rightarrow [\![\theta^\#,\ E^\#[\langle \texttt{flet}\ (p\ M^\#)\ [\![\emptyset,\ F^\#[p]]\!]\rangle]]\!] \equiv {S_1}^\star.$$

Also the above case, the transition $S_2 \Rightarrow {S_1}^\star$ is speculative because the transition is by (Spec).

Lemma 5.11 (5')

The cases that $S_1 \Rightarrow S_2$ is speculative and $S_1 \Rightarrow {S_1}^\star$ is mandatory correspond to the ones which are shown in the case (Spec) in the proof of (4).

$S_1 \Rightarrow S_2$ **is (*Spec*) and** $S_1 \Rightarrow S_1{}^\star$ **is (*App*)**

We can put that $S_1 \equiv \langle\!|\theta,\ E[((\lambda x.M)\ V)]|\!\rangle$ *and* $S_2 \equiv \langle\!|\theta',\ E'[((\lambda x.M')\ V')]|\!\rangle$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $M \Rightarrow_s M'$ *and* $V \Rightarrow_s V'$, *and* $S_1{}^\star \equiv \langle\!|\theta^\#,\ E^\#[M^\#\{x := V^\#\}]|\!\rangle$. *Then, it is clear that the transition* $S_2 \Rightarrow S_1{}^\star$ *is by (App).*

$S_1 \Rightarrow S_2$ **is (*Spec*) and** $S_1 \Rightarrow S_1{}^\star$ **is (*Fork*)**

We can put that $S_1 \equiv \langle\!|\theta,\ E[F[(\texttt{future}\ M)]]|\!\rangle$ *and* $S_2 \equiv \langle\!|\theta',\ E'[F'[(\texttt{future}\ M')]]|\!\rangle$ *where* $\theta \Rightarrow_s \theta'$, $E \Rightarrow_s E'$, $F \Rightarrow_s F'$ *and* $M \Rightarrow_s M'$, *and* $S_1{}^\star \equiv \langle\!|\theta^\#,\ E^\#[\langle \texttt{flet}\ (p\ M^\#)\ \langle\!|\emptyset,\ F^\#[p]|\!\rangle\rangle]|\!\rangle$. *Then, it is clear that the transition* $S_2 \Rightarrow S_1{}^\star$ *is by (Fork).*

■

Finally, we will prove Theorem 5.5.

Proof (Lemma 5.5)**.** *If we assume that* $S_1 \rightarrow^* S_2$ *and* $S_1 \rightarrow^* S_3$, *we have that* $S_1 \Rightarrow^* S_2$ *and* $S_1 \Rightarrow^* S_3$ *by Lemma 5.11 (1). Let* k *be the max number of the length of these transition sequence. Then, if we assume that* S_4 *is a state which can be obtained by applying the maximum-parallel transition* $(-)^\star$ *for* k *times, we have* $S_2 \Rightarrow^* S_4$ *and* $S_3 \Rightarrow^* S_4$ *by Lemma 5.11. Therefore, by Lemma 5.11 (2), we can find that* $\rightarrow$ *is confluent.* ■

5.4.3 Uniqueness of Transitions in AM-PFSR

In this subsection, we will show the following theorem for the final step to prove the theorem 5.2.

Theorem 5.6. *If* $A_1, A_2 \in eval_p(P)$, $A_1 = A_2$ □

To prove Theorem 5.6, we will have to prove several lemmas.

Lemma 5.15. *For AM-PFSR, we have the following properties.*

(1) If $S_1 \rightarrow S_2$ *is speculative and* $S_2 \rightarrow S_3$ *is mandatory transition, there exists a transition sequence* $S_1 \rightarrow^* S_3$ *whose first transition is mandatory.*

(2) If there is an infinite transition sequence which starts from S_1 *and contains infinite mandatory transitions, there exists a transition sequence which consists of mandatory transitions only.*

(3) There is no transition sequence which consists of (join) transition only.

□

We will re-define a relation $\rightarrow_{spec}$ like the relation which is defined in the previous subsection.

Def 5.14. *(1)* $M \rightarrow_{spec} M'$ *if and only if* $\langle\!|\theta,\ M|\!\rangle \rightarrow \langle\!|\theta,\ M'|\!\rangle$ *for an arbitrary* θ *by (spec),*

(2) $E \rightarrow_{spec} E'$ *if and only if* $\langle\!|\theta,\ E[M]|\!\rangle \rightarrow \langle\!|\theta,\ E'[M]|\!\rangle$ *for an arbitrary* θ *and* M *by (spec),*

(3) $\theta \rightarrow_{spec} \theta'$ *if and only if* $\langle\!|\theta,\ M|\!\rangle \rightarrow \langle\!|\theta',\ M|\!\rangle$ *for an arbitrary* M *by (store).*

Then, we will represent the reflexive and transitive closure of $\rightarrow_{spec}$ as $\rightarrow^*_{spec}$.

Lemma 5.16. *(1) If* $V \rightarrow_{spec} V'$, $C[M\{x := V\}] \rightarrow^*_{spec} C[M\{x := V'\}]$ *for an arbitrary* C,

(2) If $M \rightarrow_{spec} M'$, $M\{x := V\} \rightarrow_{spec} M'\{x := V\}$,

(3) If $E \rightarrow_{spec} E'$, $E\{x := V\} \rightarrow_{spec} E'\{x := V\}$,

(4) If $V \rightarrow_{spec} V'$, $\theta\{x := V\} \rightarrow^*_{spec} \theta\{x := V'\}$.

Proof . *(1) Proof by induction on the structure of* M.

$M \equiv c, b$

These are trivial.

$M \equiv y(x \neq y)$ **or** $M \equiv p(x \neq p)$

Since $M\{x := V\} \equiv M$, *this case is also trivial.*

$\underline{M \equiv y(x = y)}$ **or** $\underline{M \equiv p(x = p)}$

Since $M\{x := V\} \equiv V$ *and* $M\{x := V'\} \equiv V'$*, we have* $C[M\{x := V\}] \equiv C[V] \rightarrow_{spec} C[V'] \equiv C[M\{x := V'\}]$.

$\underline{M \equiv (\lambda y.M_1)(y \neq x)}$

Since $M\{x := V\} \equiv (\lambda y.M_1\{x := V\})$ *and* $M\{x := V'\} \equiv (\lambda y.M_1\{x := V'\})$*, we have the following:*

$$\begin{aligned} C[M\{x := V\}] &\equiv C[(\lambda y.M_1\{x := V\})] \\ &\rightarrow^*_{spec} C[(\lambda y.M_1\{x := V'\})] && \textit{by I.H.} \\ &\equiv C[(\lambda y.M_1)\{x := V'\}] \\ &\equiv C[M\{x := V'\}] \end{aligned}$$

$\underline{M \equiv (\lambda y.M_1)(y = x)}$

Since $M\{x := V\} \equiv (\lambda y.M_1) \equiv M$ *and* $M\{x := V'\} \equiv (\lambda y.M_1) \equiv M$*, this case is also trivial.*

$\underline{M \equiv (M_1\ M_2)}$

Since $M\{x := V\} \equiv (M_1\{x := V\}\ M_2\{x := V\})$ *and* $M\{x := V'\} \equiv (M_1\{x := V'\}\ M_2\{x := V'\})$*, we have the following:*

$$\begin{aligned} C[M\{x := V\}] &\equiv C[(M_1\{x := V\}\ M_2\{x := V\})] \\ &\rightarrow^*_{spec} C[(M_1\{x := V'\}\ M_2\{x := V\})] && \textit{by I.H.} \\ &\rightarrow^*_{spec} C[(M_1\{x := V'\}\ M_2\{x := V'\})] && \textit{by I.H.} \\ &\equiv C[(M_1\ M_2)\{x := V'\}] \\ &\equiv C[M\{x := V'\}] \end{aligned}$$

$\underline{M \equiv \langle M_1 \rangle}$

Since $M\{x := V\} \equiv \langle M_1\{x := V\}\rangle$ *and* $M\{x := V'\} \equiv \langle M_1\{x := V'\}\rangle$*, we have the following:*

$$\begin{aligned} C[M\{x := V\}] &\equiv C[\langle M_1\{x := V\}\rangle] \\ &\rightarrow^*_{spec} C[\langle M_1\{x := V'\}\rangle] && \textit{by I.H.} \\ &\equiv C[\langle M_1\rangle\{x := V'\}] \\ &\equiv C[M\{x := V'\}] \end{aligned}$$

