Back to Direct Style: Typed and Tight

Marius Müller
University of Tübingen, Germany

Philipp Schuster
University of Tübingen, Germany

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Klaus Ostermann
University of Tübingen, Germany

Abstract

Translating programs into continuation-passing style is a well-studied tool to explicitly deal with the control structure of programs. This is useful, for example, for compilation. In a typed setting, there also is a logical interpretation of such a translation as an embedding of classical logic into intuitionistic logic. A naturally arising question is whether there is an inverse translation back to direct style. The answer to this question depends on how the continuation-passing translation is defined and on the domain of the inverse translation. In general, translating programs from continuation-passing style back to direct style requires the use of control operators to account for the use of continuations in non-trivial ways.

We present two languages, one in direct style and one in continuation-passing style. Both languages are typed and equipped with an abstract machine semantics. Moreover, both languages allow for non-trivial control flow. We further present a translation to continuation-passing style and a translation back to direct style. We show that both translations are type-preserving and also preserve semantics in a very precise way, giving an operational correspondence between the two languages. Moreover, we show that the compositions of the translations are well-behaved. In particular, they are syntactic one-sided inverses on the full language and full syntactic inverses when restricted to trivial control flow.

Main Reference Marius Müller, Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2023. Back to Direct Style: Typed and Tight. Proc. ACM Program. Lang. 7, OOPSLA1, Article 104 (April 2023), 28 pages. https://doi.org/10.1145/3586056

Comments This is an extended version of the main reference. Compared to the published paper, this report contains the full appendix: full typing rules for both machines, formal description of the pure fragment of λ_{D}, typing rules and operational semantics for conditionals and detailed proofs not given in the published version and the formalization in Idris 2.

1 Introduction

Continuation-passing style (CPS) is a representation of programs where control flow is explicit and named, which is why it is useful as a compiler intermediate representation [1, 19]. It enables several program optimizations by inlining and reduction. Moreover, it is useful as an implementation technique for control operators, because the current continuation is always immediately available. Continuation-passing style translations translate a program into CPS. They are well-studied and exist in many different variants. In typed languages, they are defined as translations on types and terms. When interpreted logically, they correspond to a double-negation translation. However, CPS also has disadvantages [24, 6], as programs are less readable, stack frames are allocated as closures on the heap [11], and control flow is potentially arbitrary, destroying some of the guarantees of structured programming.

Direct style (DS) does not have these disadvantages. It is the preferred way in which to write programs. Many platforms have support for DS code since they come with a call stack where frames are allocated. In a typed language without control operators, programs directly correspond to proofs in intuitionistic logic [16]. In a typed language with control operators, programs correspond to proofs in classical logic [14]. Direct-style translations translate a program from CPS back to DS. DS translations have been studied as well [7, 8, 29, 3, 4], albeit not as much as CPS translations. Usually, they are presented as the inverse of some CPS translation. In a typed setting, DS translations can be seen as the inverse of a double-negation translation.

Having well-behaved translations back and forth between DS and CPS is theoretically interesting, as it helps to understand the relationship between the two styles. But there also is a more practical aspect. Programmers want to write programs in DS, as they are easier to read. However, some optimizations of programs are easier to accomplish in CPS, in particular, advanced control-flow optimizations often amount to simple beta-reduction [31]. Thus, it can be beneficial to translate programs to CPS for compilation, especially for languages with control operators. On the other hand, programs in DS can be executed more efficiently, as frames are allocated on the stack and not on the heap. So, ideally, we would also like to translate optimized programs back to DS to then run them.

To obtain theoretically satisfying results and also lay the groundwork for practical usability, the DS language, the CPS language, the DS translation, the CPS translation, and their combination should satisfy a number of properties, which we will list next. None of the existing work has all of these properties. Therefore, our contributions are the following.

We present two languages, λ_{D} in direct style with control operators and λ_{C} in continuation-passing style.

- Both have a type system, which is useful for compiler intermediate representations, and which means they admit a logical interpretation.
- Both have an abstract machine with a small-step operational semantics. This allows us to prove an operational correspondence with tight bounds on the number of steps.
- Both have the properties of progress and preservation (Theorems 1 and 2 for λ_{D}, Theorems 3 and 4 for λ_{C}). Well-typed programs do not get stuck.

We present two translations between them, a CPS translation and a DS translation.

- Both are defined on the whole language, in other words they work on all programs. This is important if we want to transform programs before translating back.
- Both are compositional, first-order, and one-pass. These are desirable properties of CPS translations and we argue of DS translations as well.
- Both preserve well-typedness (Theorem 7). This is important if we want to interpret them as a double-negation translation and its inverse respectively.
- Both preserve semantics (Theorems 8 and 12). Indeed, we prove a step-wise correspondence with a global upper bound on the number of steps.
- Neither duplicates code. This is important if we want to actually use them in a compiler, where code size matters.
- Both are implemented in the total programming language Idris 2 [5] to make sure all cases are covered. The DS translation has a lot of corner cases and side conditions.

Moreover, when taken together, these two translations have the following desirable properties.

- The DS translation is a right inverse of the CPS translation, syntactically. When we translate a term to DS and back, we arrive at the same term (Theorem 15).
- On DS programs that do not use control operators, the DS translation also is a left inverse of the CPS translation, syntactically (Theorem 17).

```
def f(y: Int): Int {
  ret y + 1
};
val z = f(x);
ret z + 2
```

(a)

```
let f(y : Int | k : ¬ Int) {
  k(y + 1)
};
cnt k₀(z) {
  done(z + 2)
};
f(x | k₀)
```

(b)

```
cnt f(y : Int) {
  cnt k₀(z : Int) {
    done(z + 2)
  };
  k₀(y + 1)
};
f(x)
```

(c)

```
val y = ret x;
val z = ret y + 1;
ret z + 2
```

(d)

Figure 1 Left to right: (a) Original program in DS, (b) CPS program obtained by translation, (c) Transformed (contified) CPS program, (d) Program translated back to DS.

- We extend both translations to machine states, which allows us to freely translate back and forth between arbitrary intermediate states.

None of the existing pairs of CPS and DS translations have all of these properties at the same time. A thorough comparison can be found in Section 4. Next, in Section 2, we introduce both languages and translations by example. In Section 3 we formally define both languages, translations, and theorems about them. Moreover, we show how the languages and translations can be extended with conditionals. Finally, in Section 5 we conclude and lay out future work.

2 Motivation

In this section, we introduce our main ideas by example. We have two languages, one in direct style (λ_D), one in continuation-passing style (λ_C), and two translations between them. We start by programming in DS, translate to CPS, perform some transformations, and translate back to DS. For better discriminability we use colors for DS programs and boldface for keywords in CPS programs.

2.1 Basic Example

To illustrate our languages and translations, we start with the basic example in Figure 1. Subfigure 1a shows a program in our DS language λ_D. We define a function `f` which increments and then returns its parameter. We then call the function `f` on a free variable `x`. Finally, we return the result `z` incremented by two.

Translating to CPS From this program, our CPS translation produces the λ_C program in Subfigure 1b. The function f now receives a continuation parameter k. Instead of returning, it jumps to this continuation with the result. We then construct a continuation k_0. It represents the rest of the program after the call to f. It increments z by two and calls the free top-level continuation `done`. Finally, the call to f now receives this newly constructed continuation. Running our DS translation on this program in CPS produces exactly the original program in Subfigure 1a: it converts programs back to direct style.

Transforming We now transform the program to an equivalent one. An example for such a transformation is *contification*, which specializes a function to one of its call sites. It is often applied in optimizing compilers [19] that use CPS as intermediate language. The program in

Subfigure 1c is the result of applying contification to function f and its call site. The function f has become a continuation, hence the name contification. Instead of taking a continuation as a parameter, it now always jumps to k_0.

Translating back to DS Finally, we translate the manipulated CPS program back to direct style. Subfigure 1d shows the result of our DS translation. In this example, the resulting program is a sequence of bindings. All continuations have been eliminated. Running our CPS translation on this program produces exactly the program in CPS shown in Subfigure 1c.

Both translations are defined for the whole language without any side conditions. This allows us to freely transform programs before translating back. In this example, we did not use any control operators, in other words, programs were *pure*. On the pure fragment, both of our translations are inverses of each other, syntactically (Theorems 15 and 17). Finally, both translations not only preserve semantics, but on the pure fragment they also execute exactly the same number of steps before reaching the final state (Theorem 9 and Corollary 20). In the next subsection, we will leave the pure fragment and demonstrate the use of control operators.

2.2 Control Operators

In the previous subsection, we demonstrated our translations on pure programs. However, our DS language λ_{D} also includes control operators for two reasons: Firstly, we want to enable programmers to use them to structure their programs. Secondly, translating CPS programs back to DS requires the insertion of control operators sometimes. Our goal is to insert them only where necessary, *i.e.* where continuations are used non-trivially.

The example programs in Figure 2 use the following type of results that may either be a success, or a failure.

```
data Result { Success(String), Failure(String) }
```

Furthermore, they use a primitive `#readFile` operation for asynchronous file access. It receives a file path and two callbacks: one for success and one for failure.

Programming with control operators In Subfigure 2a we use control operators to define a function `readFile` that returns a `Result` but internally uses the asynchronous operation `#readFile`. In the implementation of `readFile`, we use the control operator `suspend` to *capture and remove* the current call stack and make it available under the name `onReturn`. We then define two *new processes* `onSuccess` and `onFailure` and invoke `#readFile` with these. Both processes resume the original computation, one with `Success` and the other with `Failure`. It is important to emphasize: the body of `suspend` does *not* run in the context of a call stack. This means that we can neither return a value nor call a function. All we can do is call primitive operations or resume processes. The same is true for the bodies of processes. The function `tryReadingFile` can now be written straightforwardly in direct style. The function `readFile` acts as a shield that protects the rest of the program from being infected by the asynchronous nature of the operation `#readFile`.

Translating control operators to CPS We again translate this program to CPS. The result is the program in Subfigure 2b. This is roughly what we would have to write manually in a language without control operators. The whole program, including the function tryReadingFile,

```
def readFile(path: Path): Result {
  suspend { onReturn ⇒
    process onSuccess(content: String) {
      resume onReturn(Success(content))
    };
    process onFailure(error: String) {
      resume onReturn(Failure(error))
    };
    #readFile(path, onSuccess, onFailure)
  }
};

def tryReadingFile(path: Path): String {
  val result = readFile(path);
  match result {
    case Success(_) ⇒ ret "success"
    case Failure(_) ⇒ ret "failure"
  }
}
```

(a) Original λ_D program.

```
let readFile(path : Path | onReturn : ¬ Result) {
  cnt onSuccess(content) {
    onReturn(Success(content))
  };
  cnt onFailure(error) {
    onReturn(Failure(error))
  };
  #readFile(path, onSuccess, onFailure)
};

let tryReadingFile(path : Path | cont : ¬ String) {
  cnt k0(result) {
    match result {
      case Success(_) ⇒ cont("success")
      case Failure(_) ⇒ cont("failure")
    }
  };
  readFile(path | k0)
}
```

(b) CPS translation into λ_C.

```
def tryReadingFile(path: Path): String {
  suspend { cont ⇒
    process onSuccess(_) {
      resume cont("success") };
    process onFailure(_) {
      resume cont("failure") };
    #readFile(path, onSuccess, onFailure)
  }
}
```

(c) DS translation of transformed version into λ_D.

```
let tryReadingFile(path : Path | cont : ¬ String) {
  cnt onSuccess(_) {
    cont("success") };
  cnt onFailure(_) {
    cont("failure") };
  #readFile(path, onSuccess, onFailure)
}
```

(d) Transformed λ_C program.

Figure 2 Example program using control operators to program with asynchronous IO.

must be written in CPS as well. A CPS translation automates this process and can serve as an implementation technique for control operators.

Transforming control operators We again transform the CPS program. This time we use some standard inlining and reduction. The transformed version of the previous program is shown in Subfigure 2d. Concretely, we have inlined readFile, inlined k_0, and reduced the exposed matches on known constructors. While this transformation is trivial in CPS, it would be difficult to achieve the same result directly in DS due to the use of control operators.

Translating control operators back to DS Unfortunately, the resulting program is still in CPS and does not use the call stack anymore. On platforms that have call stacks, we want to avoid programs in CPS since they might allocate continuations on the heap, rather than using the stack. Luckily, we can translate this program back to direct style. Our translation results in approximately[1] the program in Subfigure 2c. In this example, our translation

[1] The actual result uses a more complicated combination of control operators in order to make the theorem of operational correspondence true.

had to insert a use of the control operator `suspend`. There is no way around this, as the continuation `cont` is used twice. The resulting function `tryReadingFile` has the same type as the original one and it can be used in direct style wherever the original function was used. Indeed, this is always the case for our translations (Theorem 7). Moreover, of course, our translations guarantee that the resulting programs have the same semantics (Corollaries 11 and 14).

2.3 Classical Logic

Since our direct-style language λ_D is a typed language with control operators, programs in this language can be seen as proofs in classical logic; indeed, the typing rules of our control operators `suspend` and `run` can be seen as the axioms of double negation elimination and cut respectively [14]. Similar pairs of control operators appear in [23, 2, 20].

$$\frac{\Gamma,\ k\ :\ \neg\,\tau \vdash\ s\ :\ \#}{\Gamma \vdash\ \textbf{suspend}\ \{\ k \Rightarrow s\ \}\ :\ \tau}\ [\textsc{Suspend}] \qquad \frac{\Gamma \vdash\ k\ :\ \neg\,\tau \qquad \Gamma \vdash\ s\ :\ \tau}{\Gamma \vdash\ \textbf{run}(k)\ \{\ s\ \}\ :\ \#}\ [\textsc{Run}]$$

In a judgment $\Gamma \vdash\ s\ :\ \#$ the statement s is *undelimited*. It does not return anything, because there is nowhere to return to. In other words, it proves a contradiction. The **suspend** statement is of type τ when its body uses the current continuation of type $\neg\,\tau$ to prove a contradiction. Similarly, the **run** statement proves a contradiction from a proof of $\neg\,\tau$ and a proof of τ. Operationally, the statement s is *delimited* by k. The previously used form **resume** $k(v)$ simply is syntactic sugar for **run**(k) $\{$ **ret** v $\}$, which first installs the delimiter k and then immediately returns the value v to it.

The typing rule for **process** creates a proof of the negation $\neg\,\tau$ by deriving a contradiction from a proof of τ.

$$\frac{\Gamma,\ x\ :\ \tau \vdash\ s_0\ :\ \# \qquad \Gamma,\ k\ :\ \neg\,\tau \vdash\ s\ :\ \#}{\Gamma \vdash\ \textbf{process}\ k(x\ :\ \tau)\ \{\ s_0\ \};\ s\ :\ \#}\ [\textsc{Process}]$$

Given these, it is possible to construct the classical proof of the law of excluded middle using the axiom of double negation elimination (example left).

```
def lawOfExcludedMiddle(): ¬A + A {
  suspend { k ⇒
    process k0(a: A) {
      resume k(Inr(a)) };
    resume k(Inl(k0))
  }
}
```

```
def callcc(program: (A → B) → A): A {
  suspend { k ⇒
    def escape(a: A): B {
      suspend { _ ⇒ resume k(a) } };
    run(k) { program(escape) }
  }
}
```

Operationally, it captures and removes the current stack `k` and uses it twice. First, to construct a new process `k0` of type `¬A`. Then, to delimit a statement that returns this new process injected as the left alternative `Inl(k0)`. It is also possible to recover the classical control operator `callcc`, whose type corresponds to Peirce's law (example on the right, above). Operationally, it captures and removes the current stack `k`, and then defines a function `escape` that, when called, will remove and discard the current stack and resume with `k`, instead. It then calls the given program with `escape`. Let us stress that all programs of type `Int`, even when they use these control operators, or in other words even when they correspond to classical proofs, compute a value of type `Int` (Theorem 5).

Composing our translations between λ_C and λ_D results in the same program when starting at λ_C. In the other direction this is not always the case. Consider the following contrived example on the left, which uses control operators to define the identity function.

```
def identity(a: A): A {
  suspend { k ⇒
    run(k) { ret a }
  }
}
```

```
let identity(a : A ∣ k : ¬ A) {
  k(a)
}
```

```
def identity(a: A) {
  ret a
}
```

Our CPS translation yields the term in λ_C in the middle. When we convert this program back to direct style we obtain the result on the right. Indeed, since the continuation is used trivially in this example, the use of control operators is not needed. In such cases, the composition of our translations will remove these unnecessary appeals to classical reasoning. However, a round trip like this is not always a simplification, even if it does not result in the same program. This is because different combinations of control operators in a DS term that are not simplifications of one another can yield the same CPS term. Nevertheless, a round trip will not insert an unreasonable amount of control operators as can be seen from the results on operational correspondence.

2.4 Section Conclusion

We present a language λ_D in direct style with control operators and a language λ_C in continuation-passing style. Moreover, we present two translations that allow us to go back and forth between the two. In our typed setting, they are theoretically interesting. Moreover, they also are practically useful as they allow us to translate a program to CPS, transform it, and go back to DS.

3 Technical Development

In this section, we formally introduce our direct-style language λ_D and our continuation-passing style language λ_C. Note that the language constructs **resume** (which has already been desugared in the previous section) and `done` are not included and instead defined in terms of other language constructs. For each language, we define a type system and operational semantics in terms of abstract machines. Moreover, we describe the translations between the two languages. Both translations are defined on the full language and satisfy several meta-theoretical properties, both individually and when composed with each other.

The paper is accompanied by an intrinsically-typed formalization of both languages and their operational semantics in Idris 2 [5], which implies type safety (Theorems 1, 2, 3, 4). This formalization also includes the two translations between the languages and the intrinsically-typed nature implies well-typedness preservation (Theorem 7). For the other properties we have pen-and-paper proofs which are given in Appendix D.

3.1 Direct Style

Figure 3 defines the syntax of our DS language λ_D. We syntactically distinguish between statements, which can have control effects, and expressions that can not, *i.e.* we use fine-grain call-by-value [22]. The only expressions are variables and literals. In particular, this means that there are no anonymous functions, and each function must be given a name.