$\underline{M \equiv (\mathcal{S}k.M_1)(k \neq x)}$

Since $M\{x := V\} \equiv (\mathcal{S}k.M_1\{x := V\})$ *and* $M\{x := V'\} \equiv (\mathcal{S}k.M_1\{x := V'\})$*, we have the following:*

$$\begin{aligned} C[M\{x := V\}] &\equiv C[(\mathcal{S}k.M_1\{x := V\})] \\ &\rightarrow^*_{spec} C[(\mathcal{S}k.M_1\{x := V'\})] && \textit{by I.H.} \\ &\equiv C[(\mathcal{S}k.M_1)\{x := V'\}] \\ &\equiv C[M\{x := V'\}] \end{aligned}$$

$\underline{M \equiv (\mathcal{S}k.M_1)(k = x)}$

Since $M\{x := V\} \equiv (\mathcal{S}k.M_1) \equiv M$ *and* $M\{x := V'\} \equiv (\mathcal{S}k.M_1) \equiv M$*, this case is also trivial.*

$\underline{M \equiv (\textbf{future}\ M_1)}$

Since $M\{x := V\} \equiv (\texttt{future}\ M_1\{x := V\})$ *and* $M\{x := V'\} \equiv (\texttt{future}\ M_1\{x := V'\})$*, we have the following:*

$$\begin{aligned} C[M\{x := V\}] &\equiv C[(\texttt{future}\ M_1\{x := V\})] \\ &\rightarrow^*_{spec} C[(\texttt{future}\ M_1\{x := V'\})] && \textit{by I.H.} \\ &\equiv C[(\texttt{future}\ M_1)\{x := V'\}] \\ &\equiv C[M\{x := V'\}] \end{aligned}$$

$\underline{M \equiv \langle \textbf{flet}\ (p\ M_1)\ [\![\theta,\ M_2]\!]\rangle, (x \neq p)}$

Since $M\{x := V\} \equiv \langle \texttt{flet}\ (p\ M_1\{x := V\})\ [\![\theta\{x := V\},\ M_2\{x := V\}]\!]\rangle$ *and* $M\{x := V\} \equiv \langle \texttt{flet}\ (p\ M_1\{x := V'\})\ [\![\theta\{x := V'\},\ M_2\{x := V'\}]\!]\rangle$*, we have the following:*

$$\begin{aligned}
C[M\{x := V\}] &\equiv C[\langle \texttt{flet}\ (p\ M_1\{x := V\})\ [\![\theta\{x := V\},\ M_2\{x := V\}]\!]\rangle] \\
&\to^*_{spec} C[\langle \texttt{flet}\ (p\ M_1\{x := V'\})\ [\![\theta\{x := V\},\ M_2\{x := V\}]\!]\rangle] && \textit{by I.H.} \\
&\to^*_{spec} C[\langle \texttt{flet}\ (p\ M_1\{x := V'\})\ [\![\theta\{x := V'\},\ M_2\{x := V\}]\!]\rangle] && \textit{by I.H. of (4)} \\
&\to^*_{spec} C[\langle \texttt{flet}\ (p\ M_1\{x := V'\})\ [\![\theta\{x := V'\},\ M_2\{x := V'\}]\!]\rangle] && \textit{by I.H.} \\
&\equiv C[\langle \texttt{flet}\ (p\ M_1)\ [\![\theta,\ M_2]\!]\rangle\{x := V'\}] \\
&\equiv C[M\{x := V'\}]
\end{aligned}$$

$\underline{M \equiv \langle \textbf{flet}\ (p\ M_1)\ [\![\theta,\ M_2]\!]\rangle, (x = p)}$

Since $M\{x := V\} \equiv \langle \texttt{flet}\ (p\ M_1\{x := V\})\ [\![\theta,\ M_2]\!]\rangle$ *and* $M\{x := V\} \equiv \langle \texttt{flet}\ (p\ M_1\{x := V'\})\ [\![\theta,\ M_2]\!]\rangle$*, we have the following.*

$$\begin{aligned}
C[M\{x := V\}] &\equiv C[\langle \texttt{flet}\ (p\ M_1\{x := V\})\ [\![\theta\{x := V\},\ M_2\{x := V\}]\!]\rangle] \\
&\to^*_{spec} C[\langle \texttt{flet}\ (p\ M_1\{x := V'\})\ [\![\theta\{x := V\},\ M_2\{x := V\}]\!]\rangle] && \textit{by I.H.} \\
&\equiv C[\langle \texttt{flet}\ (p\ M_1)\ [\![\theta,\ M_2]\!]\rangle\{x := V'\}] \\
&\equiv C[M\{x := V'\}]
\end{aligned}$$

(2) This can be proved almost same as the above one.

(3) This is also same as the above.

(4) Let θ *be* $\{b_1 \mapsto V_1, \cdots, b_n \mapsto V_n\}$*. Since* $\theta\{x := V\} \equiv \{b_1 \mapsto V_1\{x := V\}, \cdots, b_n \mapsto V_n\{x := V\}\}$*, we have the following.*

$$\begin{aligned}
\theta\{x := V\} &\equiv \{b_1 \mapsto V_1\{x := V\}, \cdots, b_n \mapsto V_n\{x := V\}\} \\
&\to^*_{spec} \{b_1 \mapsto V_1\{x := V'\}, \cdots, b_n \mapsto V_n\{x := V\}\} && \textit{by I.H. of (1)} \\
&\cdots \\
&\to^*_{spec} \{b_1 \mapsto V_1\{x := V'\}, \cdots, b_n \mapsto V_n\{x := V'\}\} && \textit{by I.H. of (1)} \\
&\equiv \theta\{x := V'\}
\end{aligned}$$

■

Proof (Lemma 5.15(1))**.** *If a transition* $S_1 \to S_2$ *is speculative and* $S_2 \to S_3$ *is mandatory, we can apply the mandatory transition rule to* S_1*. So we will analyze the all cases of* $S_2 \to S_3$*.*

(app)

$S_2 \equiv [\![\theta,\ E[((\lambda x.M)\ V)]]\!]$ *and* $S_3 \equiv [\![\theta,\ E[M\{x := V\}]]\!]$*. Since* S_1 *is in the same form* S_2 *and the transition* $S_1 \to S_2$ *is speculative, there are four possibilities of the target of the transition:* $\theta' \to_f \theta$*,* $E' \to_f E$*,* $M' \to_f M$ *or* $V' \to_f V$ *where* X' *is a part of* S_1 *corresponding to the part of* S_2*.*

$\underline{\theta' \to_f \theta'}$

Since $S_1 \equiv [\![\theta',\ E[((\lambda x.M)\ V)]]\!]$*, we have the following sequence.*

$$\begin{aligned}
S_1 &\equiv [\![\theta',\ E[((\lambda x.M)\ V)]]\!] \\
&\to [\![\theta',\ E[M\{x := V\}]]\!] && \textit{(app)} \\
&\to [\![\theta,\ E[M\{x := V\}]]\!] && \textit{(store)} \\
&\to S_3
\end{aligned}$$

$\underline{E' \to_f E}$

Since $S_1 \equiv \langle\!\langle \theta,\ E'[((\lambda x.M)\ V)] \rangle\!\rangle$*, we have the following sequence.*

$$\begin{aligned} S_1 &\equiv \langle\!\langle \theta,\ E'[((\lambda x.M)\ V)] \rangle\!\rangle \\ &\to \langle\!\langle \theta,\ E'[M\{x := V\}] \rangle\!\rangle && \textit{(app)} \\ &\to \langle\!\langle \theta,\ E[M\{x := V\}] \rangle\!\rangle && \textit{(spec)} \\ &\to S_3 \end{aligned}$$

$\underline{M' \to_f M}$

Since $S_1 \equiv \langle\!\langle \theta,\ E[((\lambda x.M')\ V)] \rangle\!\rangle$*, we have the following sequence.*

$$\begin{aligned} S_1 &\equiv \langle\!\langle \theta,\ E[((\lambda x.M')\ V)] \rangle\!\rangle \\ &\to \langle\!\langle \theta,\ E[M'\{x := V\}] \rangle\!\rangle && \textit{(app)} \\ &\to \langle\!\langle \theta,\ E[M\{x := V\}] \rangle\!\rangle && \textit{(spec) with the lemma 5.16 (2)} \\ &\to S_3 \end{aligned}$$

$\underline{V' \to_f V}$

Since $S_1 \equiv \langle\!\langle \theta,\ E[((\lambda x.M)\ V')] \rangle\!\rangle$*, we have the following sequence.*

$$\begin{aligned} S_1 &\equiv \langle\!\langle \theta,\ E[((\lambda x.M)\ V')] \rangle\!\rangle \\ &\to \langle\!\langle \theta,\ E[M\{x := V'\}] \rangle\!\rangle && \textit{(app)} \\ &\to^* \langle\!\langle \theta,\ E[M\{x := V\}] \rangle\!\rangle && \textit{(spec) with the lemma 5.16 (1)} \\ &\to S_3 \end{aligned}$$

(*rv*)

This case is almost clear because we can show this case as the (app) case.

(*rs*)

$S_2 \equiv \langle\!\langle \theta,\ E[F[(\mathcal{S}k.M)]] \rangle\!\rangle$ *and* $S_3 \equiv \langle\!\langle \theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle] \rangle\!\rangle$*. Since* S_1 *is in the same form* S_2 *and the transition* $S_1 \to S_2$ *is speculative, there are four possibilities of the target of the transition:* $\theta' \to_f \theta$*,* $E' \to_f E$*,* $F' \to_f F$ *or* $M' \to_f M$ *where each* X' *is a part of* S_1 *corresponding to the part of* S_2*.*

$\underline{\theta' \to_f \theta'}$

Since $S_1 \equiv \langle\!\langle \theta',\ E[F[(\mathcal{S}k.M)]] \rangle\!\rangle$*, we have the following sequence.*

$$\begin{aligned} S_1 &\equiv \langle\!\langle \theta',\ E[F[(\mathcal{S}k.M)]] \rangle\!\rangle \\ &\to \langle\!\langle \theta',\ E[\langle M\{k := (\lambda v.F[v])\}\rangle] \rangle\!\rangle && \textit{(rs)} \\ &\to \langle\!\langle \theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle] \rangle\!\rangle && \textit{(store)} \\ &\to S_3 \end{aligned}$$

$\underline{E' \to_f E}$

Since $S_1 \equiv \langle\!\langle \theta,\ E'[F[(\mathcal{S}k.M)]] \rangle\!\rangle$*, we have the following sequence.*

$$\begin{aligned} S_1 &\equiv \langle\!\langle \theta,\ E'[F[(\mathcal{S}k.M)]] \rangle\!\rangle \\ &\to \langle\!\langle \theta,\ E'[\langle M\{k := (\lambda v.F[v])\}\rangle] \rangle\!\rangle && \textit{(rs)} \\ &\to \langle\!\langle \theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle] \rangle\!\rangle && \textit{(spec)} \\ &\to S_3 \end{aligned}$$

$\underline{F' \to_f F}$

Since $S_1 \equiv \langle\!\langle \theta,\ E[F'[(\mathcal{S}k.M)]] \rangle\!\rangle$*, we have the following sequence.*

$$\begin{aligned} S_1 &\equiv \langle\!\langle \theta,\ E[F'[(\mathcal{S}k.M)]] \rangle\!\rangle \\ &\to \langle\!\langle \theta,\ E[\langle M\{k := (\lambda v.F'[v])\}\rangle] \rangle\!\rangle && \textit{(rs)} \\ &\to^* \langle\!\langle \theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle] \rangle\!\rangle && \textit{(spec) with the lemma 5.16 (1)} \\ &\to S_3 \end{aligned}$$

$\underline{M' \to_f M}$

Since $S_1 \equiv [\![\theta,\ E[F[(\mathcal{S}k.M')]]]\!]$*, we have the following sequence.*

$$\begin{aligned}
S_1 &\equiv [\![\theta,\ E[F[(\mathcal{S}k.M')]]]\!] \\
&\to [\![\theta,\ E[\langle M'\{k := (\lambda v.F[v])\}\rangle]]\!] && \textit{(rs)} \\
&\to [\![\theta,\ E[\langle M\{k := (\lambda v.F[v])\}\rangle]]\!] && \textit{(spec) with the lemma 5.16 (2)} \\
&\to S_3
\end{aligned}$$

(*make*)

$S_2 \equiv [\![\theta,\ E[(\texttt{make}\ V)]]\!]$ *and* $S_3 \equiv [\![\theta \cup \{b \mapsto V\},\ E[b]]\!]$ *where* b *is fresh. Since* S_1 *is in the same form of* S_2 *and the transition* $S_1 \to S_2$ *is speculative, there are four possibilities of the target of the transition:* $\theta' \to_f \theta$*,* $E' \to_f E$ *or* $V' \to_f V$ *where each* X' *is a part of* S_1 *corresponding to the part of* S_2*.*

$\underline{\theta' \to_f \theta'}$

Since $S_1 \equiv [\![\theta',\ E[(\texttt{make}\ V)]]\!]$*, we have the following sequence.*

$$\begin{aligned}
S_1 &\equiv [\![\theta',\ E[(\texttt{make}\ V)]]\!] \\
&\to [\![\theta' \cup \{b \mapsto V\},\ E[b]]\!] && \textit{(make)} \\
&\to [\![\theta \cup \{b \mapsto V\},\ E[b]]\!] && \textit{(store)} \\
&\to S_3
\end{aligned}$$

$\underline{E' \to_f E}$

Since $S_1 \equiv [\![\theta,\ E'[(\texttt{make}\ V)]]\!]$*, we have the following sequence.*

$$\begin{aligned}
S_1 &\equiv [\![\theta,\ E'[(\texttt{make}\ V)]]\!] \\
&\to [\![\theta \cup \{b \mapsto V\},\ E'[b]]\!] && \textit{(make)} \\
&\to [\![\theta \cup \{b \mapsto V\},\ E[b]]\!] && \textit{(store)} \\
&\to S_3
\end{aligned}$$

$\underline{V' \to_f V}$

Since $S_1 \equiv [\![\theta,\ E[(\texttt{make}\ V')]]\!]$*, we have the following sequence.*

$$\begin{aligned}
S_1 &\equiv [\![\theta,\ E[(\texttt{make}\ V')]]\!] \\
&\to [\![\theta \cup \{b \mapsto V'\},\ E[b]]\!] && \textit{(make)} \\
&\to [\![\theta \cup \{b \mapsto V\},\ E[b]]\!] && \textit{(store)} \\
&\to S_3
\end{aligned}$$

(*set*) and (*deref*)

These two cases can be shown almost same as the (make) case.

(*fork*)

$S_2 \equiv [\![\theta,\ E[F[(\texttt{future}\ M)]]]\!]$ *and* $S_3 \equiv [\![\theta,\ E[\langle \texttt{flet}\ (p\ M)\ [\![\emptyset,\ F[p]]\!]\rangle]]\!]$ *where* p *is a fresh communication variable. Since* S_1 *has the same form of* S_2 *and the transition* $S_1 \to S_2$ *is speculative, there are four possibilities of the target of the transition:* $\theta' \to_f \theta$*,* $E' \to_f E$*,* $F' \to_f F$ *or* $M' \to_f M$ *where each* X' *is a part of* S_1 *corresponding to the part of* S_2*. All the cases can be shown as the case (app)*

(*join*)

$S_2 \equiv [\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta,\ \theta_1]\!]M_1\rangle]]\!]$ *and* $S_3 \equiv [\![\theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}]]\!]$*. Since* S_1 *has the same form of* S_2 *and the transition* $S_1 \to S_2$ *is speculative, there are four possibilities of the target of the transition:* $\theta' \to_f \theta$*,* $E' \to_f E$*,* $V' \to_f V$ *or* $[\![\theta_2,\ M_2]\!] \to [\![\theta_1,\ M_1]\!]$ *where each* X' *is a part of* S_1 *corresponding to the part of* S_2*. Since the case* $\theta' \to_f \theta$ *and* $E' \to_f E$ *can be shown as the above cases, we will show the rest two cases.*

$\underline{V' \to_f V}$

By Lemma 5.16 (4), $\theta_1\{p := V'\} \to_f \theta_1\{p := V\}$, and by Lemma 5.16 (1), $M_1\{p := V'\} \to_f M_1\{p := V\}$. Then, we have the following sequence.