Syntax of statements For a cleaner presentation, sequencing of statements and returning expressions is explicit. Moreover, functions are always unary, so application is always on

Syntax of λ_{D}

Statements	s	$::=$	**val** $x = s;\ s$	sequence
		$\mid$	**ret** e	return
		$\mid$	**def** $f(x : \tau)\ \{\ s\ \};\ s$	define
		$\mid$	$f(e)$	call
		$\mid$	**process** $k(x : \tau)\ \{\ s\ \};\ s$	process
		$\mid$	**suspend** $\{\ k \Rightarrow s\ \}$	suspend
		$\mid$	**run**$(e)\ \{\ s\ \}$	run
		$\mid$	**exit** e	exit
Variables	v	$::=$	$x,\ f,\ k$	variables
Expressions	e	$::=$	v	variables
		$\mid$	$0 \mid 1 \mid \ldots$	integers

Syntax of Types

Types	τ	$::=$	$\tau \to \tau$	function type
		$\mid$	$\neg\, \tau$	stack type
		$\mid$	Int	base type
Environment Type	Γ	$::=$	$\Gamma,\ x : \tau$	extended environment
		$\mid$	$\emptyset$	empty environment
Context Type	ξ	$::=$	τ	delimited context
		$\mid$	$\#$	undelimited context

Figure 3 The direct-style language λ_{D}.

a single expression argument. This means that all intermediate results need to be named which greatly helps avoiding administrative redexes in the translations between λ_{D} and λ_{C}. While our DS language shares this property with ANF, it is different from the latter in that we allow nesting of sequencing statements. The control operator **suspend** $\{\ k \Rightarrow s\ \}$ captures the current continuation, *i.e.* the current stack, and makes it available as a process in its body s by binding it to variable k. As we will see shortly it is more akin to the control operator $\mathcal{C}$ proposed in [12] than to `call/cc` since it does not leave the stack in place. This is important since continuations in the CPS language λ_{C} can be discarded and this possibility should be reflected in λ_{D} in order to facilitate translating back to direct style. The counterpart **run**$(e)\ \{\ s\ \}$ runs statement s in the context of a process e. Using the **process** $k(x : \tau)\ \{\ s_1\ \};\ s_2$ statement such a process can be constructed and bound to k in the environment in s_2. Having explicit names for processes is again helpful for the translations since the continuations in the CPS language are also explicitly named, as we will see shortly. A program can be terminated with an expression e using **exit** e.

Syntax of types and context types There is one base type Int and a standard function type. Furthermore, the system includes a special type $\neg\,\tau$ for stacks expecting an argument of type τ. Typing environments are entirely standard.

We have two kinds of contexts, and consequently two kinds of context types. Besides the types τ indicating that a statement runs in the delimiting context of a stack it returns to, there is also the context type $\#$ which signifies that a statement is undelimited and does not return at all.

Statement Typing $\boxed{\Gamma \vdash s : \xi}$

$$\frac{\Gamma \vdash s_0 : \tau_0 \qquad \Gamma,\, x_0 : \tau_0 \vdash s : \tau}{\Gamma \vdash \textbf{val}\ x_0 = s_0;\ s : \tau}\ [\textsc{Sequence}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{ret}\ e : \tau}\ [\textsc{Return}]$$

$$\frac{\Gamma,\, x : \tau \vdash s_0 : \tau_0 \qquad \Gamma,\, f : \tau \to \tau_0 \vdash s : \xi}{\Gamma \vdash \textbf{def}\ f(x : \tau)\ \{\ s_0\ \};\ s : \xi}\ [\textsc{Define}]$$

$$\frac{\Gamma(f) = \tau \to \tau_0 \qquad \Gamma \vdash e : \tau}{\Gamma \vdash f(e) : \tau_0}\ [\textsc{Call}]$$

$$\frac{\Gamma,\, x : \tau \vdash s_0 : \# \qquad \Gamma,\, k : \neg\,\tau \vdash s : \xi}{\Gamma \vdash \textbf{process}\ k(x : \tau)\ \{\ s_0\ \};\ s : \xi}\ [\textsc{Process}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{exit}\ e : \#}\ [\textsc{Exit}]$$

$$\frac{\Gamma,\, k : \neg\,\tau \vdash s : \#}{\Gamma \vdash \textbf{suspend}\ \{\ k \Rightarrow s\ \} : \tau}\ [\textsc{Suspend}] \qquad \frac{\Gamma \vdash e : \neg\,\tau \qquad \Gamma \vdash s : \tau}{\Gamma \vdash \textbf{run}(e)\ \{\ s\ \} : \#}\ [\textsc{Run}]$$

Expression Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash 19 : \mathsf{Int}}\ [\textsc{Int}]$$

Figure 4 Typing rules for the direct-style language λ_D.

3.1.1 Typing

The typing rules for λ_D are given in Figure 4. The judgment and rules for expressions are completely standard. Statements, however, are typed with a context type ξ, *i.e.* either with an ordinary type τ or with $\#$. That is, some of them require a stack to be present, others do not, yet others are polymorphic. The rules for sequencing, returning, function definition, and application are mostly unsurprising. But note that since the only purpose of the **val**-construct is to sequence statements it cannot be typed with context type $\#$. It would, however, be straightforward to add another construct which allows to bind an expression to a variable in a subsequent statement that has an arbitrary context type. Moreover, in rules Define and Process the whole statement has to have the same context type ξ as the rest of the statement s. This means that such statements have context type $\#$ exactly when they occur in a context where no stack is available. Processes never return and as such the body of a process definition has context type $\#$. Exiting a program does of course not return either as can be seen in rule Exit. In rule Suspend, the overall statement has type τ, thus the captured stack has type $\neg\,\tau$. It is available in the body s under the name k. The body does not return, and the captured stack is undelimited, so **suspend** can be understood as shifting to the undelimited world. Dually, as rule Run shows, **run** shifts back to the delimited world. The delimiting process e has type $\neg\,\tau$ and consequently the statement s has to return a value of type τ. In the following, we will usually assume the expression e to be a variable.

Syntax of Machine States for λ_{D}

Values	V	$::=$	$\{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow s\ \}$	closure
		$\mid$	K	stack
		$\mid$	$0\ \mid 1\ \mid \ldots$	integer
Environments	E	$::=$	$\mathsf{E},\ x \mapsto \mathsf{V}$	binding
		$\mid$	$\bullet$	empty
Stacks	K	$::=$	$\{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow s\ \}$	underflow
		$\mid$	$\{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow s\ \}\ ::\ \mathsf{K}$	frame
Configurations	M	$::=$	$\langle\ \mathsf{E} \parallel s \parallel \mathsf{K}\ \rangle$	delimited execution
		$\mid$	$\langle\ \mathsf{E} \parallel s\ \rangle$	undelimited execution

Figure 5 Operational semantics of λ_{D}.

Configuration Typing $\boxed{\vdash\ \mathsf{M}}$

$$\frac{\mathsf{E}\ \vdash_{env}\ \Gamma \qquad \Gamma \vdash\ s\ :\ \tau \qquad \tau\ \vdash_{stk}\ \mathsf{K}}{\vdash\ \langle\ \mathsf{E} \parallel s \parallel \mathsf{K}\ \rangle}\ [\textsc{Delim}]$$

$$\frac{\mathsf{E}\ \vdash_{env}\ \Gamma \qquad \Gamma \vdash\ s\ :\ \#}{\vdash\ \langle\ \mathsf{E} \parallel s\ \rangle}\ [\textsc{Undelim}]$$

Figure 6 Typing rules for machine states for λ_{D}.

3.1.2 Semantics

The operational semantics of λ_{D} is given by an abstract machine.

Machine states Figure 5 defines the syntax of the machine. There are two different modes of machine states, one for delimited execution and one for undelimited execution. Both consist of a statement currently in focus and an environment, but the delimited mode additionally has a stack the statement returns to. The statement in the undelimited mode is non-returning and thus there is no stack. Environments are mappings from variables to values where values are either closures, integers or stacks. A stack is a list of frames whose last frame is an underflow frame. Note that while the syntax of closures and underflow frames is the same, the body of a closure is a returning statement whereas the body of an underflow frame is undelimited. The final states of the machine are of the form $\langle\ \mathsf{E} \parallel \textbf{exit}\ e\ \rangle$ with e being the final result. We further define `done` to be $\{\ \bullet,\ (x\ :\ \tau) \Rightarrow \textbf{exit}\ x\ \}$ and use this as a bottom frame to start evaluation of a closed program s in state $\langle\ \bullet \parallel s \parallel \texttt{done}\ \rangle$. Typing of the machine is unsurprising (Figure 6 defines the rules for configurations, the full set of rules is given in Appendix A). For a machine state to be well-typed, the free variables of the statement in focus must be covered by the environment. Moreover, in the case of delimited configurations the type of the statement must agree with the type the stack expects.

Machine steps The evaluation steps of the abstract machine are defined in Figure 7. The first three rules *(push)*, *(ret-1)*, and *(ret-0)* are mostly standard. The rules for returning to the stack come in two flavors, depending on whether there is more than one frame left on the

$$\begin{array}{lll}
\textit{(push)} & \langle\, \mathsf{E} \parallel \textbf{val}\ x_0 \ = \ s_0;\ s \parallel \mathsf{K} \,\rangle & \rightarrow \langle\, \mathsf{E} \parallel s_0 \parallel \{\ \mathsf{E},\ (x_0 \ :\ \tau_0) \Rightarrow s\ \} \ :: \ \mathsf{K} \,\rangle \\
\textit{(ret-1)} & \langle\, \mathsf{E},\ v \mapsto \mathsf{V} \parallel \textbf{ret}\ v \parallel \{\ \mathsf{E}_0,\ (x \ :\ \tau) \Rightarrow s\ \} \ :: \ \mathsf{K} \,\rangle & \rightarrow \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \parallel \mathsf{K} \,\rangle \\
\textit{(ret-0)} & \langle\, \mathsf{E},\ v \mapsto \mathsf{V} \parallel \textbf{ret}\ v \parallel \{\ \mathsf{E}_0,\ (x \ :\ \tau) \Rightarrow s\ \} \,\rangle & \rightarrow \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \,\rangle \\
\textit{(def-1)} & \langle\, \mathsf{E} \parallel \textbf{def}\ f(x \ :\ \tau)\ \{\ s_0\ \};\ s \parallel \mathsf{K} \,\rangle & \rightarrow \langle\, \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \ :\ \tau) \Rightarrow s_0\ \} \parallel s \parallel \mathsf{K} \,\rangle \\
\textit{(def-0)} & \langle\, \mathsf{E} \parallel \textbf{def}\ f(x \ :\ \tau)\ \{\ s_0\ \};\ s \,\rangle & \rightarrow \langle\, \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \ :\ \tau) \Rightarrow s_0\ \} \parallel s \,\rangle \\
\textit{(call)} & \langle\, \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \ :\ \tau) \Rightarrow s\ \},\ v \mapsto \mathsf{V} \parallel f(v) \parallel \mathsf{K} \,\rangle & \rightarrow \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \parallel \mathsf{K} \,\rangle \\
\textit{(proc-1)} & \langle\, \mathsf{E} \parallel \textbf{process}\ k(x \ :\ \tau)\ \{\ s_0\ \};\ s \parallel \mathsf{K} \,\rangle & \rightarrow \langle\, \mathsf{E},\ k \mapsto \{\ \mathsf{E},\ (x \ :\ \tau) \Rightarrow s_0\ \} \parallel s \parallel \mathsf{K} \,\rangle \\
\textit{(proc-0)} & \langle\, \mathsf{E} \parallel \textbf{process}\ k(x \ :\ \tau)\ \{\ s_0\ \};\ s \,\rangle & \rightarrow \langle\, \mathsf{E},\ k \mapsto \{\ \mathsf{E},\ (x \ :\ \tau) \Rightarrow s_0\ \} \parallel s \,\rangle \\
\textit{(sus)} & \langle\, \mathsf{E} \parallel \textbf{suspend}\ \{\ k \Rightarrow s\ \} \parallel \mathsf{K} \,\rangle & \rightarrow \langle\, \mathsf{E},\ k \mapsto \mathsf{K} \parallel s \,\rangle \\
\textit{(run)} & \langle\, \mathsf{E},\ v \mapsto \mathsf{K} \parallel \textbf{run}(v)\ \{\ s\ \} \,\rangle & \rightarrow \langle\, \mathsf{E},\ v \mapsto \mathsf{K} \parallel s \parallel \mathsf{K} \,\rangle
\end{array}$$

Figure 7 Machine steps of λ_{D}.

stack or not. If the frame was the last one, then the machine continues in undelimited mode, else execution goes on with the remaining stack. We have omitted the rules for returning integers, as they are the same, except that they do not need a binding in the environment. The rules for defining a function add a closure to the environment and focus on the remaining statement s. Statement s returns if and only if there is a stack. When calling a function f, it is looked up in the environment and the environment that was captured in f is reinstalled. The body of f comes into focus and an additional binding for the parameter is added to the environment. We have again omitted the case where the argument is an integer. The rules for defining a process are essentially the same as the ones for defining functions, but note that such a process always consists of one undelimited underflow frame instead of a closure. Rule *(sus)* captures the current stack, binds it as a process to k in the environment and executes the undelimited statement in the body. In particular, if k is not used in the statement, then the stack is discarded. Rule *(run)* looks up the process bound to variable v in the environment and runs the body s with this process installed as a stack. Note that the binding for the process in the environment persists as s may still contain v free.

3.2 Continuation-Passing Style

Figure 8 defines the syntax of our CPS language λ_{C}. The syntax for expressions and types is identical to those of λ_{D}. Note, however, that there is no notion of context type in λ_{C} as no term ever returns. Instead, they will jump to the next continuation.

Syntax of terms Functions in this calculus have exactly one ordinary parameter, but they additionally have a separate continuation parameter, which is syntactically separated with a bar. This is also reflected in function application. The **cnt** $k(x \ :\ \tau)\ \{\ t\ \};\ t$ term can be used to construct a continuation. Such continuations take exactly one parameter and are called with exactly one expression. As in λ_{D} there is a construct **exit** e which terminates the program with an expression e.

Syntax of λ_{C}

Terms	t	$::=$	$\textbf{let}\ f(x : \tau \mid k : \neg\,\tau)\ \{\ t\ \};\ t$	function
		$\mid$	$f(e \mid e)$	application
		$\mid$	$\textbf{cnt}\ k(x : \tau)\ \{\ t\ \};\ t$	continuation
		$\mid$	$k(e)$	jump
		$\mid$	$\textbf{exit}\ e$	exit
Variables	v	$::=$	$x,\ f,\ k$	variables
Expressions	e	$::=$	v	variables
		$\mid$	$0 \mid 1 \mid \ldots$	integers

Syntax of Types

Types	τ	$::=$	$\tau \to \tau$	function type
		$\mid$	$\neg\,\tau$	continuation type
		$\mid$	Int	base type
Environment Type	Γ	$::=$	$\Gamma,\ x : \tau$	extended environment
		$\mid$	$\emptyset$	empty environment

Figure 8 The continuation-passing style language λ_{C}.

Term Typing $\boxed{\Gamma \vdash t}$

$$\frac{\Gamma,\ x : \tau,\ k : \neg\,\tau_0 \vdash t_0 \qquad \Gamma,\ f : \tau \to \tau_0 \vdash t}{\Gamma \vdash \textbf{let}\ f(x : \tau \mid k : \neg\,\tau_0)\ \{\ t_0\ \};\ t}\ [\textsc{Let}]$$

$$\frac{\Gamma(f) = \tau \to \tau_0 \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash c : \neg\,\tau_0}{\Gamma \vdash f(e \mid c)}\ [\textsc{App}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{exit}\ e}\ [\textsc{Ext}]$$

$$\frac{\Gamma,\ x : \tau \vdash t_0 \qquad \Gamma,\ k : \neg\,\tau \vdash t}{\Gamma \vdash \textbf{cnt}\ k(x : \tau)\ \{\ t_0\ \};\ t}\ [\textsc{Cnt}] \qquad \frac{\Gamma(k) = \neg\,\tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash k(e)}\ [\textsc{Jmp}]$$

Expression Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash 19 : \mathsf{Int}}\ [\textsc{Int}]$$

Figure 9 Typing rules for λ_{C}.

3.2.1 Typing

Figure 9 defines the typing rules for λ_{C}. The judgments and rules for expressions are again the same as in λ_{D}. Since terms do not return anything, they are only typed in an environment, but not against a type. The typing rules are entirely unsurprising. In rule Let the return type of the function being defined is the type expected by its continuation parameter, as usual in CPS. Rule App shows that the continuation expression c of a function application must have continuation type. In the following, we will thus usually assume c to be a variable.

Syntax of Machine States for λ_C

Values	V	$::=$	$\{\ \mathsf{E},\ (x\ :\ \tau \mid k\ :\ \neg\ \tau) \Rightarrow t\ \}$	function closure
		$\mid$	$\{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow t\ \}$	continuation closure
		$\mid$	$0 \mid 1 \mid \ldots$	integer
Environments	E	$::=$	$\mathsf{E},\ x \mapsto \mathsf{V}$	binding
		$\mid$	$\bullet$	empty
Configurations	M	$::=$	$\langle\ \mathsf{E} \parallel t\ \rangle$	execution

Figure 10 Operational semantics of λ_C.

(let) $\langle\ \mathsf{E} \parallel \textbf{let}\ f(x\ :\ \tau \mid k\ :\ \neg\ \tau_0)\ \{\ t_0\ \};\ t\ \rangle \rightarrow$
$\langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x\ :\ \tau \mid k\ :\ \neg\ \tau_0) \Rightarrow t_0\ \} \parallel t\ \rangle$

(app) $\langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x\ :\ \tau \mid k\ :\ \neg\ \tau_0) \Rightarrow t\ \},\ v \mapsto \mathsf{V},\ c \mapsto W \parallel f(v \mid c)\ \rangle \rightarrow$
$\langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V},\ k \mapsto W \parallel t\ \rangle$

(cnt) $\langle\ \mathsf{E} \parallel \textbf{cnt}\ k(x\ :\ \tau)\ \{\ t_0\ \};\ t\ \rangle \rightarrow$
$\langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow t_0\ \} \parallel t\ \rangle$

(jmp) $\langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x\ :\ \tau) \Rightarrow t\ \},\ v \mapsto \mathsf{V} \parallel k(v)\ \rangle \rightarrow$
$\langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel t\ \rangle$

Figure 11 Machine steps of λ_C

3.2.2 Semantics

The operational semantics of λ_C is again given by an abstract machine.

Machine states The syntax of the machine is shown in Figure 10. In contrast to the DS machine, there is only one machine mode, with an environment and a term in focus. Environments look the same as in direct style, but the values are different (except for integers). Closures now come in two flavors, function closures with a continuation parameter and continuation closures without one. The latter correspond to stacks in the direct-style language. This is also visible in the typing of values (the rules are given in Appendix A). While closures with a continuation parameter are of function type, those without are of continuation type. Similar to the DS-machine the final states are $\langle\ \mathsf{E} \parallel \textbf{exit}\ e\ \rangle$. We again define `done` to be $\{\ \bullet,\ (x\ :\ \tau) \Rightarrow \textbf{exit}\ x\ \}$ and define a closed program t to have one free variable k and start evaluation in state $\langle\ k \mapsto \texttt{done} \parallel t\ \rangle$.