$$\begin{aligned}
S_1 &\equiv [\![\theta,\ E[\langle \texttt{flet}\ (p\ V')\ [\![\theta,\ \theta_1]\!] M_1\rangle]]\!] \\
&\to [\![\theta \cup (\theta_1\{p := V'\}),\ E[M_1\{p := V'\}]]\!] && \textit{(join)} \\
&\to^* [\![\theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V'\}]]\!] && \textit{(spec) with the lemma 5.16 (4)} \\
&\to^* [\![\theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}]]\!] && \textit{(spec) with the lemma 5.16 (1)} \\
&\equiv S_3
\end{aligned}$$

$\underline{[\![\theta_2,\ M_2]\!] \to [\![\theta_1,\ M_1]\!]}$

In this case, we will use the property such that if $S \to S'$ then $S\{p := V\} \to S\{p := V\}$, which can be shown easily by the induction. Then, we have the following sequence.

$$\begin{aligned}
S_1 &\equiv [\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta,\ \theta_2]\!] M_2\rangle]]\!] \\
&\to [\![\theta \cup (\theta_2\{p := V\}),\ E[M_2\{p := V\}]]\!] \\
&\to [\![\theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}]]\!] \\
&\equiv S_2
\end{aligned}$$

■

Proof (Lemma 5.15(2))**.** *Assume that there is an infinite transition sequence starting with S_1 and its first mandatory transition is $S_{k+1} \to S_{k+2}$. (Namely, its first k transitions are speculative)*

By applying the clause (1) of Lemma 5.15 to the sequence k times, we obtain a finite transition sequence $S_1 \to^* S_{k+2}$, whose first transition is mandatory. By appending the suffix of the given infinite sequence starting with S_{k+2} to the obtained one, we get an infinite sequence from S_1 whose first transition is mandatory.

By repeating this process n times, we get, for each n, an infinite sequence from S_1 whose first n transitions are mandatory. Moreover, the n-th infinite sequence and the $n+1$-st one share the first $n+1$ states and the first n transitions all of which are mandatory.

Let S^i_j be the j-th state of the i-th infinite sequence (for $i, j \geq 1$), and we put $T_i \equiv S^i_i$ for $i \leq 1$. Then we have $T_1 \equiv S_1$ and $T_i \to T_{i+1}$ is a mandatory transition, and therefore we have obtained the desired infinite transition sequence. ■

Next, we will define a new relation $|X|$ and prove Lemma 5.17 in order to show Lemma 5.15 (3).

Def 5.15. *We will define a natural number $|X|$ as follows. (In the following, we define $|M|$ and $|C|$ simultaneously so we assume that meta variable M denotes terms or contexts.)*

$$\begin{aligned}
|v| &\stackrel{\text{def}}{=} 0 && v = c, x, p, b, \Box \\
|(\lambda x.M)| &\stackrel{\text{def}}{=} 0 \\
|(M_1\ M_2)| &\stackrel{\text{def}}{=} |M_1| + |M_2| \\
|\langle M\rangle| &\stackrel{\text{def}}{=} |M| \\
|(\mathcal{S}k.M)| &\stackrel{\text{def}}{=} |M| \\
|(\texttt{future}\ M)| &\stackrel{\text{def}}{=} |M| \\
|\langle \texttt{flet}\ (p_i\ M)\ S\rangle| &\stackrel{\text{def}}{=} |M| + |S| + 1 \\
|[\![\theta,\ M]\!]| &\stackrel{\text{def}}{=} |M|
\end{aligned}$$

Lemma 5.17. *(1)* $|E[M]| = |E| + |M|$

(2) $|M\{p_i := V\}| = |M|$

Proof . *(1) Proof by induction on the structure of E.*

$\underline{E \equiv \square}$

This case is trivial so we omit.

$\underline{E \equiv (E_1\ M_1)}$

We have the following.

$$\begin{aligned}
|E[M]| &\equiv |(E_1[M]\ M_1)| \\
&= |E_1[M]| + |M_1| && \textit{by definition} \\
&= |E_1| + |M| + |M_1| && \textit{by I.H.} \\
&= |(E_1\ M_1)| + |M| \\
&= |E| + |M|
\end{aligned}$$

$\underline{E \equiv (V\ E_1)}$

We have the following.

$$\begin{aligned}
|E[M]| &\equiv |(V\ E_1[M])| \\
&= |V| + |E_1[M]| && \textit{by definition} \\
&= |V| + |E_1| + |M| && \textit{by I.H.} \\
&= |(V\ E_1)| + |M| \\
&= |E| + |M|
\end{aligned}$$

$\underline{E \equiv \langle E_1 \rangle}$

We have the following.

$$\begin{aligned}
|E[M]| &\equiv |\langle E_1[M] \rangle| \\
&= |E_1[M]| && \textit{by definition} \\
&= |E_1| + |M| && \textit{by I.H.} \\
&= |\langle E_1 \rangle| + |M| \\
&= |E| + |M|
\end{aligned}$$

$\underline{E \equiv \langle \texttt{flet}\ (p_i\ E_1)\ S \rangle}$

We have the following.

$$\begin{aligned}
|E[M]| &\equiv |\langle \texttt{flet}\ (p_i\ E_1[M])\ S \rangle| \\
&= |E_1[M]| + |S| + 1 && \textit{by definition} \\
&= |E_1| + |M| + |S| + 1 && \textit{by I.H.} \\
&= |\langle \texttt{flet}\ (p_i\ E_1)\ S \rangle| + |M| \\
&= |E| + |M|
\end{aligned}$$

(2) Proof by induction on the structure of M.

$\underline{M \equiv x, c, p, b, (\lambda x.M)}$

This case is trivial because $M\{p := V\}$ must become a value.

$\underline{M \equiv (M_1\ M_2)}$

We have the following.

$$\begin{aligned}
|M\{p := V\}| &\equiv |(M_1\ M_2)\{p := V\}| \\
&= |(M_1\{p := V\}\ M_2\{p := V\})| \\
&= |M_1\{p := V\}| + |M_2\{p := V\}| && \textit{by def} \\
&= |M_1| + |M_2| && \textit{by I.H.} \\
&\equiv |(M_1\ M_2)|
\end{aligned}$$

$\underline{M \equiv \langle M_1 \rangle}$
We have the following.

$$\begin{aligned}
|M\{p := V\}| &\equiv |\langle M_1 \rangle \{p := V\}| \\
&= |\langle M_1\{p := V\} \rangle| \\
&= |M_1\{p := V\}| && \textit{by def} \\
&= |M_1| && \textit{by I.H.} \\
&\equiv |\langle M_1 \rangle|
\end{aligned}$$

$\underline{M \equiv (\mathcal{S}k.M_1)}$
We have the following.

$$\begin{aligned}
|M\{p := V\}| &\equiv |(\mathcal{S}k.M_1)\{p := V\}| \\
&= |(\mathcal{S}k.M_1\{p := V\})| \\
&= |M_1\{p := V\}| && \textit{by def} \\
&= |M_1| && \textit{by I.H.} \\
&\equiv |(\mathcal{S}k.M_1)|
\end{aligned}$$

$\underline{M \equiv (\textbf{future}\ M_1)}$
We have the following.

$$\begin{aligned}
|M\{p := V\}| &\equiv |(\texttt{future}\ M_1)\{p := V\}| \\
&= |(\texttt{future}\ M_1\{p := V\})| \\
&= |M_1\{p := V\}| && \textit{by def} \\
&= |M_1| && \textit{by I.H.} \\
&\equiv |(\texttt{future}\ M_1)|
\end{aligned}$$

$\underline{M \equiv \langle \textbf{flet}\ (p\ M_1)\ [\![\theta_2,\ M_2]\!] \rangle}$
We have the following.

$$\begin{aligned}
|M\{p := V\}| &\equiv |\langle \texttt{flet}\ (p\ M_1)\ [\![\theta_2,\ M_2]\!] \rangle \{p := V\}| \\
&= |\langle \texttt{flet}\ (p\ M_1\{p := V\})\ [\![\theta_2\{p := V\},\ M_2\{p := V\}]\!] \rangle| \\
&= |M_1\{p := V\}| + |[\![\theta_2\{p := V\},\ M_2\{p := V\}]\!]| + 1 && \textit{by def} \\
&= |M_1\{p := V\}| + |M_2\{p := V\}| + 1 \\
&= |M_1| + |M_2| + 1 && \textit{by I.H.}
\end{aligned}$$

and

$$|M| \equiv |M_1|M_2$$

By the above, we have $|M\{p := V\}| = |M|$.

Proof (Lemma 5.15(3))**.** *We will show that when* $S_1 \to S_2$ *is the (join) transition, the two states always have the relation* $|S_1| > |S_2|$. *We assume that* $S_1 \equiv [\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!] \rangle]]\!]$ *and* $S_2 \equiv [\![\theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}]]\!]$, *then we have the followings.*

$$\begin{aligned}
|S_1| &\equiv |[\![\theta,\ E[\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!] \rangle]]\!]| \\
&= |E[\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!] \rangle]| && \textit{by def} \\
&= |E| + |\langle \texttt{flet}\ (p\ V)\ [\![\theta_1,\ M_1]\!] \rangle| && \textit{by the lemma 5.17 (1)} \\
&= |E| + |V| + |[\![\theta,\ M_1]\!]| + 1 && \textit{by def} \\
&= |E| + |V| + |M_1| + 1 && \textit{by def} \\
&= |E| + |M_1| + 1 && \textit{since } |V| \textit{ is always zero}
\end{aligned}$$

and

$$\begin{aligned}
|S_2| &\equiv \llbracket \theta \cup (\theta_1\{p := V\}),\ E[M_1\{p := V\}] \rrbracket \\
&= |E[M_1\{p := V\}]| && \textit{by def} \\
&= |E| + |M_1\{p := V\}| && \textit{by the lemma 5.17 (1)} \\
&= |E| + |M_1| && \textit{by the lemma 5.17 (2)}
\end{aligned}$$

Since $|X$ is a natural number, there is no infinite transition sequence consisting of the (join) transition only. ■

Lemma 5.18. *Suppose that there is an infinite transition sequence which starts with S_1 and contains infinitely many mandatory transitions. If $S_1 \to S_1'$, there is a transition sequence $S_1' \to S_2' \to \cdots$ which contains infinitely many mandatory transitions.* □

Proof . *Assume that $S_1 \to S_1'$, and there is an infinite transition sequence which starts from S_1 and contains an infinitely many mandatory transitions.*

By Lemma 5.15 (2), there is an infinite transition sequence $S_1 \to S_2 \to S_3 \to \cdots$ which contains only mandatory transitions. By Lemma 5.15 (3), the transition sequence cannot contain infinitely many consecutive (join) transitions, and therefore infinitely many mandatory transitions other than the (join) transition must be contained in it.

Here, by Lemma 5.11 (1) and (4), there is an infinite transition sequence $S_1{}^\star \Rightarrow S_2{}^\star \Rightarrow S_3{}^\star \Rightarrow \cdots$, and we have that $S_{i+1} \Rightarrow S_i{}^\star$ for any $i \geq 1$.

Let i be a natural number such that $S_i \to S_{i+1}$ be the first transition in the sequence which is not the (join) transition (i.e., $S_1 \to S_2 \to \cdots \to S_i$ are all (join) transitions). By Lemma 5.11 (5), $S_{i+1} \Rightarrow S_i{}^\star$ is a speculative transition.

By the fact that $S_{i+1} \to S_{i+2}$ is a mandatory transition, $S_{i+1} \Rightarrow S_{i+1}{}^\#$ is also a mandatory transition. Therefore, by the lemma 5.11 (5'), we have that the transition $S_i{}^\star \Rightarrow S_{i+1}{}^\star$ is a mandatory transition.

We can apply the same reasoning to the infinite transition sequence starting with S_{i+1} to show that $S_{i+1}{}^\star \Rightarrow S_{i+2}{}^\star \Rightarrow \cdots$ has a mandatory transition. Since $S_1 \to S_2 \to S_3 \Rightarrow \cdots$ contains infinitely many mandatory transitions other that (join), by iterating the above process, we can show that $S_1{}^\star \Rightarrow mxpS_2 \Rightarrow mxpS_3 \cdots$ must contain infinitely many mandatory transitions.

Finally, by the fact $S_1' \Rightarrow S_1{}^\star$ and Lemma 5.11 (2), there is an infinite transition sequence which starts from S_1' and contains infinitely many mandatory transitions. ■

Finally, we will show the target theorem of this subsection.

Proof (Theorem 5.6)**.** *We assume $A_1, A_2 \in eval_p(P)$, then there are six cases we must consider.*

$A_1 \setminus A_2$	proper answer	$\bot$	**error**
proper answer	Case 1	Case 2	Case 3
$\bot$	Case 2	Case 4	Case 5
error	Case 3	Case 5	Case 6

Table 5.1: Combination of Answers

*Case 1 is that both A_1 and A_2 are proper answer (i.e., both are not $\bot$ and **error**). Case 2 is that one of the answers is a $\bot$ and the other is a proper answer. Case 3 is that one of the answers is a **error** and the other is a proper answer. Case 4 and 6 are that the answers are both $\bot$ or **error**. And Case 5 is that one of the answers is a $\bot$ and the other is **error**.*

Case 1

By Lemma 5.5 (confluence), $A_1 = A_2$.

Case 2

We assume that A_1 is a proper answer and A_2 is $\bot$. By the assumption, there is an infinite transition sequence $\llbracket \emptyset,\ P \rrbracket \equiv S_1 \to S_2 \to \cdots$ which contains infinitely many mandatory transitions, and also there is a transition sequence $S_1 = S_1' \to S_2' \to \cdots \to S_{n+1}' = \llbracket \theta,\ V \rrbracket$ where $Unload(V) = A_1$. We will derive the contradiction of this case by the induction on the length of the latter transition sequence n.

$n = 0$

By Lemma 5.1, $S_1 = S_1' = \llbracket \emptyset,\ V \rrbracket$. Then, there is no mandatory transition form S_1 so it is inconsistent.

$\underline{n > 0}$

By Lemma 5.18, there are some infinite transition sequences like $S'_2 \to \cdots$ which contains infinitely many mandatory transitions. Since the length of the transition $S'_2 \to^* S'_{n+1} = [\![\theta,\ V]\!]$ is $n-1$, we can find that it is inconsistent by I.H. of this lemma.

Case 3

By the assumption, there are two transition sequences: the one starts from S_0 and reaches to a `stuck` state S_1, and the other starts from S_0 and reaches to a state $S_2 \equiv [\![\theta,\ V]\!]$. By Lemma 5.5, there is a state S_3 such that $S_1 \to^* S_3$ and $S_2 \to^* S_3$. Since S_1 is a `stuck` state, S_3 is also a `stuck` state but this derives the contradiction to the fact that $S_2 \to^* S_3$.

Case 4 and 6

In this case, A_1 and A_2 is trivially equal.

Case 5

We can show that this case is contradict by the same proof of Case 2.

■

5.4.4 Proof of Theorem 5.2

By the above lemmas and theorems, we can prove Theorem 5.2.

Proof (Theorem 5.2)**.** *By Theorem 5.1, $eval_s(P)$ is a set which only contains an answer $A \in \mathbb{A}$. And, by Lemma 5.1 and Theorem 5.6, $eval_p(P)$ can be denoted as $\{A'\}$ (i.e., it is a set which contains only an answer A'). Since we have $\{a\} \subseteq \{A'\}$ by Lemma 5.3, $A = A'$.* ■

5.4.5 Comparison with Other works

Flanagan and Felleisen [17] proposed a calculus which contains `future` but not any control operators, and showed confluence of `future` in their calculus. Moreau [37, 38] extended their proof technique of confluence and proved confluence of the calculus which have both `future` and `call/cc`. Moreau's work is not a straightforward extension of Flanagan and Felleisen's one, because of a technical problem explained below.