Machine steps The evaluation steps of the abstract machine are shown in Figure 11. The rules for calling a function or continuation with an integer argument are again omitted. The rules for defining a function add a closure with a continuation parameter to the environment and focus on the remaining term. Calling a function f essentially proceeds in the same way as in λ_D but instead of having a stack, there must be a binding for the continuation argument in the environment. The rules for defining and calling a continuation are similar, but a continuation closure is added to the environment and the call needs no continuation argument.

3.3 Translations

Next, we describe the translations back and forth between the two languages presented above. In either case, we state how the translations act on typing derivations. Note that no typing information is used in an essential way and the same translations work in an untyped setting. In fact, typing information is only needed for type annotations of parameters and we often leave these annotations out in the definition of the translations.

3.3.1 From Direct Style to Continuation-Passing Style

We first describe the CPS translation on statements and then show how to extend it to the abstract machine. Note that expressions are translated trivially.

CPS translation of statements Figure 12 defines the CPS translation on statements. To avoid administrative redexes, the translation has an additional input, which stands for the current continuation. It is either a continuation variable or no continuation (indicated with $\bullet$). In the first case, the continuation variable is added to the typing context for the translated statement, while in the second case the typing context stays exactly the same.

As usual, return statements are translated to calls to the current continuation. Sequencing is translated by defining a new continuation with the translation of the second statement as the body and then translating the first statement with this new continuation. A translated function application, as usual, gets the current continuation as its continuation argument.

For function definitions there are two cases, depending on whether the remaining statement is returning or not. In either case, the translated function definition abstracts over a fresh continuation parameter which is then used to translate the body. The remaining statement is translated with the original input continuation or no continuation. The translation of defining a new process also has these two cases, *i.e.* the translation of the remaining statement is the same as for function definition. The definition of a new process is translated to a definition of a continuation with the same name and its body is translated with no continuation as it is non-returning.

In the translation of **suspend** the body is translated with no continuation but the continuation variable bound by **suspend** is replaced with the input continuation of the translation as the latter is the current continuation. Running a statement s in a process bound to k_0 is translated by translating s with continuation input k_0. The **exit**-statement is translated to its counterpart in λ_{C}.

CPS translation of machine The extension of the CPS translation to the abstract machine is shown in Figure 13. As can be seen in the translation on the typing judgments, stacks that expect values of type τ themselves become values of type $\neg\,\tau$.

The two different modes of machine configurations are translated a bit differently. For undelimited execution, where there is no stack, the statement is translated with no input continuation in the translated environment. For delimited execution the stack is translated to a value that is added to the translated environment with a fresh variable and this variable is then used as the input continuation for the translation of the statement.

Environments are translated by pointwise translation of the bound values. For values there are three cases: integers are translated trivially, the translation of stacks is defined separately, and closures are translated in essentially the same way as function definitions but put together with the translated environment. For stacks, there are two cases. Underflow frames are essentially translated in the same way as definitions of a new process but again put

$$\boxed{(\Gamma_1,\ \Gamma_2 \vdash\ s\ :\ \tau\ \times k) \quad\longrightarrow\quad \Gamma_1,\ k\ :\ \neg\,\tau,\ \Gamma_2 \vdash\ t}$$

$$\boxed{(\Gamma \vdash\ s\ :\ \#\ \times \bullet) \quad\longrightarrow\quad \Gamma \vdash\ t}$$

Translation of Statements

$$\begin{array}{lll}
\mathcal{C}[\![\ \textbf{ret}\ e\ ]\!]_k & = k(e) & \\
\mathcal{C}[\![\ \textbf{val}\ x\ =\ s_0;\ s\ ]\!]_k & = \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s\ ]\!]_k\ \};\ \mathcal{C}[\![\ s_0\ ]\!]_{k_0} & \text{where } k_0 \text{ fresh} \\
\mathcal{C}[\![\ f(e)\ ]\!]_k & = f(e \mid k) & \\
\mathcal{C}[\![\ \textbf{def}\ f(x)\ \{\ s_0\ \};\ s\ ]\!]_k & = \textbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \};\ \mathcal{C}[\![\ s\ ]\!]_k & \text{where } k_0 \text{ fresh} \\
\mathcal{C}[\![\ \textbf{def}\ f(x)\ \{\ s_0\ \};\ s\ ]\!]_\bullet & = \textbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \};\ \mathcal{C}[\![\ s\ ]\!]_\bullet & \text{where } k_0 \text{ fresh} \\
\mathcal{C}[\![\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ ]\!]_k & = \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \};\ \mathcal{C}[\![\ s\ ]\!]_k & \\
\mathcal{C}[\![\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ ]\!]_\bullet & = \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \};\ \mathcal{C}[\![\ s\ ]\!]_\bullet & \\
\mathcal{C}[\![\ \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \}\ ]\!]_k & = \mathcal{C}[\![\ s\ ]\!]_\bullet \quad \text{with } k_0\ :=\ k & \\
\mathcal{C}[\![\ \textbf{run}(k_0)\ \{\ s\ \}\ ]\!]_\bullet & = \mathcal{C}[\![\ s\ ]\!]_{k_0} & \\
\mathcal{C}[\![\ \textbf{exit}\ e\ ]\!]_\bullet & = \textbf{exit}\ e &
\end{array}$$

Figure 12 Translation to continuation-passing style.

Translation of Values $\boxed{\vdash_{val}\ \mathsf{V}\ :\ \tau\ \longrightarrow\ \vdash_{val}\ \mathsf{V}\ :\ \tau}$

$$\begin{array}{lll}
\mathcal{C}_V[\![\ \{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow s\ \}\ ]\!] & = \{\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ (x\ :\ \tau \mid k\ :\ \neg\,\tau_0) \Rightarrow \mathcal{C}[\![\ s\ ]\!]_k\ \} & \text{where } k \text{ is fresh} \\
\mathcal{C}_V[\![\ \mathsf{K}\ ]\!] & = \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] & \\
\mathcal{C}_V[\![\ 19\ ]\!] & = 19 &
\end{array}$$

Translation of Environments $\boxed{\mathsf{E}\ \vdash_{env}\ \Gamma\ \longrightarrow \mathsf{E}\ \vdash_{env}\ \Gamma}$

$$\begin{array}{ll}
\mathcal{C}_E[\![\ \bullet\ ]\!] & = \bullet \\
\mathcal{C}_E[\![\ \mathsf{E},\ x \mapsto \mathsf{V}\ ]\!] & = \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ x \mapsto \mathcal{C}_V[\![\ \mathsf{V}\ ]\!]
\end{array}$$

Translation of Stacks $\boxed{\tau\ \vdash_{stk}\ \mathsf{K}\ \longrightarrow\ \vdash_{val}\ \mathsf{V}\ :\ \neg\,\tau}$

$$\begin{array}{lll}
\mathcal{C}_K[\![\ \{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow s\ \}\ ]\!] & = \{\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ (x\ :\ \tau) \Rightarrow \mathcal{C}[\![\ s\ ]\!]_\bullet\ \} & \\
 & & \text{where } k \text{ is fresh} \\
\mathcal{C}_K[\![\ \{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow s\ \}\ ::\ \mathsf{K}\ ]\!] & = \{\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!],\ (x\ :\ \tau) \Rightarrow \mathcal{C}[\![\ s\ ]\!]_k\ \} &
\end{array}$$

Translation of Configurations $\boxed{\vdash\ \mathsf{M}\ \longrightarrow \vdash\ \mathsf{M}}$

$$\begin{array}{lll}
\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel s\ \rangle\ ]\!] & = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_\bullet\ \rangle & \\
 & & \text{where } k \text{ is fresh} \\
\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel s \parallel \mathsf{K}\ \rangle\ ]\!] & = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_k\ \rangle &
\end{array}$$

Figure 13 Translation of machine to continuation-passing style.

together with the translated environment. The translation of a frame on top of a remaining stack is similar to the case of delimited machine configurations in that it adds the translation of the remaining stack to the translated environment with a fresh variable and uses this variable as input continuation for the translation of the body.

3.3.2 From Continuation-Passing Style Back to Direct Style

Let us now consider the opposite direction. We first describe how the DS translation behaves on terms and then extend it to the abstract machine. Again, expressions are translated trivially. As the DS translation is bottom-up we often have some side conditions for different

$$\boxed{\Gamma,\ k : \neg\,\tau \vdash\ t\ \longrightarrow (\Gamma \vdash\ s\ :\ \tau \rightsquigarrow k)} \qquad \boxed{\Gamma \vdash\ t\ \longrightarrow (\Gamma \vdash\ s\ :\ \# \rightsquigarrow \bullet)}$$

Translation of Terms

$$\frac{}{\mathcal{D}\llbracket\ f(e \mid k)\ \rrbracket\ =\ \mathsf{resetIfFree}(k)(f(e))}\,[\textsc{DApp}] \qquad \frac{}{\mathcal{D}\llbracket\ k(e)\ \rrbracket\ =\ \mathbf{ret}\ e \rightsquigarrow k}\,[\textsc{DJmp}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ =\ s_0 \rightsquigarrow k_0}{\mathcal{D}\llbracket\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \}\ \rrbracket\ =\ \mathbf{def}\ f(x)\ \{\ s_0\ \}}\,[\textsc{DLet1}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ =\ s_0 \rightsquigarrow \bullet}{\mathcal{D}\llbracket\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \}\ \rrbracket\ =\ \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \}}\,[\textsc{DLet2}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ =\ s_0 \rightsquigarrow v_0 \qquad v_0\ \neq\ k_0}{\mathcal{D}\llbracket\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \}\ \rrbracket\ =\ \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow \mathbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \}}\,[\textsc{DLet3}]$$

$$\frac{\mathcal{D}\llbracket\ t\ \rrbracket\ =\ s \rightsquigarrow \bullet}{\mathcal{D}\llbracket\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rrbracket\ =\ \mathcal{D}\llbracket\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \}\ \rrbracket;\ s \rightsquigarrow \bullet}\,[\textsc{DLetN}]$$

$$\frac{\mathcal{D}\llbracket\ t\ \rrbracket\ =\ s \rightsquigarrow k}{\mathcal{D}\llbracket\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rrbracket\ =\ \mathsf{resetIfFree}(k)(\mathcal{D}\llbracket\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \}\ \rrbracket;\ s)}\,[\textsc{DLetR}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ =\ s_0 \rightsquigarrow \bullet}{\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \}\ \rrbracket\ =\ \mathbf{process}\ k_0(x)\ \{\ s_0\ \}}\,[\textsc{DCnt1}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ =\ s_0 \rightsquigarrow x}{\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \}\ \rrbracket\ =\ \mathbf{process}\ k_0(x)\ \{\ \mathbf{run}(x)\ \{\ s_0\ \}\ \}}\,[\textsc{DCnt2}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ =\ s_0 \rightsquigarrow v_0 \qquad v_0\ \neq\ x \qquad \mathcal{D}\llbracket\ t\ \rrbracket\ =\ s \rightsquigarrow \bullet}{\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rrbracket\ =\ \mathsf{resetIfFree}(v_0)(\mathbf{val}\ x\ =\ \mathbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0)}\,[\textsc{DCntNV}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ \mathbf{else} \qquad \mathcal{D}\llbracket\ t\ \rrbracket\ =\ s \rightsquigarrow \bullet}{\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rrbracket\ =\ \mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \}\rrbracket;\ s \rightsquigarrow \bullet}\,[\textsc{DCntN}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ =\ s_0 \rightsquigarrow v_0 \qquad v_0\ \neq\ x \qquad \mathcal{D}\llbracket\ t\ \rrbracket\ =\ s \rightsquigarrow k \qquad k\ \neq\ k_0}{\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rrbracket\ =\ \mathsf{resetIfFree}(v_0)(\mathbf{val}\ x\ =\ \mathbf{suspend}\ \{\ k_0 \Rightarrow \mathbf{run}(k)\ \{\ s\ \}\ \};\ s_0)}\,[\textsc{DCntROV}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ \mathbf{else} \qquad \mathcal{D}\llbracket\ t\ \rrbracket\ =\ s \rightsquigarrow k \qquad k\ \neq\ k_0}{\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rrbracket\ =\ \mathsf{resetIfFree}(k)(\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \}\rrbracket;\ s)}\,[\textsc{DCntRO}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ =\ s_0 \rightsquigarrow v_0 \qquad v_0\ \neq\ x \qquad \mathcal{D}\llbracket\ t\ \rrbracket\ =\ s \rightsquigarrow k_0}{\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rrbracket\ =\ \mathsf{resetIfFree}(v_0)(\mathbf{val}\ x\ =\ s;\ s_0)}\,[\textsc{DCntRTV}]$$

$$\frac{\mathcal{D}\llbracket\ t_0\ \rrbracket\ \mathbf{else} \qquad \mathcal{D}\llbracket\ t\ \rrbracket\ =\ s \rightsquigarrow k_0}{\mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rrbracket\ =\ \mathcal{D}\llbracket\ \mathbf{cnt}\ k_0(x)\ \{\ t_0\ \}\rrbracket;\ \mathbf{run}(k_0)\ \{\ s\ \} \rightsquigarrow \bullet}\,[\textsc{DCntRT}]$$

$$\frac{}{\mathcal{D}\llbracket\ \mathbf{exit}\ e\ \rrbracket\ =\ \mathbf{exit}\ e \rightsquigarrow \bullet}\,[\textsc{DExit}]$$

Figure 14 Translation back to direct style.

cases which is why we give the translation in the style of inference rules.

DS translation of terms Figure 14 defines the translation on terms. Where the CPS translation has an additional input, the DS translation has an additional output. If the translated term is delimited, there is an additional variable as output standing for the stack the translated term returns to. This variable is then removed from the environment. If the translated term is non-returning, then there is no variable as output, again indicated by $\bullet$, and the environment stays the same. Let us refer to this additional output as continuation output. The translation is bottom-up in that it first recursively translates the subterms and then decides what to do depending on the continuation outputs of the translated subterms. In some cases we have to use the constructs **suspend** and **run** in λ_{D}, which allow us to flexibly adapt the stack where necessary, and with the bottom-up approach we try to recognize where to insert these control operators in order to minimize their use. There are three base cases without subterms. The **exit**-term is translated to its counterpart in λ_{D} and does not return. For the translation of a function application $f(e \mid k)$ we use the function resetIfFree to first check whether the continuation parameter k appears free in $f(e)$ (which by typing means $e = k$).

$$\mathsf{resetIfFree}(k)(s) \;=\; \textbf{if } k \in \mathsf{FV}(s) \textbf{ then } \textbf{run}(k)\ \{\ s\ \} \rightsquigarrow \bullet \textbf{ else } s \rightsquigarrow k$$

If not, the translation has k as its output continuation as this represents the current continuation, *i.e.* the stack the translated term returns to. Otherwise, we need a control operator, since in a pure statement the current continuation cannot be the argument of a function. Specifically, we reset $f(e)$ to run in the process bound to k, *i.e.* **run**(k) $\{\ f(e)\ \}$. Note that this statement then has no output continuation as it does not return. In the case of calling a continuation $k(e)$, typing makes it impossible that k is free in e (*i.e.* $e = k$) since $\neg\,\tau \neq \tau$. Thus, the translation simply returns e with continuation output k. Again, this is the stack that the translated term returns to. Without typing we could again reset with k upon free occurrence.

For function definitions there are quite a few cases to consider, depending on the continuation outputs of translating the subterms. In any case, the result is again a function definition in λ_{D}, but we have to insert control operators in some cases.

Rules DLet1, DLet2, and DLet3 define a helper function that determines what happens with the actual function definition as this is independent of the remaining term. In abuse of notation, we use the same name as for the translation function. The pure case is in DLet1 where the translated body of the function definition returns to the continuation parameter k_0 which is the current continuation. Note that this means in particular that k_0 is not free in the translated body and we can just drop the continuation parameter. If the translated body does not return, as in DLet2, the current continuation represented by the function parameter must have been used in a non-pure way, so we have to insert a **suspend** to capture the current stack. If the stack v_0 the translated body returns to is different from the current continuation (DLet3), we also have to capture the latter but we additionally have to insert a **run**(v_0) to reset the translated body with the proper stack.

Rules DLetN and DLetR then determine how to translate the whole term for function definitions based on the result of translating the remaining term. If the translation of the remaining term does not return, then neither does the whole statement. If it does return to stack k, we again have to check whether k occurs free in the translated term and reset with k if it does.

For the definition of a continuation there are even more cases to consider. Depending on the continuation output of the translation of the body, s_0, we either obtain a sequencing statement or a definition of a new process.

The latter is the case if s_0 does not return or if it returns to the parameter x of the continuation. We have again defined a helper function for these two cases, DCnt1 and DCnt2, which only acts on the actual definition of the continuation. In the case where the translated body returns to x we insert a **run**(x). In either of these two cases we then distinguish three cases for the translation of the remaining term t to a statement s. The non-returning case DCntN and the case DCntRO where s returns to a stack k different from the process k_0 just defined are similar to the corresponding cases of function definition, *i.e.* in the second case we reset with k if it occurs free. In the third case DCntRT where s returns to k_0 we insert a **run**(k_0), as the process just defined cannot be the current continuation.

The case left to consider is when the translated body s_0 returns to a stack v_0 different from the parameter x. In this case we can translate to a sequencing statement that binds x and has s_0 as its second statement. Depending on the result s of translating the remaining term we have to insert control operators. The pure case is when s returns to k_0 as in DCntRTV. If s does not return as in DCntNV, we have to insert a **suspend** to capture the current stack and bind it to k_0. Finally, in DCntROV where s returns to a stack k different from k_0 we capture the current stack to k_0 and then reset with k. In any case, we have to check whether v_0 appears free, in which case we have to reset the whole statement with v_0.

DS translation of machine Figure 15 shows how the DS translation is extended to the abstract machine. Note how, inverse to the CPS translation, values of type $\neg\,\tau$ become stacks that expect a value of type τ. Machine configurations are translated based on the translation s of the term in focus. If s is non-returning, it is executed in an undelimited configuration with the translated environment. If s returns to k, the value bound to k is removed from the translated environment – we write $\mathsf{E}.\mathsf{rm}(k)$ for removing the binding to k from environment E – and its translation is used as stack for the delimited machine configuration.

Environments are again translated by pointwise translation of the bound values. For values, we distinguish returning from non-returning ones. Those that return are integers, which are translated trivially, and function closures, which are again translated in essentially the same way as function definitions put together with the translated environment. The non-returning values are translated to stacks. The translation of these continuation closures is similar to the translation of continuation definitions, in particular in the cases where the translated body does not return or returns to the parameter of the continuation closure. The case when the translated body returns to a stack k different from the parameter is more akin to the translation of a delimited machine configuration in that the binding for k is removed from the translated environment and the translation of the value bound to k is used as the remaining stack on top of which the frame is put.