To illustrate the problem, let us assume that $S_1 \to S_2$ and $S_1 \to S_3$ hold for the following states S_1, S_2 and S_3 [4]:

$$S_1 = [\![\theta,\ \langle \texttt{flet}\ (p\ (\texttt{call/cc}\ (\lambda k.M)))\ [\![\emptyset,\ N]\!]\rangle]\!]$$
$$S_2 = [\![\theta,\ \langle \texttt{flet}\ (p\ (\texttt{call/cc}\ (\lambda k.M)))\ [\![\emptyset,\ N']\!]\rangle]\!] \text{ (where } [\![\emptyset,\ N]\!] \to [\![\emptyset,\ N']\!])$$
$$S_3 = [\![\theta,\ \langle \texttt{flet}\ (p\ M\{k := K\})\ [\![\emptyset,\ N]\!]\rangle]\!] \text{ (where } K = (\lambda v.(\texttt{abort}\ \langle \texttt{flet}\ (p\ v)\ [\![\emptyset,\ N]\!]\rangle)))$$

Here, the control operator `call/cc` is used, whose behavior can be defined as follows [5]:

$$[\![\theta,\ E[(\texttt{call/cc}\ (\lambda k.M))]]\!] \to [\![\theta,\ E[M\{k := (\lambda v.(\texttt{abort}\ E[v]))\}]]\!]$$
$$[\![\theta,\ E[(\texttt{abort}\ V)]]\!] \to [\![\theta,\ V]\!]$$

In his calculus, an expression in the body of an abstraction cannot be reduced so S_2 and S_3 cannot transit to the same state. To resolve this problem, Moreau introduced a new binary relation $\overset{\sim}{\to}_n$ which, roughly speaking, means as follows:

> When the captured continuation in M_1 can be reduced to their counter-part of M_2 by the n step reductions, then $M_1 \overset{\sim}{\to}_n M_2$.

Although this relation is not transitive, it is not a problem in Moreau's work (see below for the reason), and by regrading this relation as an Equivalence relation, he was able to show confluence of his calculus.

Unfortunately, we cannot apply his method to our calculus. While continuations captured by `call/cc` cannot be composed (i.e., when a captured continuation is applied, the continuation at that time is discarded), continuations captured by `shift` can be composed. This fact implies that we need to compose the binary relation $\overset{\sim}{\to}_n$ to itself, but since it is

[4] Here, we modified Moreau's notion to our style.

[5] The definitions are written in our style.

not transitive, the composed relation is not $\stackrel{\sim}{\to}_n$, and we get stuck. This means that we cannot prove confluence of our calculus using Moreau's method.

Instead, we devised a new "proof-method". By analyzing the above problem, we noticed that we should change the definition of the transitions so that a flet-term may be reduced in any contexts, in particular, under binders. Surprisingly, we can apply Takahashi's parallel reduction method to our calculus and prove confluence under the new definition of the transition for flet-terms. As a consequence, we can avoid the complicated reasoning about the number of transition steps, which made our proof much clearer.

In summary, we have changed the definition of the calculus, rather than creating a new proof method, which works very well in our setting.

5.5 Conclusion of this chapter

In this chapter, we have proposed a new calculus which has both parallelization operator `future` and delimited-control operator `shift/reset`. And we have shown the semantical transparency of the `future` by defining two abstract machines and proving the equivalence of the two machines.

Chapter 6

An Implementation of λ_{sr}^{ft}

We will explain our study about an implementation of the parallel calculus under the AM-PFSR [55].

6.1 Goal and Overview

This chapter explains an implementation of the language λ_{sr}^{ft} under the parallel semantics given by AM-PFSR. It is a proof-of-concept implementation in the sense that we do not aim for processing practical, large-scale applications using our implementation. Instead, we want to show, by concrete examples, the feasibility and usability of our themes: we maintain the transparency of the semantics, and at the same time, we get a sufficient level of parallelism.

We designed a compiler as a translator from the calculus λ_{sr}^{ft} to an existing language which has support for true parallelism. Using an existing language as the target of the compilation, we can obtain the effect of parallelization mechanism with minimal cost.

Our compiler (translator) is CPS-translation based, and we chose Erlang as the target language. In the next section, we describe the translation in detail.

6.2 Overview of The Compiler

Since the primitive `shift`, `reset` and `future` do not exist in Erlang, we have to translate them away through the compilation.

The reasons that we chose Erlang as the target language are as follows:

Erlang is a functional language

Since Erlang is a functional programming language, it can treat functions as values: functions can be arguments of functions, which are so called higher-order functions. By CPS translations, the continuations are represented as functions and are passed as explicit arguments in each function call. Therefore, we need higher-order functions in the target language.

Erlang can execute programs in truly parallel

Erlang supports parallel executions of programs. This means that, if we have hardware supports for true parallelism, then Erlang programs can be executed in parallel. Furthermore, Erlang's parallelization process is similar to our `future` parallelization: the computation passed to the `spawn` primitive in Erlang runs in parallel with the computation which generated the new computation.

Others

Erlang is adopts Call-by-Value strategy

The evaluation strategy of Erlang is Call-by-Value. Our language under the parallel semantics given by AM-PFSR is also Call-by-Value so it is not too difficult to map computations to target computations.

Erlang is relatively purely functional

Since Erlang is purely functional with the exception of process computations, we only have to care about parallelization mechanisms in generate (compiled) programs in terms of side effects. We believe that this fact makes our implementation be safer than the one with effectful target languages.

The idea of translation is to use a CPS translation. Since the semantics of `shift/reset` are not different to the original ones and the translation of `shift/reset` is proposed by Danvy and Filinski, we use their translation as it is.

The translation of `future` is a challenge of this implementation. In our semantics, a term $E[F[(\texttt{future}\ M)]]$ works as follows (in the following sequences, we ignore stores):

(1) a communication variable p is generated,

(2) a new process which executes $F[p]$ is generated and its computation starts,

(3) the original process starts the computation M,

(4) when the term M becomes a value V, the resulting term N of the computation of $F[p]$ is collected, and the original process begins the computation of $E[N\{p := V\}]$.

By reflecting the behavior of `future` described above, the translation of `future` is given as follows:

$$\begin{aligned}[\![(\texttt{future}\ M)]\!] \stackrel{\text{def}}{=}\ & \lambda v\kappa\gamma.\texttt{let}\ p\ =\ \texttt{genComVar}\ ()\ \texttt{in} \\ & \quad \texttt{let}\ ()\ =\ \texttt{genProcess}(\lambda().\kappa\ p)\ \texttt{in} \\ & \quad [\![M]\!]\ v\ (\lambda v'\gamma'.(\texttt{getResult}\ p)\ v\ \gamma')\ \gamma\end{aligned}$$

To focus on the essence, we have shown a simplified definition above, and will mention a real, more complicated translation later in this section. The three primitives functions `genComVar`, `genProcess` and `getResult` are as follows:

`genComVar`

This primitive function generates a unique communication variable, which is a kind of communication channel used to pass the result of a computation $\kappa\ p$.

`genProcess`

This primitive function generates a new process to execute a computation. The generating process and the generated process can execute each computation in parallel.

`getResult`

This primitive function gets the result of the computation $\kappa\ p$ through the communication variable p.

The actual implementation is more complicated than the above one, because we have to care about the following issues:

- When `shift` captures the continuation containing other processes (such cases occur when a term like $F[(\texttt{future}\ G(\mathcal{S}k.M))]$ is evaluated), the substitution of the captured continuation to the variables *increase* the number of contained processes. In our implementation, we devised a mechanism to manage this kind of terms: we do not make copies of processes. Instead, we execute the continuation only, namely, the surrounding evaluation context of processes which does not contain processes themselves but contains pointers of the processes.
- In the defined semantics, we can operate mutable cells so each CPS-translated functions must pass the store like its continuations. Furthermore, we have to implement a mechanism to prevent the simultaneous access by several processes to the same mutable variable. In our real implementation, each process has own store and can access mutable variables only if the variables are contained in the store. The computation of the process is blocked when a process tries to access a mutable variable which is not contained in its store.

6.3 Implementation Issues

The source language is based on λ_{sr}^{ft} defined in the previous chapter: the syntax is like Scheme, and the semantics is based on AM-PFSR. The language is a functional programming language (i.e., functions are higher-order) and it has `shift/reset` and `future`. To enhance the usability of the calculus, we add other basic primitives such as arithmetic operations, conditional expressions and recursive functions.

The compiler (translator) is written in OCaml. It parses a source program and makes its abstract syntax tree (AST), then it translate the AST into the target AST. The target AST is output as an Erlang program. We note that the translations of almost all expressions other than `future` are based on general 2-CPS translation.

6.4 Running Examples

In this section, we will show some running examples.