Translation of Values $\boxed{\vdash_{val}\ \mathsf{V} : \tau \longrightarrow \vdash_{val}\ \mathsf{V} : \tau}$

$$\frac{\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow k_0}{\mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x : \tau \mid k_0 : \neg\,\tau_0) \Rightarrow t_0\ \}\ ]\!] = \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x : \tau) \Rightarrow s_0\ \}}\ [\text{DFClo1}]$$

$$\frac{\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet}{\mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x : \tau \mid k_0 : \neg\,\tau_0) \Rightarrow t_0\ \}\ ]\!] = \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x : \tau) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \}}\ [\text{DFClo2}]$$

$$\frac{\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow v_0 \qquad v_0 \neq k_0}{\mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x : \tau \mid k_0 : \neg\,\tau_0) \Rightarrow t_0\ \}\ ]\!] = \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x : \tau) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \}}\ [\text{DFClo3}]$$

$$\frac{}{\mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ ]\!] = \mathcal{D}_K[\![\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ ]\!]}\ [\text{DCClo}] \qquad \frac{}{\mathcal{D}_V[\![\ 19\ ]\!] = 19}\ [\text{DVInt}]$$

Translation of Environments $\boxed{\mathsf{E} \vdash_{env} \Gamma \longrightarrow \mathsf{E} \vdash_{env} \Gamma}$

$$\frac{}{\mathcal{D}_E[\![\ \bullet\ ]\!] = \bullet}\ [\text{DEnv1}] \qquad \frac{}{\mathcal{D}_E[\![\ \mathsf{E},\ x \mapsto \mathsf{V}\ ]\!] = \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!]}\ [\text{DEnv2}]$$

Translation of Continuation Closures $\boxed{\vdash_{val}\ \mathsf{V} : \neg\,\tau \longrightarrow \tau \vdash_{stk}\ \mathsf{K}}$

$$\frac{\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet}{\mathcal{D}_K[\![\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ ]\!] = \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x : \tau) \Rightarrow s_0\ \}}\ [\text{DCClo1}]$$

$$\frac{\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow x}{\mathcal{D}_K[\![\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ ]\!] = \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x : \tau) \Rightarrow \textbf{run}(x)\ \{\ s_0\ \}\ \}}\ [\text{DCClo2}]$$

$$\frac{\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow k \qquad k \neq x}{\mathcal{D}_K[\![\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ ]\!] = \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ (x : \tau) \Rightarrow s_0\ \} :: \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]}\ [\text{DCClo3}]$$

Translation of Configurations $\boxed{\vdash\ \mathsf{M} \longrightarrow \vdash\ \mathsf{M}}$

$$\frac{\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet}{\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel t\ \rangle\ ]\!] = \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel s\ \rangle}\ [\text{DMch1}]$$

$$\frac{\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k}{\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel t\ \rangle\ ]\!] = \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle}\ [\text{DMch2}]$$

Figure 15 Translation of machine back to direct style.

3.4 Properties

With the languages and translations in place, we want to show that they are well-behaved. To this end, we show several meta-theoretical properties.

Soundness We start with the soundness of the languages. The following progress and preservation theorems have been proven in Idris 2 by the intrinsic typing and the totality of the two `step`-functions on the machines.

▶ **Theorem 1** (Progress (DS)).
For DS machine state M, if $\vdash\ \mathsf{M}$ *then either M is of the form* $\langle\ \mathsf{E} \parallel \textbf{exit}\ e\ \rangle$ *or there exists*

a unique machine state M' such that $M \to M'$.

▶ **Theorem 2** (Preservation (DS)).
For DS machine state M, if $\vdash M$ and $M \to M'$ then $\vdash M'$.

▶ **Theorem 3** (Progress (CPS)).
For CPS machine state M, if $\vdash M$ then either M is of the form $\langle\, E \parallel \textbf{exit}\ e \,\rangle$ or there exists a unique machine state M' such that $M \to M'$.

▶ **Theorem 4** (Preservation (CPS)).
For CPS machine state M, if $\vdash M$ and $M \to M'$ then $\vdash M'$.

Next, we show that if a closed program with type τ does not contain **exit**, then the resulting expression has the same type. Note that if a program does contain **exit**, it may end with an expression having a different type.

▶ **Theorem 5** (Type preservation (DS)).
*If $\vdash\ s\ :\ \tau$ and s does not contain **exit** and $\langle\, \bullet \parallel s \parallel \texttt{done} \,\rangle \to^* \langle\, E \parallel \textbf{exit}\ e \,\rangle$, then $\Gamma \vdash\ e\ :\ \tau$ where $E \vdash_{env} \Gamma$.*

Proof. None of the machine steps can add an **exit**, so the final **exit** comes from returning to the initial `done`. Thus, e has the type which `done` expects, *i.e.*, τ. ◀

▶ **Theorem 6** (Type preservation (CPS)).
*If $k\ :\ \neg\,\tau \vdash\ t$ and t does not contain **exit** and $\langle\, k \mapsto \texttt{done} \parallel t \,\rangle \to^* \langle\, E \parallel \textbf{exit}\ e \,\rangle$, then $\Gamma \vdash\ e\ :\ \tau$ where $E \vdash_{env} \Gamma$.*

Proof. None of the machine steps can add an **exit**, so the final **exit** comes from calling the initial `done`. Thus, e has the type which `done` expects, *i.e.*, τ. ◀

Before going on, we identify the pure fragment of λ_{D}. The pure fragment consists of those statements built up from sequencing, returning, function definition and application. Moreover, there are no continuation types anymore and the only undelimited frame a stack can contain is `done` (the syntax is given in Appendix B). The typing rules and machine steps are restricted accordingly. We write $\mathsf{pure}(s)$ if the statement s is contained in the pure fragment of λ_{D} and similarly for the other syntactic categories. The progress and preservation theorems as above also hold for the pure fragment as is proven by a restriction of our formalization in Idris 2.

Typability preservation Now we prove some theorems about the translations, starting with typability preservation. This has been proven in Idris 2 by the intrinsic typing and the various `transform`-functions.

▶ **Theorem 7** (Typability preservation).
All translations take typing derivations to typing derivations as stated above their definitions.

Semantics preservation Next, we prove semantics preservation. Since we use a small-step abstract machine semantics, we can be very precise about how semantics is preserved, including upper and lower bounds for the number of steps the translated machine takes. In the following, we identify machine configurations and frames which can be identified by weakening, contraction or reordering of environments. Moreover, we consider equality always modulo α-equivalence. We start with the CPS translation. For every step the DS-machine takes, the CPS-machine either takes no step or one step.

▶ **Theorem 8** (Simulation of $\mathcal{C}_M$).
If $\vdash\ M$ *and* $M \to M'$, *then* $\mathcal{C}_M[\![\ M\ ]\!] \to^? \mathcal{C}_M[\![\ M'\ ]\!]$, *where* $\to^?$ *means* 0 *or* 1 *machine step.*

The reason why the correspondence is not 1-to-1 is that there are steps for the control operators **suspend** and **run** in the DS-machine but the CPS translation already includes those steps. For example, in the translation of **suspend** $\{\ k \Rightarrow s\ \}$ we simply translate the body s and replace k by the variable in the environment to which the current continuation is bound upon translating the machine state. In the DS-version, this capturing of the stack into the environment is done in a machine step. But this also means that for the pure fragment, we can be even more precise: for every step the DS-machine takes, the CPS-machine also takes exactly one step.

▶ **Theorem 9** (Simulation of $\mathcal{C}_M|_{pure}$).
If $\vdash\ M$ *and* pure(M) *and* $M \to M'$, *then* $\mathcal{C}_M[\![\ M\ ]\!] \to \mathcal{C}_M[\![\ M'\ ]\!]$.

Proof. Inspection of the proof of Theorem 8. ◀

As a corollary we obtain that the CPS-machine takes at most as many steps as the DS-machine.

▶ **Corollary 10** (Operational Reduction (CPS)).
If $\vdash\ M$ *and* $M \to^n M'$, *then* $\mathcal{C}_M[\![\ M\ ]\!] \to^m \mathcal{C}_M[\![\ M'\ ]\!]$ *where* $m \leq n$.

Proof. By induction on n using Theorem 8. ◀

In particular, a closed term that evaluates to a result in DS, evaluates to the same result in CPS in at most as many steps.

▶ **Corollary 11** (Evaluation (CPS)).
If $\vdash\ s\ :\ \tau$ *and* $\langle\ \bullet \parallel s \parallel \texttt{done}\ \rangle \to^n \langle\ E \parallel \textbf{exit}\ e\ \rangle$
then $\langle\ k \mapsto \texttt{done} \parallel \mathcal{C}[\![\ s\ ]\!]_k\ \rangle \to^m \langle\ \mathcal{C}_E[\![\ E\ ]\!] \parallel \textbf{exit}\ e\ \rangle$ *where* k *is fresh and* $m \leq n$.

Let us now look at the DS translation. The CPS-machine needs fewer steps, but we can give an upper bound for the DS-machine: for every step the CPS-machine takes, the DS-machine takes at least one and at most four steps. The reason is that we insert up to three control operators for one construct during translation, each of which needs its own step in the DS-machine.

▶ **Theorem 12** (Simulation of $\mathcal{D}_M$).
If $\vdash\ M$ *and* $M \to M'$, *then* $\mathcal{D}_M[\![\ M\ ]\!] \to^{[1-4]} \mathcal{D}_M[\![\ M'\ ]\!]$, *where* $\to^{[1-4]}$ *means* 1, 2, 3 *or* 4 *machine steps.*

We obtain similar corollaries as for the CPS translation above.

▶ **Corollary 13** (Operational Reduction (DS)).
If $\vdash\ M$ *and* $M \to^n M'$, *then* $\mathcal{D}_M[\![\ M\ ]\!] \to^m \mathcal{D}_M[\![\ M'\ ]\!]$ *where* $m \leq 4n$.

Proof. By induction on n using Theorem 12. ◀

▶ **Corollary 14** (Evaluation (DS)).
If $\vdash\ t$ *with* $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow k$ *and* $\langle\ k \mapsto \texttt{done} \parallel t\ \rangle \to^n \langle\ E \parallel \textbf{exit}\ e\ \rangle$
then $\langle\ \bullet \parallel s \parallel \texttt{done}\ \rangle \to^m \langle\ \mathcal{D}_E[\![\ E\ ]\!] \parallel \textbf{exit}\ e\ \rangle$ *where* $m \leq 4n$.

Round trips Another interesting question is how the compositions of the two translations behave. The round trip for λ_{C} has the particularly pleasing property of yielding the syntactically same term.

▶ **Theorem 15** ($\mathcal{D}$ is right inverse of $\mathcal{C}$).
If $\Gamma \vdash\ t$ *and* $\mathcal{D}[\![\ t\ ]\!] \ =\ s \rightsquigarrow k$*, then* $\mathcal{C}[\![\ s\ ]\!]_k \ =\ t$.
If $\Gamma \vdash\ t$ *and* $\mathcal{D}[\![\ t\ ]\!] \ =\ s \rightsquigarrow \bullet$*, then* $\mathcal{C}[\![\ s\ ]\!]_\bullet \ =\ t$.

We can extend this property to the abstract machines.

▶ **Theorem 16** (Right inverse for machines).
If $\vdash\ M$*, then* $\mathcal{C}_M[\![\ \mathcal{D}_M[\![\ M\ ]\!]\ ]\!] \ =\ M$.
If $\vdash_{val}\ V\ :\ \tau$*, then* $\mathcal{C}_V[\![\ \mathcal{D}_V[\![\ V\ ]\!]\ ]\!] \ =\ V$.
If $E\ \vdash_{env}\ \Gamma$*, then* $\mathcal{C}_E[\![\ \mathcal{D}_E[\![\ E\ ]\!]\ ]\!] \ =\ E$.
If $\vdash_{val}\ V\ :\ \neg\ \tau$*, then* $\mathcal{C}_K[\![\ \mathcal{D}_K[\![\ V\ ]\!]\ ]\!] \ =\ V$.

This means, in particular, that the CPS translation is surjective and the DS translation is injective. As we have seen in Section 2, the converse is not true, since in general there are multiple DS-statements, *i.e.* with different combinations of control operators, that are mapped to the same CPS-term. Therefore, we do not in general insert the same control operators during the DS translation that are removed when CPS-translating. This also means that there is some choice of which combination of control operators to insert in the DS translation. The choices here are made in such a way that we can easily give the tight bounds for the number of machine steps when evaluating.

Moreover, as the round trip also affects the environment and the stack, we cannot expect that the original machine reduces to the one after the round trip. But semantics preservation at least gives us that both machine configurations eventually evaluate to the same result (if they terminate) with the given upper bound for the number of steps.

For the pure fragment of λ_{D} we can again be much more precise and in fact again obtain the syntactically same term after the round trip.

▶ **Theorem 17** ($\mathcal{D}$ is left inverse of $\mathcal{C}|_{pure}$).
If $\Gamma \vdash\ s\ :\ \tau$ *with* $\mathsf{pure}(s)$*, then* $\mathcal{D}[\![\ \mathcal{C}[\![\ s\ ]\!]_k\ ]\!] \ =\ s \rightsquigarrow k$ *where* k *is fresh.*

This can also be extended to the abstract machine.

▶ **Theorem 18** (Restricted left inverse for machines).
If $\vdash\ M$ *and* $\mathsf{pure}(M)$ *then* $\mathcal{D}_M[\![\ \mathcal{C}_M[\![\ M\ ]\!]\ ]\!] \ =\ M$.
If $\vdash_{val}\ V\ :\ \tau$ *and* $\mathsf{pure}(V)$*, then* $\mathcal{D}_V[\![\ \mathcal{C}_V[\![\ V\ ]\!]\ ]\!] \ =\ V$.
If $E\ \vdash_{env}\ \Gamma$ *and* $\mathsf{pure}(E)$*, then* $\mathcal{D}_E[\![\ \mathcal{C}_E[\![\ E\ ]\!]\ ]\!] \ =\ E$.
If $\tau\ \vdash_{stk}\ K$ *and* $\mathsf{pure}(K)$*, then* $\mathcal{D}_K[\![\ \mathcal{C}_K[\![\ K\ ]\!]\ ]\!] \ =\ K$.

Semantics reflection With the round-trip results in place, we get as corollaries some results about reflection of machine steps. Let us first look at the DS translation. If there is a step in the DS-machine between translated machine configurations, then there must be a step in the CPS-machine between the original states.

▶ **Corollary 19** ($\mathcal{D}_M$ is step-reflecting).
Given $\vdash\ M$ *and* $\vdash\ M'$*, then* $\mathcal{D}_M[\![\ M\ ]\!] \rightarrow \mathcal{D}_M[\![\ M'\ ]\!]$ *implies* $M \rightarrow M'$.

For the pure fragment of λ_{D} we have a similar result. But first, let us prove another corollary, which states that evaluation in the CPS-machine is closed on the image of the pure fragment and in lockstep with the DS-machine.

Syntax of λ_{D}

Statements	s	$::=$	$\dots \mid \textbf{if } e \textbf{ then } s \textbf{ else } s$	conditionals

Syntax of λ_{C}

Terms	t	$::=$	$\dots \mid \textbf{if } e \textbf{ then } t \textbf{ else } t$	conditionals

Syntax of Expressions

Expressions	e	$::=$	$\dots \mid \mathsf{true} \mid \mathsf{false}$	booleans

Syntax of Types

Types	τ	$::=$	$\dots \mid \mathsf{Bool}$	base type

Figure 16 Syntax for the extension of the languages with conditionals.

▶ **Corollary 20** (CPS-evaluation is closed and in lockstep on pure fragment)**.**
Given $\vdash M$ *and* $\mathsf{pure}(M)$*, then* $\mathcal{C}_M[\![\ M\ ]\!] \to M'$ *implies* $M \to \mathcal{D}_M[\![\ M'\ ]\!]$ *and* $\mathsf{pure}(\mathcal{D}_M[\![\ M'\ ]\!])$*.*

▶ **Corollary 21** ($C_M|_{pure}$ is step-reflecting)**.**
Given $\vdash M$ *and* $\vdash M'$ *and* $\mathsf{pure}(M)$ *and* $\mathsf{pure}(M')$*, then* $\mathcal{C}_M[\![\ M\ ]\!] \to \mathcal{C}_M[\![\ M'\ ]\!]$ *implies* $M \to M'$*.*

3.5 Conditional Statements

The languages considered so far are rather minimalistic. To make them more practically usable, we now show how they can be extended with conditional statements. Consider a DS translation of an **if**-statement

$$\mathcal{D}[\![\ \textbf{if } e \textbf{ then } t_1 \textbf{ else } t_2\ ]\!] \;=\; \textbf{if } e \textbf{ then } \dots s_1 \dots \textbf{ else } \dots s_2 \dots \rightsquigarrow ?$$

where $\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow k_1$ and $\mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow k_2$. Both branches have to return to the same stack as the whole statement. Thus, if $k_1 = k_2$, we do not need any control operators and the whole statement returns to this stack. This is always the case for the pure fragment. However, if $k_1 \neq k_2$, we have to insert control operators to adapt the stacks the two branches return to.

3.5.1 Biased Solution

An easy solution is to pick one branch and treat it as the preferred branch, adapting the other one. Let us pick the second branch as default. The translation is

$$\mathcal{D}[\![\ \textbf{if } e \textbf{ then } t_1 \textbf{ else } t_2\ ]\!] \;=\; \textbf{if } e \textbf{ then suspend } \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \} \textbf{ else } s_2 \rightsquigarrow k_2$$

Figure 16 shows the straightforward syntax extensions for the two languages with conditionals. As expressions and types are extended identically they are shown only once. The scrutinee of the conditionals is an expression, since all intermediate results have to be named. The typing and operational semantics are straightforward and given in Appendix C.

The CPS translation for conditionals is unsurprising, as either both branches are returning or both are non-returning. In the first case we translate both branches with the input

Translation of Statements

$$\begin{aligned}\mathcal{C}[\![\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ ]\!]_k &= \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_k\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_k\\ \mathcal{C}[\![\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ ]\!]_\bullet &= \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_\bullet\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_\bullet\end{aligned}$$

Figure 17 Translation of conditionals to continuation-passing style.

Translation of Terms

$$\frac{\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow k \qquad \mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow k}{\mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] = \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow k}\ [\text{DIfRR}]$$

$$\frac{\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow \bullet \qquad \mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow \bullet}{\mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] = \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \bullet}\ [\text{DIfNN}]$$

$$\frac{\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow \bullet \qquad \mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow k}{\mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] = \textbf{if}\ e\ \textbf{then suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2 \rightsquigarrow k}\ [\text{DIfRN}]$$

$$\frac{\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow k \qquad \mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow \bullet}{\mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] = \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else suspend}\ \{\ k \Rightarrow s_2\ \} \rightsquigarrow k}\ [\text{DIfNR}]$$

$$\frac{\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow k_1 \qquad \mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow k_2 \qquad k_1 \neq k_2}{\mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] = \textbf{if}\ e\ \textbf{then suspend}\ \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ \textbf{else}\ s_2 \rightsquigarrow k_2}\ [\text{DIfRRD}]$$

Figure 18 Translation of conditionals back to direct style.

continuation of the whole statement, in the second case we translate both branches without an input continuation. The CPS translation is shown in Figure 17.

As explained above, the interesting part of the extension is the DS translation. There are different cases to consider. If both branches return to the same stack or no stack, then we do not have to insert control operators and the whole translated term returns to the common stack in the former case or is non-returning in the latter case. The more interesting cases are if the branches return to different stacks or if one is returning and the other one is not. We have explained the first case above. In the second case we have several options: a) we can again always adapt the branch which is not the preferred one, or b) we insert a **run** for the returning branch resulting in a non-returning statement for the whole term, or c) we insert a **suspend** for the non-returning branch resulting in a returning statement for the whole term. We choose option c) since we aim to obtain a returning statement, as would be the case for a pure term. In particular, in the case that the term surrounding the **if** is pure and the returning branch is too, the only place where control operators are inserted is in the non-returning branch. The DS translation is displayed in Figure 18.

The extension fits in seamlessly with our languages. All properties in the previous subsection still hold exactly as stated.

3.5.2 Other Alternative Designs

While the above solution works without any changes to other parts of the DS translation, it is a bit unsatisfactory with regard to avoiding the insertion of unnecessary control operators. For example, if the continuation output of the preferred branch is not the current continuation, then the whole **if**-statement has to be adapted with control operators again. But in the case that the current continuation is the one the non-preferred branch returns to, this is unnecessary. We actually could have adapted the non-preferred branch instead, resulting in an overall **if**-statement that already returns to the correct stack. To do so, however, we would have to know which continuation is the current continuation.

Passing the current continuation down One option is to pass this information downwards during the DS translation by having an additional continuation variable as input for the DS translation. This works well for the translation of terms. However, when extending the translation to the machine things become more difficult. For intermediate machine states that are not closed it is not always clear which continuation is the current one at the toplevel of the term in focus and for the bindings in the environment. A potential solution could be to annotate machine states with the current continuation and adapt this annotation appropriately when taking a machine step. This approach thus appears to require some change to the existing languages and more investigation is needed to understand whether it is actually feasible.

Non-deterministic translation There is another option that avoids passing down the current continuation in the DS translation. We could instead try to defer the decision which branch of the conditional to adapt. Later, when the current continuation is known, we choose the right one. To do so we have to remember all possible translations for a conditional. Then the translation for the case when the branches return to different continuations as above returns a list as follows

$$\begin{aligned}
&\mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\\
&= [\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ \textbf{suspend}\ \{\ k_1 \Rightarrow \textbf{run}(k_2)\ \{\ s_2\ \}\ \} \rightsquigarrow k_1,\\
&\quad\ \textbf{if}\ e\ \textbf{then}\ \textbf{suspend}\ \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ \textbf{else}\ s_2 \rightsquigarrow k_2,\\
&\quad\ \textbf{if}\ e\ \textbf{then}\ \textbf{suspend}\ \{\ k? \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ \textbf{else}\ \textbf{suspend}\ \{\ k? \Rightarrow \textbf{run}(k_2)\ \{\ s_2\ \}\ \} \rightsquigarrow k?]
\end{aligned}$$

Here $k?$ has to be replaced by the current continuation when it is known, if that case is chosen. Now, if the translation of a subexpression yields a non-singleton list, we either pick the correct option and return that as a singleton list if the current continuation is known, or we do the further translation for all elements in the list and return the resulting list.

At the toplevel we then pick the right option in accordance with the current continuation. However, similar to the solution which passes the current continuation downwards, for non-closed machine states it is not always clear which is the current continuation at the toplevel and hence which option to pick if the translation yields multiple possible results.

4 Related Work

There is a huge body of work on translations back to direct style and in this section we discuss how they relate to what we have presented here. Figure 19 shows several properties and whether they are shown to hold (✓) by the different translations or not (✗), or are conjectured (❍).

4.1 Back to Direct Style I and II

A direct-style translation was first considered by Danvy [7]. They only work on a pure direct-style language but extend their translation to a language with `call/cc` in a follow-up paper [8]. Both papers use higher-order translations to reduce administrative redexes at translation-time. To deal with this in proofs they factor out the administrative reductions for both translations and obtain staged versions, which is more thoroughly investigated

	Typed	One-Pass First-Order Compositional	Small-Step Preserv.	Reduction Theory	Right Inverse DS	Detects Pure Terms
*Calculi **without** control operators*						
Danvy [7]	✗	✗	✗	✗	✓	—
Sabry and Felleisen [28]	✓	✗	✗	✓	✓	—
Flanagan et al. [13]	✗	✗	✓	✗	✓	—
Sabry and Wadler [30]	❍	✓	❍	✓	✓	—
Danvy and Pfenning [10]	✗	✗	✗	✗	✗	—
Nielsen [27]	✗	✓	✗	✗	✓	—
Hatcliff and Danvy [15]	✓	✗	✗	✗	✗	—
*Calculi with **undelimited** control*						
Danvy and Lawall [8]	✗	✗	✗	✗	✗	✓
Sabry and Felleisen [29]	❍	✗	✗	✗	✗	✗
This work	✓	✓	✓	✗	✓	✓
*Calculi with **delimited** control*						
Kameyama and Hasegawa [18]	✗	✗	✗	✗	✗	✗
Kameyama [17]	✗	✗	✗	✗	✗	✗
Biernacki et al. [3]	✗	✓	✗	✓	✓	✓
Biernacki et al. [4]	✗	✓	✗	✓	✓	✓

Figure 19 Summary of related work.

in [21]. In the pure case, where the CPS language is restricted to the image of the CPS translation via an attribute grammar, they obtain as a result that the DS translation is inverse to the CPS translation. For the language extended with `call/cc` they use a counting analysis for continuation identifiers to decide where to insert a control operator. This analysis checks whether a continuation identifier occurs free in the scope of another continuation identifier. This is sufficient as they do not have an abortion operator. We instead rely on a bottom-up translation to also capture the case when no continuation identifier occurs. For these extended languages, the translations only satisfy the weaker property of forming a Galois connection with respect to a kind of normalization, which is to be expected, since the CPS translation removes superfluous control operators which are not inserted again by the DS translation. The normalization consists of performing a round trip, so on terms after a round trip the translations are inverse. As our DS translation is the right inverse of the CPS translation we have a Galois connection in this sense, too. Note, however, that this is different from a Galois connection with respect to reduction (see below), and neither paper above considers reduction or equational theories at all.

4.2 Development of ANF

Another direct-style translation was studied by Sabry and Felleisen [29]. Their goal is to derive a set of rewrite rules for the computational λ_c-calculus [25]. They achieve this goal and present an equational theory that proves the same equations as the CPS-counterpart. This eventually leads to the system of A-normal forms for pure terms. They also give an extension to a calculus with `call/cc` and an abortion operator, which together are equivalent to the control operator of Felleisen et al. [12]. The translations used in either case are based on evaluation contexts, making them non-compositional. In the pure case, the DS translation works on a CPS language which is closed under $\beta\eta$-reduction and thus a bit larger than just the image of the CPS translation. On the full languages, the translations are not inverse of each other, but terms after a round trip are connected to the original terms with respect to an equational theory. A round trip on the DS language brings terms into ANF. When restricting the DS language to ANF the CPS translation becomes injective, so a restriction of the CPS language to the image of the CPS translation makes the two translations inverses. Moreover, with the correct rewrite rules for ANF both translations preserve reduction. An argument for ANF over CPS as a compiler intermediate language was made by Flanagan et al. [13]. There, the result is extended to abstract machines for the two languages, similar to our result for the pure fragment. A difference, however, is that the pure fragment of our language is not in ANF as we allow for nesting of **val**-statements. For their extension with control operators they do not consider reduction, but only an equational theory. For this reason, their DS translation inserts a `call/cc` for every binding of a continuation and explicit calls to the continuation even when unnecessary, in contrast to Danvy and Lawall's and this work. In the end, they state that their results also hold in a typed setting, but do not elaborate on that point much further.

4.3 Further Developments for Calculi Without Control Operators

In subsequent years there were further developments in different directions for direct-style calculi without control operators. Hatcliff and Danvy [15] consider Moggi's monadic meta language λ_{ml} [26] in a typed setting and give a translation from an appropriate CPS language back to the meta language, forming an equational correspondence. As the source language is λ_{ml} this forms a core for DS translations for different evaluation strategies which can be obtained by giving translations of different calculi into λ_{ml} and back.

Sabry and Wadler [30] improve upon previous results by considering a reduction theory for λ_c instead of merely equational correspondences. They show that their translations back and forth form a reflection with respect to reduction, *i.e.* a Galois-connection with the DS translation being the right inverse. This is a stronger result than the one of Sabry and Felleisen [29], which for the composition of the DS translation with the CPS translation only have a correspondence with respect to an equational theory (but already prove the remaining properties of a Galois-connection). The CPS translation used in the paper is not quite compositional but this could be remedied easily by unfolding the translation a bit further. The authors moreover conjecture that their results hold in a typed setting and also when reduction in arbitrary contexts is replaced by deterministic evaluation. Note that this work does not consider a reduction theory but only evaluation with an abstract machine. We have shown that the translations on the pure fragment are two-sided inverses. Perhaps, since everything is named in our calculi, this is not too surprising. It would be interesting to also consider a reduction theory for our approach in general.

A somewhat different direct-style translation is given by Danvy and Pfenning [10] who aim to mechanically prove claims in [7] about continuation parameters. It relies on a stack of intermediate results maintained during the translation, promising more efficiency and simpler proofs. In particular, Nielsen [27] builds on the same technique to give a very simple proof by structural induction that the first-order, one-pass and compositional CPS translation [9] is the left inverse of the DS translation. This kind of DS translation is partly bottom-up in the sense that it uses the stack produced by subexpressions in some cases.

4.4 Back to Direct Style with Delimited Control

A bit later some papers on direct-style translations for calculi with delimited control operators appeared. Kameyama and Hasegawa [18] consider a calculus with one level of `shift` and `reset`, but as they deal with an equational theory they insert a control operator for every occurrence of a continuation binding, similar to Sabry and Felleisen [29]. They show that their translations form an equational correspondence in an untyped setting. This result was later extended to the CPS-hierarchy [17].

Very recently, Biernacki et al. [3] improved upon this result in a similar way as Sabry and Wadler [30] improved upon earlier work [29]. They consider a reduction theory for an untyped calculus with `shift` and `reset` and show that their translations back and forth form a reflection with respect to reduction. A similar result for `shift0` and `reset0` has been shown [4]. To do so they use a CPS-calculus with a mildly context-sensitive grammar tracking which continuation variable is the current continuation and reduction rules that are specifically designed to keep the calculus closed under reduction. This makes it easy to recognize the combinators that are translated to control operators when going back to direct style. It would be interesting to see whether our approach can be extended to delimited control operators, too.

5 Conclusion

We have presented a translation from continuation-passing style back to direct style in a typed setting with undelimited control operators. It is the right inverse of our CPS translation and on the pure fragment even the two-sided inverse. Both translations are total and preserve the small-step evaluation of the abstract machine semantics of the languages with tight bounds. Moreover, both translations are first-order, one-pass, and compositional, facilitating proofs by structural induction. Our DS translation is bottom-up, in order to recognize where to insert control operators and minimize their use.

We have shown an extension of our basic calculi with conditionals such that all theorems hold as stated. However, in some cases the DS translation inserts more control operators than necessary. We have sketched two more sophisticated approaches that should be investigated further. Another interesting question is whether our translations can be extended to delimited control operators.

It is possible to choose between different equivalent possibilities for the DS translation in some of the cases. We have designed our translation in such a way that it fits particularly nicely with the abstract machine. It would be interesting to investigate whether different choices yield better results in other regards such as the behavior of round trips starting at the DS language. Passing down the current continuation might open up even more possibilities in that respect. Moreover, it would be nice to also consider reduction theories for our calculi, in particular, to be able to reason better about optimizations in the CPS language.

Acknowledgments

The work on this project was supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) – project number DFG-448316946.

References

1 A. W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7. doi: 10.1017/CBO9780511609619.

2 M. Biernacka, D. Biernacki, and S. Lenglet. Typing control operators in the cps hierarchy. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, page 149–160, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307765. doi: 10.1145/2003476.2003498. URL `https://doi.org/10.1145/2003476.2003498`.

3 D. Biernacki, M. Pyzik, and F. Sieczkowski. A Reflection on Continuation-Composing Style. In Z. M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-155-9. doi: 10.4230/LIPIcs.FSCD.2020.18. URL `https://drops.dagstuhl.de/opus/volltexte/2020/12340`.

4 D. Biernacki, M. Pyzik, and F. Sieczkowski. Reflecting stacked continuations in a fine-grained direct-style reduction theory. In *23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP 2021, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386890. doi: 10.1145/3479394.3479399. URL `https://doi.org/10.1145/3479394.3479399`.

5 E. Brady. Idris 2: Quantitative type theory in action. Technical report, University of St Andrews, Scotland, UK, 2020. URL `https://www.type-driven.org.uk/edwinb/papers/idris2.pdf`.

6 Y. Cong, L. Osvald, G. M. Essertel, and T. Rompf. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.*, 3(ICFP):79:1–79:28, July 2019. ISSN 2475-1421. doi: 10.1145/3341643.

7 O. Danvy. Back to direct style. In *Symposium Proceedings on 4th European Symposium on Programming*, ESOP'92, page 130–150, Berlin, Heidelberg, 1992. Springer-Verlag. ISBN 0387552537. doi: 10.1007/3-540-55253-7_8. URL `https://doi.org/10.1007/3-540-55253-7_8`.

8 O. Danvy and J. L. Lawall. Back to direct style ii: First-class continuations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, page 299–310, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897914813. doi: 10.1145/141471.141564. URL `https://doi.org/10.1145/141471.141564`.

9 O. Danvy and L. R. Nielsen. A first-order one-pass cps transformation. *Theor. Comput. Sci.*, 308(1–3):239–257, Nov. 2003. ISSN 0304-3975. doi: 10.1016/S0304-3975(02)00733-8. URL `https://doi.org/10.1016/S0304-3975(02)00733-8`.

10 O. Danvy and F. Pfenning. The occurrence of continuation parameters in CPS terms. Technical Report CMU-CS-95-121, Department of Computer Science, Carnegie Mellon University, Feb 1995.

11 K. Farvardin and J. Reppy. From folklore to fact: Comparing implementations of stacks and continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 75–90, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385994. URL `https://doi.org/10.1145/3385412.3385994`.

12 M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987. ISSN 0304-3975.

doi: 10.1016/0304-3975(87)90109-5. URL https://doi.org/10.1016/0304-3975(87)90109-5.

13 C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 237–247, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915984. doi: 10.1145/155090.155113. URL https://doi.org/10.1145/155090.155113.

14 T. G. Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 47–58, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913434. doi: 10.1145/96709.96714. URL https://doi.org/10.1145/96709.96714.

15 J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 458–471, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916360. doi: 10.1145/174675.178053. URL https://doi.org/10.1145/174675.178053.

16 W. A. Howard. The formulae-as-types notion of construction. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 44:479–490, 1980.

17 Y. Kameyama. Axioms for control operators in the cps hierarchy. *Higher Order Symbol. Comput.*, 20(4):339–369, Dec. 2007. ISSN 1388-3690. doi: 10.1007/s10990-007-9009-x. URL https://doi.org/10.1007/s10990-007-9009-x.

18 Y. Kameyama and M. Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, page 177–188, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137567. doi: 10.1145/944705.944722. URL https://doi.org/10.1145/944705.944722.

19 A. Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 177–190, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938152. doi: 10.1145/1291151.1291179. URL https://doi.org/10.1145/1291151.1291179.

20 O. Kiselyov and C.-c. Shan. A substructural type system for delimited continuations. In S. R. Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 223–239, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73228-0. doi: 10.1007/978-3-540-73228-0_17. URL https://doi.org/10.1007/978-3-540-73228-0_17.

21 J. L. Lawall and O. Danvy. Separating stages in the continuation-passing style transformation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 124–136, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915607. doi: 10.1145/158511.158613. URL https://doi.org/10.1145/158511.158613.

22 P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003. ISSN 0890-5401. doi: 10.1016/S0890-5401(03)00088-9. URL https://doi.org/10.1016/S0890-5401(03)00088-9.

23 M. Materzok and D. Biernacki. A dynamic interpretation of the CPS hierarchy. In R. Jhala and A. Igarashi, editors, *Programming Languages and Systems*, pages 296–311, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-35182-2. doi: 10.1007/978-3-642-35182-2_21. URL https://doi.org/10.1007/978-3-642-35182-2_21.

24 L. Maurer, P. Downen, Z. M. Ariola, and S. L. Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062380. URL `http://doi.acm.org/10.1145/3062341.3062380`.

25 E. Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989. doi: 10.1109/LICS.1989.39155.

26 E. Moggi. Notions of computation and monads. *Information and Computation*, 93 (1):55–92, 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4. URL `https://doi.org/10.1016/0890-5401(91)90052-4`. Selections from 1989 IEEE Symposium on Logic in Computer Science.

27 L. R. Nielsen. A simple correctness proof of the direct-style transformation. *BRICS Report Series*, 9(2), Jan. 2002. doi: 10.7146/brics.v9i2.21719. URL `https://tidsskrift.dk/brics/article/view/21719`.

28 A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, page 288–298, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897914813. doi: 10.1145/141471.141563. URL `https://doi.org/10.1145/141471.141563`.

29 A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6:289–360, 1993. doi: 10.1007/BF01019462. URL `https://doi.org/10.1007/BF01019462`.

30 A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, Nov. 1997. ISSN 0164-0925. doi: 10.1145/267959.269968. URL `https://doi.org/10.1145/267959.269968`.

31 P. Schuster, J. I. Brachthäuser, and K. Ostermann. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.*, 4(ICFP), Aug. 2020. doi: 10.1145/3408975.