6.4.1 Fibonacci Functions

First, we will show the impact of our `future` by using the Fibonacci functions. We will show two programs the first of which is the sequential program (`future` is not used) and the second is its parallelized version.

```
1  (define (fib n)
2    (if (= n 1) 1
3        (if (= n 2) 1
4            (+ (fib (- n 1)) (fib (- n 2))))))
5
6  (define N 30)
7
8  (let* ((x1 (fib N))
9         (x2 (fib N))
10        (x3 (fib N))
11        (x4 (fib N)))
12   (+ (+ x1 x2) (+ x3 x4)))
```

These programs simply compute Fibonacci numbers four times. In the following programs, the four Fibonacci computations are parallelized by `future`.

```
1  (define (fib n)
2    (if (= n 1) 1
3        (if (= n 2) 1
4            (+ (fib (- n 1)) (fib (- n 2))))))
5
6  (define N 30)
7
8  (let* ((x1 (future (fib N)))
9         (x2 (future (fib N)))
10        (x3 (future (fib N)))
11        (x4 (future (fib N))))
12   (+ (+ x1 x2) (+ x3 x4)))
```

The result of execution times of these programs are shown in Table 6.1.

number of cores	1	2	3	4
Sequential	14.58	14.48	14.71	14.41
Parallel	14.95	10.09	8.74	8.33

Table 6.1: Execution Times of Fibonacci

The first line of Table 6.1 shows the number of used CPS cores, and the second and third lines show the results of the sequential program and parallel one respectively.

As we can see the table, the parallelized programs run faster than the sequential one, which means that our `future` has a certain speed up by parallelization.

6.4.2 N-Queen

In this section, we will show N-queen programs to show that we can use `shift/reset` and `future` simultaneously. First we will show the complete program, which is not parallelized.

```
;; Backtrack mechanism
(define fail (lambda (v) (shift (k) 0)))

(define succeed
  (lambda (board)
    (begin (print board)
           (fail 0))))

(define pchoose
  (lambda (lst)
    (shift (k)
      (letrec
          ((loop
            (lambda (lst)
              (if (is_null lst)
                  (fail 0)
                  (begin (k (car lst))
                         (loop (cdr lst)))))))
        (loop lst)))))

;; Main part of N-Queen program
(define max 10)

(define make_lst
  (lambda (n)
    (letrec ((loop (lambda (i)
                     (if (> i n) '()
                         (cons i (loop (+ i 1)))))))
      (loop 1))))

(define next_board
  (lambda (board)
    (let ((next (pchoose (make_lst max))))
      (append board (list next)))))

(define is_vert
  (lambda (board)
    (letrec ((loop (lambda (xs)
                     (if (is_null (cdr xs)) #f
                         (if (memq (car xs) (cdr xs)) #t
                             (loop (cdr xs)))))))
      (loop board))))

(define make_strict_board
  (lambda (board)
    (letrec ((loop (lambda (xs)
                     (if (is_null xs) '()
                         (cons (cons (length xs) (car xs))
                               (loop (cdr xs)))))))
      (reverse (loop (reverse board))))))

(define is_skew_pair
  (lambda (p1 p2)
    (= (abs (- (car p1) (car p2))) (abs (- (cdr p1) (cdr p2))))))

(define is_skew_strict
  (lambda (sbd)
    (letrec ((loop (lambda (hd tl)
                     (if (is_null tl) #f
                         (if (is_skew_pair hd (car tl)) #t
                             (loop hd (cdr tl)))))))
      (if (is_null sbd) #f
          (loop (car sbd) (cdr sbd))))))

(define is_skew
  (lambda (board)
    (is_skew_strict (reverse (make_strict_board board)))))

(define is_correct
  (lambda (board)
```

```
71|    (not (or (is_vert board) (is_skew board)))))
72|
73| (define is_finished
74|   (lambda (board)
75|     (= (length board) max)))
76|
77| (define (search board)
78|     (if (is_finished board)
79|         (succeed board)
80|         (let ((board2 (next_board board)))
81|           (if (is_correct board2)
82|               (search board2)
83|               (fail 0)))))
84|
85| (reset (search '()))
```

The above program shows the complete N-Queen program which can be compiled by our compiler. Here, the number of Queen `N` is defined as 10.

This programs uses `shift/reset` to implement a backtrack mechanism, which are defined as `fail`, `succeed` and `choose` functions in the program.

The behavior of the function `choose` is essentially the same as the `flip` function in Chapter 2. It gets a list as its argument and captures the continuation, then it recursively applies the continuation to the element of the argument list.

Next we will show the parallel version of the program.

```
1| (define pchoose
2|   (lambda (lst lc)
3|     (shift (k)
4|       (letrec
5|           ((loop
6|             (lambda (lst)
7|               (if (is_null lst)
8|                   (fail 0)
9|                   (begin (future (k (car lst)))
10|                          (loop (cdr lst)))))))
11|        (loop lst)))))
```

The above programs shows only the difference of the two programs. The difference is in the `choose` function (for readability, we renamed the `choose` function as `pchoose`). The third line from the the bottom of the program is a parallelized point: the application of the continuation to the list element is run in parallel with the other applications of the other elements of the list.

While this is the one of the simplest parallelization examples of this N-Queen program, this parallelization cannot work well. This way of parallelization generates a lot of processes, which causes hard load to the hardware. Furthermore, since each process has too small computation, we cannot obtain the effect of the parallelization.

To address this problem, we write the parallelized `choose` function as follows.

```
1| (define depth 1)
2|
3| (define pchoose
4|   (lambda (lst)
5|     (shift (k)
6|       (let ((lc (length lst)))
7|         (letrec
8|             ((loop
9|               (lambda (lst)
10|                (if (is_null lst)
11|                    (fail 0)
12|                    (begin (if (<= lc depth)
13|                               (future (k (car lst)))
14|                               (k (car lst)))
15|                           (loop (cdr lst)))))))
16|          (loop lst))))))
```

This program shows the revised version of parallel `choose` function. In this implementation, `future` is invoked when the program tries to define the first line of the board, i.e., the program generated only N processes.

The results are shown in Table 6.2.

As in the table, the execution times are decreased in the third line.

number of cores	1	2	3	4
Sequential	19.42	18.85	18.79	18.65
Parallel	23.03	13.77	11.00	9.79

Table 6.2: Execution Times of N-Queen

6.5 Conclusion

We have implemented the parallel semantics defined by AM-PFSR, which lets us execute programs with true parallelism. Since this implementation is a proof-of-concept one, the impact of parallelizations is not so drastic. The parallelization by `future`, however, has shown a certain speed up in our experiments.

We have succeeded to parallelize a certain kinds of program, which were impossible to be executed in parallel if they were written with `call/cc`. We, therefore, believe that our thesis that `shift/reset` is useful in parallel languages has been substantiated.

Chapter 7

Conclusion

Control abstractions are an important abstraction for building large and complex programs. In this thesis, we study delimited-control operators which we think are a better tool theoretically and practically than ordinary (undelimited) control operators such as `call/cc`. We have studied the semantic foundation of lambda calculi with delimited-control operators `shift` and `reset`, in particular, direct-style equational axiomatization of the CPS semantics and determinism in the presence of parallel primitives. We believe that these are important steps to obtain powerful control operators in future main-stream programming languages.

In Chapter 3, we have successfully shown a sound and complete axiomatization in the equational form for the CbN calculus with `shift/reset`. Our axioms make it possible for us to verify programs with `shift/reset` directly in direct style. In Chapter 4, we have extended the calculus with layered-`shift/reset` and connected it with the CbV variant in term of the thunk translation. This connection has a fruitful consequence which allows us to prove several important properties, and thus we did not have to re-prove the same properties.

In Chapter 5, we have designed a calculus with `shift/reset` and a parallel primitives `future`, and studied its properties. This calculus is practically interesting because one can express various controlful programs and execute them in the parallel computing environment very easily. It is also theoretically interesting since the result of a program in this calculus is the same as that of a program after removing all the occurrences of `future`, namely, the calculus has a deterministic semantics. An implication of this result is that control abstraction and parallel computation can coexist in our calculus.

Finally, we have implemented the parallel semantics in terms of a program translation into Erlang programs, and done several experiments for concrete programs. The results of these experiments are encouraging; we have obtained a certain speed-up in a truly parallel computing environment.

The future challenges of our researches are as follows:

Application to more controlful programs

We selected `shift/reset` to abstract controls since it was proved that they can express all monadic effects. However, as we described in Chapter 4, there are programs which have more comprehensive control structures, for example, a program which has both a backtrack and an exception. In those cases, we must use more expressive control operators such as multi-prompt `shift/reset`.