Value Typing $\boxed{\vdash_{val} \mathsf{V} : \tau}$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \Gamma, x : \tau \vdash s : \tau_0}{\vdash_{val} \{\, \mathsf{E}, (x : \tau) \Rightarrow s \,\} : \tau \to \tau_0}\ [\textsc{Closure}]$$

$$\frac{\tau \vdash_{stk} \mathsf{K}}{\vdash_{val} \mathsf{K} : \neg\, \tau}\ [\textsc{Stack}] \qquad \frac{}{\vdash_{val} 19 : \mathsf{Int}}\ [\textsc{Int}]$$

Environment Typing $\boxed{\mathsf{E} \vdash_{env} \Gamma}$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \vdash_{val} \mathsf{V} : \tau}{\mathsf{E}, x \mapsto \mathsf{V} \vdash_{env} \Gamma, x : \tau}\ [\textsc{ExtendEnv}] \qquad \frac{}{\bullet \vdash_{env} \emptyset}\ [\textsc{EmptyEnv}]$$

Stack Typing $\boxed{\tau \vdash_{stk} \mathsf{K}}$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \Gamma, x : \tau \vdash s : \tau_0 \qquad \tau_0 \vdash_{stk} \mathsf{K}}{\tau \vdash_{stk} \{\, \mathsf{E}, (x : \tau) \Rightarrow s \,\} :: \mathsf{K}}\ [\textsc{Frame}]$$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \Gamma, x : \tau \vdash s : \#}{\tau \vdash_{stk} \{\, \mathsf{E}, (x : \tau) \Rightarrow s \,\}}\ [\textsc{Underflow}]$$

Configuration Typing $\boxed{\vdash \mathsf{M}}$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \Gamma \vdash s : \tau \qquad \tau \vdash_{stk} \mathsf{K}}{\vdash \langle\, \mathsf{E} \parallel s \parallel \mathsf{K} \,\rangle}\ [\textsc{Delim}]$$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \Gamma \vdash s : \#}{\vdash \langle\, \mathsf{E} \parallel s \,\rangle}\ [\textsc{Undelim}]$$

Figure 20 Typing rules for machine states for λ_{D}.

A Typing of machines

Figure 20 shows the full typing rules for the abstract machine of λ_{D} and Figure 21 shows those of λ_{C}. Not how undelimited stack frames in λ_{D} correspond to closures without continuation parameter in λ_{C}.

Value Typing $\boxed{\vdash_{val}\ \mathsf{V} : \tau}$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \Gamma,\ x : \tau,\ k : \neg\,\tau_0 \vdash\ t_0}{\vdash_{val}\ \{\ \mathsf{E},\ (x : \tau \mid k : \neg\,\tau_0) \Rightarrow t_0\ \}\ :\ \tau \to \tau_0}\ [\textsc{FunClosure}]$$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \Gamma,\ x : \tau \vdash\ t_0}{\vdash_{val}\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ :\ \neg\,\tau}\ [\textsc{CntClosure}] \qquad \frac{}{\vdash_{val}\ 19\ :\ \mathsf{Int}}\ [\textsc{Int}]$$

Environment Typing $\boxed{\mathsf{E} \vdash_{env} \Gamma}$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \vdash_{val}\ \mathsf{V} : \tau}{\mathsf{E},\ x \mapsto \mathsf{V} \vdash_{env} \Gamma,\ x : \tau}\ [\textsc{ExtendEnv}] \qquad \frac{}{\bullet \vdash_{env} \emptyset}\ [\textsc{EmptyEnv}]$$

Configuration Typing $\boxed{\vdash\ \mathsf{M}}$

$$\frac{\mathsf{E} \vdash_{env} \Gamma \qquad \Gamma \vdash\ t}{\vdash\ \langle\, \mathsf{E} \parallel t \,\rangle}\ [\textsc{Execution}]$$

Figure 21 Typing rules for machine states for λ_{C}.

Syntax of Pure λ_{D}

Statements	s	$::=$	$\textbf{val}\ x\ =\ s;\ s$	sequence
		$\mid$	$\textbf{ret}\ e$	return
		$\mid$	$\textbf{def}\ f(x\ :\ \tau)\ \{\ s\ \};\ s$	define
		$\mid$	$f(e)$	call
Variables	v	$::=$	$x,\ f,\ k$	variables
Expressions	e	$::=$	v	variables
		$\mid$	$0\ \mid 1\ \mid \ldots$	integers

Syntax of Types for Pure λ_{D}

Types	τ	$::=$	$\tau \to \tau$	function type
		$\mid$	Int	base type
Environment Type	Γ	$::=$	$\Gamma,\ x\ :\ \tau$	extended environment
		$\mid$	$\emptyset$	empty environment

Syntax of Machine States for Pure λ_{D}

Values	V	$::=$	$\{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow s\ \}$	closure
		$\mid$	$0\ \mid 1\ \mid \ldots$	integer
Environments	E	$::=$	$\mathsf{E},\ x \mapsto \mathsf{V}$	binding
		$\mid$	$\bullet$	empty
Stacks	K	$::=$	$\{\ \mathsf{E},\ (x\ :\ \tau) \Rightarrow s\ \}\ ::\ \mathsf{K}$	frame
		$\mid$	$\{\ \bullet,\ (x\ :\ \tau) \Rightarrow \textbf{exit}\ x\ \}$	done
Configurations	M	$::=$	$\langle\ \mathsf{E}\ \Vert\ s\ \Vert\ \mathsf{K}\ \rangle$	delimited execution

Figure 22 The pure fragment of λ_{D}.

B Pure fragment

Figure 22 shows the syntax of the pure fragment of λ_{D}. The typing rules for statements and the machine are restricted accordingly, as are the reduction steps for the machine. Note that the only undelimited frame is `done` which always is the bottom of the stack.

Statement Typing

$$\frac{\Gamma \vdash\ e\ :\ \mathsf{Bool} \qquad \Gamma \vdash\ s_1\ :\ \xi \qquad \Gamma \vdash\ s_2\ :\ \xi}{\Gamma \vdash\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ :\ \xi}\ [\textsc{If}]$$

Term Typing

$$\frac{\Gamma \vdash\ e\ :\ \mathsf{Bool} \qquad \Gamma \vdash\ t_1 \qquad \Gamma \vdash\ t_2}{\Gamma \vdash\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2}\ [\textsc{If}]$$

Expression Typing

$$\frac{}{\Gamma \vdash\ \mathsf{true}\ :\ \mathsf{Bool}}\ [\textsc{True}] \qquad \frac{}{\Gamma \vdash\ \mathsf{false}\ :\ \mathsf{Bool}}\ [\textsc{False}]$$

Figure 23 Typing rules for the extension of the languages with conditionals.

Machine Steps of λ_D

(iftru-1)	$\langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel s_1 \parallel \mathsf{K}\ \rangle$
(iftru-0)	$\langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle$	$\rightarrow$	$\langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel s_1\ \rangle$
(iffls-1)	$\langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K}\ \rangle$	$\rightarrow$	$\langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel s_2 \parallel \mathsf{K}\ \rangle$
(iffls-0)	$\langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle$	$\rightarrow$	$\langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel s_2\ \rangle$

Machine Steps of λ_C

(iftru)	$\langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle$	$\rightarrow$	$\langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel t_1\ \rangle$
(iffls)	$\langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle$	$\rightarrow$	$\langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel t_2\ \rangle$

Figure 24 Machine steps for the extension of the languages with conditionals.

C Conditionals

Figure 23 shows the typing rules for conditionals in both languages which are straightforward. The same is true for the additional machine steps displayed in Figure 24.

D Proofs

We now give the proofs omitted in the paper. Note that by type soundness and typability preservation everything in sight is well-typed. However, typing is not essential and the proofs would work similarly in an untyped setting by replacing induction on typing derivations by induction the syntactic structure.

D.1 Simulation (Semantics Preservation)

We prove that $\mathcal{C}_M[\![\ \cdot\ ]\!]$ (Theorem 8) and $\mathcal{D}_M[\![\ \cdot\ ]\!]$ (Theorem 12) are step-preserving and the corollaries for evaluation (Theorems 11 and 14).

D.1.1 CPS Translation

We start with the CPS translation. For every step the DS-machine takes, the CPS-machine either takes no step or one step.

Proof. We distinguish cases according to the machine steps.

case *(push)*
We have

$$\langle\ \mathsf{E} \parallel \textbf{val}\ x_0\ =\ s_0;\ s \parallel \mathsf{K}\ \rangle \to \langle\ \mathsf{E} \parallel s_0 \parallel \{\ \mathsf{E},\ (x_0) \Rightarrow s\ \}\ ::\ \mathsf{K}\ \rangle$$

The translations of the machine configurations are

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{val}\ x_0\ =\ s_0;\ s \parallel \mathsf{K}\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ \textbf{val}\ x_0\ =\ s_0;\ s\ ]\!]_k\ \rangle && (k \text{ is fresh})\\
&=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \textbf{cnt}\ k_0(x_0)\ \{\ \mathcal{C}[\![\ s\ ]\!]_k\ \};\ \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel s_0 \parallel \{\ \mathsf{E},\ (x_0) \Rightarrow s\ \}\ ::\ \mathsf{K}\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \mathcal{C}_K[\![\ \{\ \mathsf{E},\ (x_0) \Rightarrow s\ \}\ ::\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \rangle && (k_0 \text{ is fresh})\\
&=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!],\ (x_0) \Rightarrow \mathcal{C}[\![\ s\ ]\!]_k\ \} \parallel \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \rangle && (k \text{ is fresh})
\end{aligned}$$

By rule *(cnt)* we have

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{val}\ x_0\ =\ s_0;\ s \parallel \mathsf{K}\ \rangle\ ]\!]\\
&\to \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!],\ k_0 \mapsto \{\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!],\ (x_0) \Rightarrow \mathcal{C}[\![\ s\ ]\!]_k\ \} \parallel \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \rangle
\end{aligned}$$

Up to the additional mapping for k this is just what we want and since k was chosen fresh, it is not free in $\mathcal{C}[\![\ s_0\ ]\!]_{k_0}$ we can identify the two configurations by weakening

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel s_0 \parallel \{\ \mathsf{E},\ (x_0) \Rightarrow s\ \}\ ::\ \mathsf{K}\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!],\ k_0 \mapsto \{\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!],\ (x_0) \Rightarrow \mathcal{C}[\![\ s\ ]\!]_k\ \} \parallel \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \rangle
\end{aligned}$$

case *(retv-1)*
We have

$$\langle\ \mathsf{E},\ v \mapsto \mathsf{V} \parallel \textbf{ret}\ v \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ::\ \mathsf{K}\ \rangle \to \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \parallel \mathsf{K}\ \rangle$$

The translations of the machine configurations are

$$\begin{array}{lll} \mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{V} \parallel \textbf{ret}\ v \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ::\ \mathsf{K}\,\rangle\,]\!] & & \\ \quad = \langle\, \mathcal{C}_E[\![\, \mathsf{E}\,]\!],\ v \mapsto \mathcal{C}_V[\![\,\mathsf{V}\,]\!],\ k \mapsto \mathcal{C}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ::\ \mathsf{K}\,]\!] \parallel \mathcal{C}[\![\, \textbf{ret}\ v\,]\!]_k\,\rangle & & (k \text{ is fresh}) \\ \quad = \langle\, \mathcal{C}_E[\![\, \mathsf{E}\,]\!],\ v \mapsto \mathcal{C}_V[\![\,\mathsf{V}\,]\!],\ k \mapsto \{\ \mathcal{C}_E[\![\,\mathsf{E}_0\,]\!],\ k_0 \mapsto \mathcal{C}_K[\![\,\mathsf{K}\,]\!],\ (x) \Rightarrow \mathcal{C}[\![\, s\,]\!]_{k_0}\ \} \parallel k(v)\,\rangle & & (k_0 \text{ is fresh}) \end{array}$$

and

$$\begin{array}{ll} \mathcal{C}_M[\![\, \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \parallel \mathsf{K}\,\rangle\,]\!] & \\ \quad = \langle\, \mathcal{C}_E[\![\,\mathsf{E}_0\,]\!],\ x \mapsto \mathcal{C}_V[\![\,\mathsf{V}\,]\!],\ k_0 \mapsto \mathcal{C}_K[\![\,\mathsf{K}\,]\!] \parallel \mathcal{C}[\![\, s\,]\!]_{k_0}\,\rangle & (k_0 \text{ is fresh}) \end{array}$$

By rule *(jmpv)* we thus obtain (since environments are unordered)

$$\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{V} \parallel \textbf{ret}\ v \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ::\ \mathsf{K}\,\rangle\,]\!] \to \mathcal{C}_M[\![\, \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \parallel \mathsf{K}\,\rangle\,]\!]$$

case *(retv-0)*
We have

$$\langle\, \mathsf{E},\ v \mapsto \mathsf{V} \parallel \textbf{ret}\ v \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\,\rangle \to \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s\,\rangle$$

The translations of the machine configurations are

$$\begin{array}{ll} \mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{V} \parallel \textbf{ret}\ v \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\,\rangle\,]\!] & \\ \quad = \langle\, \mathcal{C}_E[\![\, \mathsf{E}\,]\!],\ v \mapsto \mathcal{C}_V[\![\,\mathsf{V}\,]\!],\ k \mapsto \mathcal{C}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\,]\!] \parallel \mathcal{C}[\![\, \textbf{ret}\ v\,]\!]_k\,\rangle & (k \text{ is fresh}) \\ \quad = \langle\, \mathcal{C}_E[\![\, \mathsf{E}\,]\!],\ v \mapsto \mathcal{C}_V[\![\,\mathsf{V}\,]\!],\ k \mapsto \{\ \mathcal{C}_E[\![\,\mathsf{E}_0\,]\!],\ (x) \Rightarrow \mathcal{C}[\![\, s\,]\!]_{\bullet}\ \} \parallel k(v)\,\rangle & \end{array}$$

and

$$\begin{array}{l} \mathcal{C}_M[\![\, \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s\,\rangle\,]\!] \\ \quad = \langle\, \mathcal{C}_E[\![\,\mathsf{E}_0\,]\!],\ x \mapsto \mathcal{C}_V[\![\,\mathsf{V}\,]\!] \parallel \mathcal{C}[\![\, s\,]\!]_{\bullet}\,\rangle \end{array}$$

By rule *(jmpv)* we thus obtain

$$\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{V} \parallel \textbf{ret}\ v \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\,\rangle\,]\!] \to \mathcal{C}_M[\![\, \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s\,\rangle\,]\!]$$

case *(reti-1)*
We have

$$\langle\, \mathsf{E} \parallel \textbf{ret}\ 19 \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ::\ \mathsf{K}\,\rangle \to \langle\, \mathsf{E}_0,\ x \mapsto 19 \parallel s \parallel \mathsf{K}\,\rangle$$

The translations of the machine configurations are

$$\begin{array}{ll} \mathcal{C}_M[\![\, \langle\, \mathsf{E} \parallel \textbf{ret}\ 19 \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ::\ \mathsf{K}\,\rangle\,]\!] & \\ \quad = \langle\, \mathcal{C}_E[\![\, \mathsf{E}\,]\!],\ k \mapsto \mathcal{C}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ::\ \mathsf{K}\,]\!] \parallel \mathcal{C}[\![\, \textbf{ret}\ 19\,]\!]_k\,\rangle & (k \text{ is fresh}) \\ \quad = \langle\, \mathcal{C}_E[\![\, \mathsf{E}\,]\!],\ k \mapsto \{\ \mathcal{C}_E[\![\,\mathsf{E}_0\,]\!],\ k_0 \mapsto \mathcal{C}_K[\![\,\mathsf{K}\,]\!],\ (x) \Rightarrow \mathcal{C}[\![\, s\,]\!]_{k_0}\ \} \parallel k(19)\,\rangle & (k_0 \text{ is fresh}) \end{array}$$

and

$$\begin{array}{ll} \mathcal{C}_M[\![\, \langle\, \mathsf{E}_0,\ x \mapsto 19 \parallel s \parallel \mathsf{K}\,\rangle\,]\!] & \\ \quad = \langle\, \mathcal{C}_E[\![\,\mathsf{E}_0\,]\!],\ x \mapsto 19,\ k_0 \mapsto \mathcal{C}_K[\![\,\mathsf{K}\,]\!] \parallel \mathcal{C}[\![\, s\,]\!]_{k_0}\,\rangle & (k_0 \text{ is fresh}) \end{array}$$

By rule *(jmpi)* we thus obtain (since environments are unordered)

$$\mathcal{C}_M[\![\, \langle\, \mathsf{E} \parallel \textbf{ret}\ 19 \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ::\ \mathsf{K}\,\rangle\,]\!] \to \mathcal{C}_M[\![\, \langle\, \mathsf{E}_0,\ x \mapsto 19 \parallel s \parallel \mathsf{K}\,\rangle\,]\!]$$

case *(reti-0)*
We have

$$\langle\, \mathsf{E} \parallel \textbf{ret}\ 19 \parallel \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\,\rangle \to \langle\, \mathsf{E}_0,\ x \mapsto 19 \parallel s\,\rangle$$

The translations of the machine configurations are

$$\begin{aligned}
&\mathcal{C}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{ret } 19 \parallel \{ \, \mathsf{E}_0, \ (x) \Rightarrow s \, \} \, \rangle \,]\!] \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ k \mapsto \mathcal{C}_K[\![\, \{ \, \mathsf{E}_0, \ (x) \Rightarrow s \, \} \,]\!] \parallel \mathcal{C}[\![\, \textbf{ret } 19 \,]\!]_k \, \rangle && (k \text{ is fresh}) \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ k \mapsto \{ \, \mathcal{C}_E[\![\, \mathsf{E}_0 \,]\!], \ (x) \Rightarrow \mathcal{C}[\![\, s \,]\!]_\bullet \, \} \parallel k(19) \, \rangle
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{C}_M[\![\, \langle \, \mathsf{E}_0, \ x \mapsto 19 \parallel s \, \rangle \,]\!] \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E}_0 \,]\!], \ x \mapsto 19 \parallel \mathcal{C}[\![\, s \,]\!]_\bullet \, \rangle
\end{aligned}$$

By rule *(jmpi)* we thus obtain

$$\mathcal{C}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{ret } 19 \parallel \{ \, \mathsf{E}_0, \ (x) \Rightarrow s \, \} \, \rangle \,]\!] \to \mathcal{C}_M[\![\, \langle \, \mathsf{E}_0, \ x \mapsto 19 \parallel s \, \rangle \,]\!]$$

case *(def-1)*
We have

$$\langle \, \mathsf{E} \parallel \textbf{def } f(x) \ \{ \ s_0 \ \}; \ s \parallel \mathsf{K} \, \rangle \to \langle \, \mathsf{E}, \ f \mapsto \{ \, \mathsf{E}, \ (x) \Rightarrow s_0 \, \} \parallel s \parallel \mathsf{K} \, \rangle$$

The translations of the machine configurations are

$$\begin{aligned}
&\mathcal{C}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{def } f(x) \ \{ \ s_0 \ \}; \ s \parallel \mathsf{K} \, \rangle \, \rangle \,]\!] \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \mathcal{C}[\![\, \textbf{def } f(x) \ \{ \ s_0 \ \}; \ s \,]\!]_k \, \rangle && (k \text{ is fresh}) \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \textbf{let } f(x \mid k_0) \ \{ \ \mathcal{C}[\![\, s_0 \,]\!]_{k_0} \ \}; \ \mathcal{C}[\![\, s \,]\!]_k \, \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{C}_M[\![\, \langle \, \mathsf{E}, \ f \mapsto \{ \, \mathsf{E}, \ (x) \Rightarrow s_0 \, \} \parallel s \parallel \mathsf{K} \, \rangle \,]\!] \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ f \mapsto \mathcal{C}_V[\![\, \{ \, \mathsf{E}, \ (x) \Rightarrow s_0 \, \} \,]\!], \ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \mathcal{C}[\![\, s \,]\!]_k \, \rangle && (k \text{ is fresh}) \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ f \mapsto \{ \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ (x \mid k_0) \Rightarrow \mathcal{C}[\![\, s_0 \,]\!]_{k_0} \, \}, \ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \mathcal{C}[\![\, s \,]\!]_k \, \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

By rule *(let)* we have

$$\begin{aligned}
&\mathcal{C}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{def } f(x) \ \{ \ s_0 \ \}; \ s \parallel \mathsf{K} \, \rangle \, \rangle \,]\!] \\
&\to \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!], \ f \mapsto \{ \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!], \ (x \mid k_0) \Rightarrow \mathcal{C}[\![\, s_0 \,]\!]_{k_0} \, \} \parallel \mathcal{C}[\![\, s \,]\!]_k \, \rangle
\end{aligned}$$

Up to the additional mapping for k in the definition of f this is just what we want and since k was chosen fresh, it is not free in $\mathcal{C}[\![\, s_0 \,]\!]_{k_0}$ and we can identify the two definitions of f by weakening and hence also the configurations.

case *(def-0)*
We have

$$\langle \, \mathsf{E} \parallel \textbf{def } f(x) \ \{ \ s_0 \ \}; \ s \, \rangle \to \langle \, \mathsf{E}, \ f \mapsto \{ \, \mathsf{E}, \ (x) \Rightarrow s_0 \, \} \parallel s \, \rangle$$

The translations of the machine configurations are

$$\begin{aligned}
&\mathcal{C}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{def } f(x) \ \{ \ s_0 \ \}; \ s \, \rangle \, \rangle \,]\!] \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!] \parallel \mathcal{C}[\![\, \textbf{def } f(x) \ \{ \ s_0 \ \}; \ s \,]\!]_\bullet \, \rangle \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!] \parallel \textbf{let } f(x \mid k_0) \ \{ \ \mathcal{C}[\![\, s_0 \,]\!]_{k_0} \ \}; \ \mathcal{C}[\![\, s \,]\!]_\bullet \, \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{C}_M[\![\, \langle \, \mathsf{E}, \ f \mapsto \{ \, \mathsf{E}, \ (x) \Rightarrow s_0 \, \} \parallel s \, \rangle \,]\!] \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ f \mapsto \mathcal{C}_V[\![\, \{ \, \mathsf{E}, \ (x) \Rightarrow s_0 \, \} \,]\!] \parallel \mathcal{C}[\![\, s \,]\!]_\bullet \, \rangle \\
&= \ \langle \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ f \mapsto \{ \, \mathcal{C}_E[\![\, \mathsf{E} \,]\!], \ (x \mid k_0) \Rightarrow \mathcal{C}[\![\, s_0 \,]\!]_{k_0} \, \} \parallel \mathcal{C}[\![\, s \,]\!]_\bullet \, \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

By rule *(let)* we thus have

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \mathbf{def}\ f(x)\ \{\ s_0\ \};\ s \parallel \mathsf{K}\ \rangle\ ]\!] \to \mathcal{C}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x) \Rightarrow s_0\ \} \parallel s\ \rangle\ ]\!]$$

case *(callv)*

We have

$$\langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \},\ v \mapsto \mathsf{V} \parallel f(v) \parallel \mathsf{K}\ \rangle \to \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \parallel \mathsf{K}\ \rangle$$

The translations of the machine configurations are

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \},\ v \mapsto \mathsf{V} \parallel f(v) \parallel \mathsf{K}\ \rangle\ ]\!] \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{C}_V[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ]\!],\ v \mapsto \mathcal{C}_V[\![\ \mathsf{V}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ f(v)\ ]\!]_k\ \rangle && (k \text{ is fresh}) \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \{\ \mathcal{C}_E[\![\ \mathsf{E}_0\ ]\!],\ (x \mid k_0) \Rightarrow \mathcal{C}[\![\ s\ ]\!]_{k_0}\ \},\ v \mapsto \mathcal{C}_V[\![\ \mathsf{V}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel f(v \mid k)\ \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \parallel \mathsf{K}\ \rangle\ ]\!] \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{C}_V[\![\ \mathsf{V}\ ]\!],\ k_0 \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_{k_0}\ \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

By rule *(appv)* we thus obtain

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \},\ v \mapsto \mathsf{V} \parallel f(v) \parallel \mathsf{K}\ \rangle\ ]\!] \to \mathcal{C}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel s \parallel \mathsf{K}\ \rangle\ ]\!]$$

case *(calli)*

We have

$$\langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \} \parallel f(19) \parallel \mathsf{K}\ \rangle \to \langle\ \mathsf{E}_0,\ x \mapsto 19 \parallel s \parallel \mathsf{K}\ \rangle$$

The translations of the machine configurations are

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \} \parallel f(19) \parallel \mathsf{K}\ \rangle\ ]\!] \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{C}_V[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ f(19)\ ]\!]_k\ \rangle && (k \text{ is fresh}) \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \{\ \mathcal{C}_E[\![\ \mathsf{E}_0\ ]\!],\ (x \mid k_0) \Rightarrow \mathcal{C}[\![\ s\ ]\!]_{k_0}\ \},\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel f(19 \mid k)\ \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto 19 \parallel s \parallel \mathsf{K}\ \rangle\ ]\!] \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 10,\ k_0 \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_{k_0}\ \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

By rule *(appi)* we thus obtain

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow s\ \} \parallel f(19) \parallel \mathsf{K}\ \rangle\ ]\!] \to \mathcal{C}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto 19 \parallel s \parallel \mathsf{K}\ \rangle\ ]\!]$$

case *(proc-1)*

We have

$$\langle\ \mathsf{E} \parallel \mathbf{process}\ k_0(x)\ \{\ s_0\ \};\ s \parallel \mathsf{K}\ \rangle \to \langle\ \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow s_0\ \} \parallel s \parallel \mathsf{K}\ \rangle$$

The translations of the machine configurations are

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \mathbf{process}\ k_0(x)\ \{\ s_0\ \};\ s \parallel \mathsf{K}\ \rangle\ ]\!] \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ \mathbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ ]\!]_k\ \rangle && (k \text{ is fresh}) \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{\bullet}\ \};\ \mathcal{C}[\![\ s\ ]\!]_k\ \rangle && (k_0 \text{ is fresh})
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{C}_M[\![\ \langle\ \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow s_0\ \} \parallel s \parallel \mathsf{K}\ \rangle\ ]\!] \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \mathcal{C}_K[\![\ \{\ \mathsf{E},\ (x) \Rightarrow s_0\ \}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_k\ \rangle && (k \text{ is fresh}) \\
&= \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \mathcal{C}[\![\ s_0\ ]\!]_{\bullet}\ \},\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_k\ \rangle
\end{aligned}$$

By rule *(cnt)* we have

$$\begin{array}{l} \mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s \parallel \mathsf{K}\ \rangle\ ]\!] \\ \to \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!],\ k_0 \mapsto \{\ \ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!],\ (x) \Rightarrow \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \} \parallel \mathcal{C}[\![\ s\ ]\!]_k\ \rangle \end{array}$$

Up to the additional mapping for k in the definition of k_0 this is just what we want and since k was chosen fresh, it is not free in $\mathcal{C}[\![\ s_0\ ]\!]_\bullet$ and we can identify the two definitions of k_0 by weakening and hence also the configurations.

case *(proc-0)*
We have

$$\langle\ \mathsf{E} \parallel \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ \rangle \to \langle\ \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow s_0\ \} \parallel s\ \rangle$$

The translations of the machine configurations are

$$\begin{array}{ll} \mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ \rangle\ ]\!] & \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!] \parallel \mathcal{C}[\![\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ ]\!]_\bullet\ \rangle & \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \};\ \mathcal{C}[\![\ s\ ]\!]_\bullet\ \rangle & (k_0 \text{ is fresh}) \end{array}$$

and

$$\begin{array}{l} \mathcal{C}_M[\![\ \langle\ \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow s_0\ \} \parallel s\ \rangle\ ]\!] \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \mathcal{C}_K[\![\ \{\ \mathsf{E},\ (x) \Rightarrow s_0\ \}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_\bullet\ \rangle \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \} \parallel \mathcal{C}[\![\ s\ ]\!]_\bullet\ \rangle \end{array}$$

By rule *(cnt)* we thus obtain

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ \rangle\ ]\!] \to \mathcal{C}_M[\![\ \langle\ \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow s_0\ \} \parallel s\ \rangle\ ]\!]$$

case *(sus)*
We have

$$\langle\ \mathsf{E} \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \} \parallel \mathsf{K}\ \rangle \to \langle\ \mathsf{E},\ k_0 \mapsto \mathsf{K} \parallel s\ \rangle$$

The translations of the machine configurations are

$$\begin{array}{ll} \mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \} \parallel \mathsf{K}\ \rangle\ ]\!] & \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \}\ ]\!]_k\ \rangle & (k \text{ is fresh}) \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_\bullet\ \rangle & (\alpha\text{-renaming}) \end{array}$$

and

$$\begin{array}{l} \mathcal{C}_M[\![\ \langle\ \mathsf{E},\ k_0 \mapsto \mathsf{K} \parallel s\ \rangle\ ]\!] \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_\bullet\ \rangle \end{array}$$

Both translations are the same, thus we obtain

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \} \parallel \mathsf{K}\ \rangle\ ]\!] \to^? \mathcal{C}_M[\![\ \langle\ \mathsf{E},\ k_0 \mapsto \mathsf{K} \parallel s\ \rangle\ ]\!]$$

case *(run)*
We have

$$\langle\ \mathsf{E},\ v \mapsto \mathsf{K} \parallel \textbf{run}(v)\ \{\ s\ \}\ \rangle \to \langle\ \mathsf{E},\ v \mapsto \mathsf{K} \parallel s \parallel \mathsf{K}\ \rangle$$

The translations of the machine configurations are

$$\begin{array}{l} \mathcal{C}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{K} \parallel \textbf{run}(v)\ \{\ s\ \}\ \rangle\ ]\!] \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ \textbf{run}(v)\ \{\ s\ \}\ ]\!]_\bullet\ \rangle \\ \quad = \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_v\ \rangle \end{array}$$

and

$$\begin{aligned}&\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{K} \parallel s \parallel \mathsf{K} \,\rangle \,]\!] \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ v \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!],\ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \mathcal{C}[\![\, s \,]\!]_k \,\rangle \qquad (k \text{ is fresh})\end{aligned}$$

Since k is fresh and is bound to the same value as v, we can α-rename k to v and then contract k and v in the environment to obtain the same configuration as above. Thus, both translations are the same and we obtain

$$\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{K} \parallel \textbf{run}(v)\ \{\ s\ \} \,\rangle \,]\!] \to^? \mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{K} \parallel s \parallel \mathsf{K} \,\rangle \,]\!]$$

case *(iftruv-1)*
We have

$$\langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K} \,\rangle \to \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel s_1 \parallel \mathsf{K} \,\rangle$$

The translations of the machine configurations are

$$\begin{aligned}&\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K} \,\rangle \,]\!] \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ v \mapsto \mathsf{true},\ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \mathcal{C}[\![\, \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,]\!]_k \,\rangle \qquad (k \text{ is fresh}) \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ v \mapsto \mathsf{true},\ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![\, s_1 \,]\!]_k\ \textbf{else}\ \mathcal{C}[\![\, s_2 \,]\!]_k \,\rangle\end{aligned}$$

and

$$\begin{aligned}&\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel s_1 \parallel \mathsf{K} \,\rangle \,]\!] \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ v \mapsto \mathsf{true},\ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \mathcal{C}[\![\, s_1 \,]\!]_k \,\rangle \qquad (k \text{ is fresh})\end{aligned}$$

By rule *(iftruv)* we thus obtain (since environments are unordered)

$$\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K} \,\rangle \,]\!] \to \mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel s_1 \parallel \mathsf{K} \,\rangle \,]\!]$$

case *(iftruv-0)*
We have

$$\langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\rangle \to \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel s_1 \,\rangle$$

The translations of the machine configurations are

$$\begin{aligned}&\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\rangle \,]\!] \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ v \mapsto \mathsf{true} \parallel \mathcal{C}[\![\, \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,]\!]_\bullet \,\rangle \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![\, s_1 \,]\!]_\bullet\ \textbf{else}\ \mathcal{C}[\![\, s_2 \,]\!]_\bullet \,\rangle\end{aligned}$$

and

$$\begin{aligned}&\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel s_1 \,\rangle \,]\!] \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ v \mapsto \mathsf{true} \parallel \mathcal{C}[\![\, s_1 \,]\!]_\bullet \,\rangle\end{aligned}$$

By rule *(iftruv)* we thus obtain

$$\mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\rangle \,]\!] \to \mathcal{C}_M[\![\, \langle\, \mathsf{E},\ v \mapsto \mathsf{true} \parallel s_1 \,\rangle \,]\!]$$

case *(iftrub-1)*
We have

$$\langle\, \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K} \,\rangle \to \langle\, \mathsf{E} \parallel s_1 \parallel \mathsf{K} \,\rangle$$

The translations of the machine configurations are

$$\begin{aligned}&\mathcal{C}_M[\![\, \langle\, \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K} \,\rangle \,]\!] \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \mathcal{C}[\![\, \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,]\!]_k \,\rangle \qquad (k \text{ is fresh}) \\ &= \langle\, \mathcal{C}_E[\![\, \mathsf{E} \,]\!],\ k \mapsto \mathcal{C}_K[\![\, \mathsf{K} \,]\!] \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ \mathcal{C}[\![\, s_1 \,]\!]_k\ \textbf{else}\ \mathcal{C}[\![\, s_2 \,]\!]_k \,\rangle\end{aligned}$$

and

$$\begin{array}{l}\mathcal{C}_M[\![\ \langle\, \mathsf{E} \parallel s_1 \parallel \mathsf{K}\,\rangle \ ]\!] \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K} \ ]\!] \parallel \mathcal{C}[\![\ s_1 \ ]\!]_k \,\rangle \qquad (k \text{ is fresh})\end{array}$$

By rule *(iftrub)* we thus obtain (since environments are unordered)

$$\mathcal{C}_M[\![\ \langle\, \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K}\,\rangle \ ]\!] \to \mathcal{C}_M[\![\ \langle\, \mathsf{E} \parallel s_1 \parallel \mathsf{K}\,\rangle \ ]\!]$$

case *(iftrub-0)*
We have

$$\langle\, \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\rangle \to \langle\, \mathsf{E}\parallel s_1 \,\rangle$$

The translations of the machine configurations are

$$\begin{array}{l}\mathcal{C}_M[\![\ \langle\, \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\rangle \ ]\!] \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!] \parallel \mathcal{C}[\![\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \ ]\!]_\bullet \,\rangle \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!] \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ \mathcal{C}[\![\ s_1 \ ]\!]_\bullet\ \textbf{else}\ \mathcal{C}[\![\ s_2 \ ]\!]_\bullet \,\rangle\end{array}$$

and

$$\begin{array}{l}\mathcal{C}_M[\![\ \langle\, \mathsf{E} \parallel s_1 \,\rangle \ ]\!] \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!] \parallel \mathcal{C}[\![\ s_1 \ ]\!]_\bullet \,\rangle\end{array}$$

By rule *(iftrub)* we thus obtain

$$\mathcal{C}_M[\![\ \langle\, \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\rangle \ ]\!] \to \mathcal{C}_M[\![\ \langle\, \mathsf{E} \parallel s_1 \,\rangle \ ]\!]$$

case *(ifflsv-1)*
We have

$$\langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K}\,\rangle \to \langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel s_2 \parallel \mathsf{K}\,\rangle$$

The translations of the machine configurations are

$$\begin{array}{l}\mathcal{C}_M[\![\ \langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K}\,\rangle \ ]\!] \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!],\ v \mapsto \mathsf{false},\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K} \ ]\!] \parallel \mathcal{C}[\![\ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \ ]\!]_k \,\rangle \qquad (k \text{ is fresh}) \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!],\ v \mapsto \mathsf{false},\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K} \ ]\!] \parallel \textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![\ s_1 \ ]\!]_k\ \textbf{else}\ \mathcal{C}[\![\ s_2 \ ]\!]_k \,\rangle\end{array}$$

and

$$\begin{array}{l}\mathcal{C}_M[\![\ \langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel s_2 \parallel \mathsf{K}\,\rangle \ ]\!] \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!],\ v \mapsto \mathsf{false},\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K} \ ]\!] \parallel \mathcal{C}[\![\ s_2 \ ]\!]_k \,\rangle \qquad (k \text{ is fresh})\end{array}$$

By rule *(ifflsv)* we thus obtain (since environments are unordered)

$$\mathcal{C}_M[\![\ \langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathsf{K}\,\rangle \ ]\!] \to \mathcal{C}_M[\![\ \langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel s_2 \parallel \mathsf{K}\,\rangle \ ]\!]$$

case *(ifflsv-0)*
We have

$$\langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\rangle \to \langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel s_2 \,\rangle$$

The translations of the machine configurations are

$$\begin{array}{l}\mathcal{C}_M[\![\ \langle\, \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\rangle \ ]\!] \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!],\ v \mapsto \mathsf{false} \parallel \mathcal{C}[\![\ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \ ]\!]_\bullet \,\rangle \\ \ = \ \langle\, \mathcal{C}_E[\![\ \mathsf{E} \ ]\!],\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![\ s_1 \ ]\!]_\bullet\ \textbf{else}\ \mathcal{C}[\![\ s_2 \ ]\!]_\bullet \,\rangle\end{array}$$

and

$$\begin{aligned}&\mathcal{C}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false}\ \|\ s_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{false}\ \|\ \mathcal{C}[\![\ s_2\ ]\!]_\bullet\ \rangle\end{aligned}$$

By rule *(ifflsv)* we thus obtain

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false}\ \|\ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle\ ]\!] \to \mathcal{C}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false}\ \|\ s_2\ \rangle\ ]\!]$$

case *(ifflsb-1)*
We have

$$\langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \|\ \mathsf{K}\ \rangle \to \langle\ \mathsf{E}\ \|\ s_2\ \|\ \mathsf{K}\ \rangle$$

The translations of the machine configurations are

$$\begin{aligned}&\mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \|\ \mathsf{K}\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!]\ \|\ \mathcal{C}[\![\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ ]\!]_k\ \rangle \qquad (k \text{ is fresh})\\ &=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!]\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_k\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_k\ \rangle\end{aligned}$$

and

$$\begin{aligned}&\mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ s_2\ \|\ \mathsf{K}\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!]\ \|\ \mathcal{C}[\![\ s_2\ ]\!]_k\ \rangle \qquad (k \text{ is fresh})\end{aligned}$$

By rule *(ifflsb)* we thus obtain (since environments are unordered)

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \|\ \mathsf{K}\ \rangle\ ]\!] \to \mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ s_2\ \|\ \mathsf{K}\ \rangle\ ]\!]$$

case *(ifflsb-0)*
We have

$$\langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle \to \langle\ \mathsf{E}\|\ s_2\ \rangle$$

The translations of the machine configurations are

$$\begin{aligned}&\mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \mathcal{C}[\![\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ ]\!]_\bullet\ \rangle\\ &=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_\bullet\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_\bullet\ \rangle\end{aligned}$$

and

$$\begin{aligned}&\mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ s_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \mathcal{C}[\![\ s_2\ ]\!]_\bullet\ \rangle\end{aligned}$$

By rule *(ifflsb)* we thus obtain

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle\ ]\!] \to \mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ s_2\ \rangle\ ]\!]$$

◀

As a corollary we obtain that a closed term that evaluates to a result in DS, evaluates to the same result in CPS in at most as many steps.