From parallel computation to concurrent one

For some applications, one may need concurrent execution rather than parallel ones. Then, `shift/reset` may not be useful because the extent of continuations captured by `shift` is always decided by `reset`, which may affect other processes running in parallel. Then, we should consider other control operators for concurrent calculus such as `spawn` [24].

Acknowledgements

I would like to express my gratitude to Prof. Yukiyoshi Kameyama and Dr. Hiroshi Unno. I am also indebted to Prof. Jiro Tanaka, Prof. Daisuke Takahashi, Dr. Atusi Maeda and Dr. Yasuhiko Minamide who have pointed out a variety of comment to this work, and the members of the programming logic group with whom I have studied and discussed for a long time. Finally, but not least, I would like to express my heartfelt thanks to my family for their continual support and encouragement.

Bibliography

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] K. Asai. Offline Partial Evaluation for Shift and Reset. In *PEPM*, pages 3–14, 2004.

[3] K. Asai. On Typing Delimited Continuations: Three New Solutions to the Printf Problem. *Higher-Order and Symbolic Computation*, 22(3):275–291, 2009.

[4] K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. In *APLAS, LNCS 4807*, pages 239–254, 2007.

[5] J. Berdine, P. O'Hearn, U. Reddy, and H. Thielecke. Linear Continuation-Passing. *Higher-Order and Symbolic Computation*, 15(2-3):191–208, 2002.

[6] M. Biernacka and D. Biernacki. Context-based Proofs of Termination for Typed Delimited-Control Operators. In *PPDP*, pages 289–300, 2009.

[7] M. Biernacka, D. Biernacki, and O. Danvy. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Logical Methods in Computer Science*, 1(2), 2005.

[8] O. Danvy and A. Filinski. A Functional Abstraction of Typed Contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.

[9] O. Danvy and A. Filinski. Abstracting Control. In *LFP*, pages 151–160, 1990.

[10] O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[11] R. K. Dybvig, S. P. Jones, and A. Sabry. A Monadic Framework for Delimited Continuations. *J. Funct. Program.*, 17(6):687–730, 2007.

[12] M. Felleisen. The Theory and Practice of First-Class Prompts. In *POPL*, pages 180–190, 1988.

[13] M. Felleisen, D.P. Friedman, E.E. Kohlbecker, and B.F. Duba. Reasoning with Continuations. In *LICS*, pages 131–141, 1986.

[14] A. Filinski. Linear Continuations. In *POPL*, pages 27–38, 1992.

[15] A. Filinski. Representing Monads. In *POPL*, pages 446–457, 1994.

[16] A. Filinski. Representing Layered Monads. In *POPL*, pages 175–188, 1999.

[17] C. Flanagan and M. Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220. ACM, 1995.

[18] C. Flanagan, A. Sabry, Bruce F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *PLDI*, pages 237–247, 1993.

[19] M. Gasbichler and M. Sperber. Final Shift for Call/cc:: Direct Implementation of Shift and Reset. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 271–282. ACM, 2002.

[20] C. A. Gunter, D. Remy, and J. G. Riecke. A Generalization of Exceptions and Control in ML-Like Languages. In *FPCA*, pages 12–23, 1995.

[21] R.H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[22] J. Hatcliff and O. Danvy. Thunks and the Lambda-Calculus. *J. Funct. Program.*, 7(3):303–319, 1997.

[23] H. Herbelin and S. Ghilezan. An Approach to Call-by-Name Delimited Continuations. In *POPL*, pages 383–394, 2008.

[24] R. Hieb and R.K. Dybvig. *Continuations and Concurrency*, volume 25. ACM, 1990.

[25] Y. Kameyama. Axioms for Control Operators in the CPS Hierarchy. *Higher-Order and Symbolic Computation*, 20(4):339–369, 2007.

[26] Y. Kameyama and M. Hasegawa. A Sound and Complete Axiomatization of Delimited Continuations. In *ICFP*, pages 177–188, 2003.

[27] Y. Kameyama and A. Tanaka. Equational Axiomatization of Call-by-Name Delimited Control. In *PPDP*, pages 77–86, 2010.

[28] M. Katz and D. Weise. Continuing Into the Future: On the Interaction of Futures and First-Class Continuations. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 176–184. ACM, 1990.

[29] R. Kelsey, W. Clinger, and J. (eds.) Rees. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[30] O. Kiselyov. Call-by-name Linguistic Side Effects. In *ESSLLI*, 2008.

[31] O. Kiselyov. Delimited Control in OCaml, Abstractly and Concretely. *Theoretical Computer Science*, 435:56–76, 2012.

[32] O. Kiselyov, C. c. Shan, and A. Sabry. Delimited Dynamic Binding. *ICFP*, pages 26–37, 2006.

[33] T. Komiya and T. Yuasa. Extended Continuations for Future-Based Parallel Scheme Languages (in Japanese). *Journal of Information Processing*, 35(11):2382–2391, 1994.

[34] P.J. Landin. A Generalization of Jumps and Labels. In *Report, UNIVAC Systems Programming Research*. Citeseer, 1965.

[35] J. L. Lawall and O. Danvy. Continuation-Based Partial Evaluation. In *LFP*, pages 227–238, 1994.

[36] E. Moggi. Computational Lambda-Calculus and Monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[37] L. Moreau. A Parallel Functional Language with First-Class Continuations. Programming Style and Semantics. *Computers and Artificial Intelligence*, 14(2):173, 1995.

[38] L. Moreau. The Semantics of Scheme with Future. In *Proceedings of the first ACM SIGPLAN International Conference on Functional Programming*, pages 146–156. ACM, 1996.

[39] C. Murthy. Control Operators, Hierarchies, and Pseudo-Classical Type Systems: A-Translation at Work. In *Proc. ACM Workshop on Continuations*, pages 49–71, 1992.

[40] J. Niehren, D. Sabel, et al. Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures. *Electronic Notes in Theoretical Computer Science*, 173:313–337, 2007.

[41] J. Niehren, J. Schwinghammer, and G. Smolka. A Concurrent Lambda-Calculus with Futures. *Theoretical Computer Science*, 364(3):338–356, 2006.

[42] M. Parigot. Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *LPAR*, pages 190–201, 1992.

[43] G. D. Plotkin. Call-by-Name, Call-by-Value and the Lambda-Calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[44] J.C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.

[45] A. Sabry and M. Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[46] A. Saurin. A Hierarchy for Delimited Control in Call-by-Name. In *FOSSACS*, pages 374–388, 2010.

[47] A. Saurin. Standardization and Böhm Trees for $\Lambda\mu$-Calculus. In *FLOPS*, pages 134–149, 2010.

[48] G.L. Steele and G.J. Sussman. Scheme: An interpreter for the extended lambda calculus. *Artificial Intelligence Lab Memo*, 349, Dec. 1975.

[49] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford Univ. Comput. Lab., Oxford, England, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000.

[50] T. Streicher and B. Reus. Classical Logic, Continuations Semantics and Abstract Machines. *J. Funct. Program.*, 8(6):543–572, 1998.

[51] E. Sumii. An Implementation of Transparent Migration on Standard Scheme. In *Scheme and Functional Programming*, pages 61–63, 2000.

[52] M. Takahashi. Parallel Reductions in λ-Calculus. *J. Symbolic Computation*, 7:113–123, 1989.

[53] A. Tanaka and Y. Kameyama. Scalable Web Application Framework based on Delimited Continuation (in Japanese). *Technical Report of IEICE. Software Science*, 109(456):163–168, 2010.

[54] A. Tanaka and Y. Kameyama. A Call-by-Name CPS Hierarchy. In *Functional and Logic Programming*, pages 260–274. Springer, 2012.

[55] A. Tanaka and Y. Kameyama. Sequential Control in Parallel Computation (in Japanese). In *Proceedings of 29th Annual JSSST Conference*, pages 1–2, Aug. 2012.

[56] A. Tanaka and Y. Kameyama. Transparent Semantics of a Calculus with Delimited Control and Future (in Japanese). *Journal of Information Processing*, 54(8):1996–2011, 2013.

[57] H. Thielecke. From Control Effects to Typed Continuation Passing. In *POPL*, pages 139–149, 2003.

[58] H. Thielecke. Answer Type Polymorphism in Call-by-Name Continuation Passing. In *ESOP*, pages 279–293, 2004.