Proof. Note that

$$\mathcal{C}_K[\![\ \mathsf{done}\ ]\!]\ =\ \{\ \mathcal{C}_E[\![\ \bullet\ ]\!],\ (x\ :\ \tau) \Rightarrow \mathcal{C}[\![\ \textbf{exit}\ x\ ]\!]_\bullet\ \}\ =\ \mathsf{done}$$

Thus, since

$$\mathcal{C}_M[\![\ \langle\ \bullet\ \|\ s\ \|\ \mathsf{done}\ \rangle\ ]\!]\ =\ \langle\ k \mapsto \mathsf{done}\ \|\ \mathcal{C}[\![\ s\ ]\!]_k\ \rangle$$

and

$$\mathcal{C}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{exit}\ e\ \rangle\ ]\!]\ =\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{exit}\ e\ \rangle$$

this follows by Corollary 10. ◀

D.1.2 DS Translation

Now we prove simulation of the DS translation. For every step the CPS-machine takes, the DS-machine takes at least one and at most four steps.

Proof. We distinguish cases according to the machine steps and then subcases according to the translation of terms.

case *(appv)*
We have

$$\langle\, \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ v \mapsto \mathsf{V},\ k \mapsto W \parallel f(v \mid k)\,\rangle \to \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V},\ k_0 \mapsto W \parallel t\,\rangle$$

We distinguish whether $k \in \mathsf{FV}(f(v))$ or not.
If so, we obtain by rule *(run)* that

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\, \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ v \mapsto \mathsf{V},\ k \mapsto W \parallel f(v \mid k)\,\rangle\ ]\!]\\ &= \langle\, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!],\ v \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!],\ k \mapsto \mathcal{D}_V[\![\ W\ ]\!] \parallel \textbf{run}(k)\ \{\ f(v)\ \}\,\rangle\\ &\to \langle\, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!],\ v \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!],\ k \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel f(v) \parallel \mathcal{D}_K[\![\ W\ ]\!]\,\rangle\end{aligned}$$

If not, we have

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\, \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ v \mapsto \mathsf{V},\ k \mapsto W \parallel f(v \mid k)\,\rangle\ ]\!]\\ &= \langle\, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!],\ v \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel f(v) \parallel \mathcal{D}_K[\![\ W\ ]\!]\,\rangle\end{aligned}$$

We now distinguish cases according to the translation of t.

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$:
In this case we have

$$\begin{aligned}&\mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!]\\ &= \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \}\ \}\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V},\ k_0 \mapsto W \parallel t\,\rangle\ ]\!]\\ &= \langle\, \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!],\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel s\,\rangle\end{aligned}$$

By rules *(callv)* and *(sus)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\, \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ v \mapsto \mathsf{V},\ k \mapsto W \parallel f(v \mid k)\,\rangle\ ]\!]\\ &\to^{1,2} \langle\, \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \} \parallel \mathcal{D}_K[\![\ W\ ]\!]\,\rangle\\ &\to \langle\, \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!],\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel s\,\rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k_0$:
In this case we have

$$\begin{aligned}&\mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!]\\ &= \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow s\ \}\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\, \mathsf{E}_0,\ x \mapsto \mathsf{V},\ k_0 \mapsto W \parallel t\,\rangle\ ]\!]\\ &= \langle\, \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ W\ ]\!]\,\rangle\end{aligned}$$

By rule *(callv)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\, \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ v \mapsto \mathsf{V},\ k \mapsto W \parallel f(v \mid k)\,\rangle\ ]\!]\\ &\to^{1,2} \langle\, \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ W\ ]\!]\,\rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t\ ]\!] \;=\; s \rightsquigarrow v_0$ where $v_0 \;\neq\; k_0$:
In this case we have

$$\begin{aligned}&\mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!]\\ &=\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s\ \}\ \}\ \}\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V},\ k_0 \mapsto W \parallel t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!].\mathsf{rm}(v_0),\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!],\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle\end{aligned}$$

By rules *(callv)*, *(sus)* and *(run)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ v \mapsto \mathsf{V},\ k \mapsto W \parallel f(v \mid k)\ \rangle\ ]\!]\\ &\to^{1,2} \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s\ \}\ \} \parallel \mathcal{D}_K[\![\ W\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!],\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel \textbf{run}(v_0)\ \{\ s\ \}\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!],\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle\end{aligned}$$

Apart from the binding for v_0 in the environment this is just what we want and since v_0 is not free in s we can identify the two configurations by weakening.

case *(appi)*
We have

$$\langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ k \mapsto W \parallel f(19 \mid k)\ \rangle \to \langle\ \mathsf{E}_0,\ x \mapsto 19,\ k_0 \mapsto W \parallel t\ \rangle$$

Note that by typing $k \;\notin \mathsf{FV}(f(19)) \;=\; \{\ f\ \}$, so we have

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ k \mapsto W \parallel f(19 \mid k)\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!] \parallel f(19) \parallel \mathcal{D}_K[\![\ W\ ]\!]\ \rangle\end{aligned}$$

We now distinguish cases according to the translation of t.

Case $\mathcal{D}[\![\ t\ ]\!] \;=\; s \rightsquigarrow \bullet$:
In this case we have

$$\begin{aligned}&\mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!]\\ &=\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \}\ \}\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto 19,\ k_0 \mapsto W \parallel t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19,\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel s\ \rangle\end{aligned}$$

By rules *(calli)* and *(sus)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ k \mapsto W \parallel f(19 \mid k)\ \rangle\ ]\!]\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19 \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \} \parallel \mathcal{D}_K[\![\ W\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19,\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel s\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t\ ]\!] \ = \ s \rightsquigarrow k_0$:
In this case we have

$$\begin{aligned}&\mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!]\\ &=\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow s\ \}\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto 19,\ k_0 \mapsto W \parallel t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19 \parallel s \parallel \mathcal{D}_K[\![\ W\ ]\!]\ \rangle\end{aligned}$$

By rule *(calli)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ k \mapsto W \parallel f(19 \mid k)\ \rangle\ ]\!]\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19 \parallel s \parallel \mathcal{D}_K[\![\ W\ ]\!]\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t\ ]\!] \ = \ s \rightsquigarrow v_0$ where $v_0 \ \neq \ k_0$:
In this case we have

$$\begin{aligned}&\mathcal{D}_V[\![\ \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \}\ ]\!]\\ &=\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s\ \}\ \}\ \}\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto 19,\ k_0 \mapsto W \parallel t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!].\mathsf{rm}(v_0),\ x \mapsto 19,\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle\end{aligned}$$

By rules *(calli)*, *(sus)* and *(run)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E}_0,\ (x \mid k_0) \Rightarrow t\ \},\ v \mapsto \mathsf{V},\ k \mapsto W \parallel f(v \mid k)\ \rangle\ ]\!]\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19 \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s\ \}\ \} \parallel \mathcal{D}_K[\![\ W\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19,\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel \textbf{run}(v_0)\ \{\ s\ \}\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19,\ k_0 \mapsto \mathcal{D}_K[\![\ W\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle\end{aligned}$$

Apart from the binding for v_0 in the environment this is just what we want and since v_0 is not free in s we can identify the two configurations by weakening.

case *(jmpv)*
We have

$$\langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \},\ v \mapsto \mathsf{V} \parallel k(v)\ \rangle \to \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel t\ \rangle$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \},\ v \mapsto \mathsf{V} \parallel k(v)\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel \textbf{ret}\ v \parallel \mathcal{D}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \}\ ]\!]\ \rangle\end{aligned}$$

We now distinguish cases according to the translation of t.

Case $\mathcal{D}[\![\ t\ ]\!] \ = \ s \rightsquigarrow \bullet$:
In this case we have

$$\begin{aligned}&\mathcal{D}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \}\ ]\!]\\ &=\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow s\ \}\ \}\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel s\ \rangle\end{aligned}$$

By rule *(retv-0)* we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \},\ v \mapsto \mathsf{V} \parallel k(v)\ \rangle\ ]\!]\\ \to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel s\ \rangle\end{array}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow x$:
In this case we have

$$\begin{array}{l}\mathcal{D}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \}\ ]\!]\\ \ =\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow \mathbf{run}(x)\ \{\ s\ \}\ \}\end{array}$$

and

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel t\ \rangle\ ]\!]\\ \ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{V}\ ]\!]\ \rangle\end{array}$$

By rules *(retv-0)* and *(run)* we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \},\ v \mapsto \mathsf{V} \parallel k(v)\ \rangle\ ]\!]\\ \to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_K[\![\ \mathsf{V}\ ]\!] \parallel \mathbf{run}(x)\ \{\ s\ \}\ \rangle\\ \to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_K[\![\ \mathsf{V}\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{V}\ ]\!]\ \rangle\end{array}$$

Apart from the binding for x in the environment this is just what we want and since x is not free in s we can identify the two configurations by weakening.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow k$ where $k\ \neq\ x$:
In this case we have

$$\begin{array}{l}\mathcal{D}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \}\ ]\!]\\ \ =\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!].\mathsf{rm}(k),\ (x) \Rightarrow s\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\end{array}$$

and

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto \mathsf{V} \parallel t\ \rangle\ ]\!]\\ \ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!].\mathsf{rm}(k),\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

By rule *(retv-1)* we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \},\ v \mapsto \mathsf{V} \parallel k(v)\ \rangle\ ]\!]\\ \to \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!].\mathsf{rm}(k),\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

which is exactly what we want.

case *(jmpi)*
We have

$$\langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \} \parallel k(19)\ \rangle \to \langle\ \mathsf{E}_0,\ x \mapsto 19 \parallel t\ \rangle$$

and

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \} \parallel k(19)\ \rangle\ ]\!]\\ \ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \mathbf{ret}\ 19 \parallel \mathcal{D}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \}\ ]\!]\ \rangle\end{array}$$

We now distinguish cases according to the translation of t.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow \bullet$:
In this case we have

$$\begin{aligned}&\mathcal{D}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \}\ ]\!]\\ &=\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ (x) \Rightarrow s\ \}\ \}\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto 19 \parallel t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto 19 \parallel s\ \rangle\end{aligned}$$

By rule *(reti-0)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \} \parallel k(19)\ \rangle\ ]\!]\\ &\rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!],\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!] \parallel s\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow x$:
This case is impossible since $x\ :\ \mathsf{Int}$ here.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow k$ where $k\ \neq\ x$:
In this case we have

$$\begin{aligned}&\mathcal{D}_K[\![\ \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \}\ ]\!]\\ &=\ \{\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!].\mathsf{rm}(k),\ (x) \Rightarrow s\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}_0,\ x \mapsto 19 \parallel t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!].\mathsf{rm}(k),\ x \mapsto 19 \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

By rule *(reti-1)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k \mapsto \{\ \mathsf{E}_0,\ (x) \Rightarrow t\ \} \parallel k(19)\ \rangle\ ]\!]\\ &\rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}_0\ ]\!].\mathsf{rm}(k),\ x \mapsto 19 \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

which is exactly what we want.

case *(let)*
We have

$$\langle\ \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rangle \rightarrow \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \} \parallel t\ \rangle$$

For the translations of the machine configurations we distinguish the cases of the translation of the left-hand term.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!]\ =\ s_0 \rightsquigarrow k_0$:
In this case we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{def}\ f(x)\ \{\ s_0\ \};\ s\ \rangle\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \} \parallel t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \}\ ]\!] \parallel s\ \rangle\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \} \parallel s\ \rangle\end{aligned}$$

By rule *(def-0)* we thus have

$$\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!] \rightarrow \mathcal{D}_M[\![\langle \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \} \parallel t \rangle]\!]$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$:
In this case we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!]\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s \rangle\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\langle \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \} \parallel t \rangle]\!]\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \}\ ]\!] \parallel s \rangle\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \} \parallel s \rangle\end{aligned}$$

By rule *(def-0)* we thus have

$$\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!] \rightarrow \mathcal{D}_M[\![\langle \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \} \parallel t \rangle]\!]$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow v_0$ where $v_0 \neq k_0$:
In this case we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!]\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{def}\ f(x)\ \{\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s \rangle\end{aligned}$$

and

$$\begin{aligned}&\mathcal{D}_M[\![\langle \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \} \parallel t \rangle]\!]\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \}\ ]\!] \parallel s \rangle\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ \parallel s \rangle\end{aligned}$$

By rule *(def-0)* we thus have

$$\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!] \rightarrow \mathcal{D}_M[\![\langle \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \} \parallel t \rangle]\!]$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow k_0$:
We have

$$\begin{aligned}&\mathcal{D}_M[\![\langle \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \} \parallel t \rangle]\!]\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \}\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \rangle\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0 \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \rangle\end{aligned}$$

We distinguish whether $k \in \mathsf{FV}(\textbf{def}\ f(x)\ \{\ s_0\ \};\ s)$ or not.
If so, we use $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k) = \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!] = \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]$ and obtain by rule *(run)* that

$$\begin{aligned}&\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!]\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{run}(k)\ \{\ \textbf{def}\ f(x)\ \{\ s_0\ \};\ s\ \} \rangle\\ &\rightarrow \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{def}\ f(x)\ \{\ s_0\ \};\ s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \rangle\end{aligned}$$

Using rule *(def-1)* hence gives us

$$\begin{aligned}&\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!]\\ &\rightarrow^2 \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \} \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \rangle\end{aligned}$$

Apart from the binding for k in the environment this is just what we want and since k is not free in s we can identify the two configurations by weakening.
If $k \notin \mathsf{FV}(\textbf{def}\ f(x)\ \{\ s_0\ \};\ s)$, we have

$$\begin{aligned}&\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!]\\ &= \langle \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel \textbf{def}\ f(x)\ \{\ s_0\ \};\ s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \rangle\end{aligned}$$

Using rule *(def-1)* hence gives us

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ (x) \Rightarrow s_0\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

Up to the missing mapping for k in the definition of f this is just what we want and since k is not free in s_0 we can identify the two definitions of f by weakening and hence also the configurations.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow k$ and $\mathcal{D}[\![\ t_0\ ]\!]\ =\ s_0 \rightsquigarrow \bullet$:
We have

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \}\ \|\ t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \}\ ]\!]\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

We distinguish whether $k\ \in \mathsf{FV}(\textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s)$ or not.
If so, we use $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)\ =\ \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!]\ =\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]$ and obtain by rule *(run)* that

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{run}(k)\ \{\ \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s\ \}\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

Using rule *(def-1)* hence gives us

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ &\to^2 \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

Apart from the binding for k in the environment this is just what we want and since k is not free in s we can identify the two configurations by weakening.
If $k\ \notin \mathsf{FV}(\textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s)$, we have

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k)\ \|\ \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

Using rule *(def-1)* hence gives us

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

Up to the missing mapping for k in the definition of f this is just what we want and since k is not free in s_0 we can identify the two definitions of f by weakening and hence also the configurations.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow k$ and $\mathcal{D}[\![\ t_0\ ]\!]\ =\ s_0 \rightsquigarrow v_0$ where $v_0\ \neq\ k_0$:
We have

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ f \mapsto \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \}\ \|\ t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ f \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x \mid k_0) \Rightarrow t_0\ \}\ ]\!]\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ f \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

We distinguish whether $k\ \in \mathsf{FV}(\textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s)$ or not.
If so, we use $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)\ =\ \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!]\ =\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]$ and obtain by rule *(run)* that

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{run}(k)\ \{\ \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s\ \}\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

Using rule *(def-1)* hence gives us

$$\begin{array}{l}\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\, f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!] \\ \to^2 \langle \mathcal{D}_E[\![\mathsf{E}]\!],\ f \mapsto \{\ \mathcal{D}_E[\![\mathsf{E}]\!],\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \} \parallel s \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle\end{array}$$

Apart from the binding for k in the environment this is just what we want and since k is not free in s we can identify the two configurations by weakening.
If $k \notin \mathsf{FV}(\textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s)$, we have

$$\begin{array}{l}\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\, f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!] \\ = \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k) \parallel \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle\end{array}$$

Using rule *(def-1)* hence gives us

$$\begin{array}{l}\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{let}\, f(x \mid k_0)\ \{\ t_0\ \};\ t \rangle]\!] \\ \to \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k),\ f \mapsto \{\ \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k),\ (x) \Rightarrow \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \} \parallel s \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle\end{array}$$

Up to the missing mapping for k in the definition of f this is just what we want and since k is not free in s_0 we can identify the two definitions of f by weakening and hence also the configurations.

case *(cnt)*
We have

$$\langle \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \rangle \to \langle \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \} \parallel t \rangle$$

For the translations of the machine configurations we distinguish the cases of the translation of the left-hand term.

Case $\mathcal{D}[\![t]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![t_0]\!] = s_0 \rightsquigarrow \bullet$:
In this case we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \rangle]\!] \\ = \langle \mathcal{D}_E[\![\mathsf{E}]\!] \parallel \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s \rangle\end{array}$$

and

$$\begin{array}{l}\mathcal{D}_M[\![\langle \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \} \parallel t \rangle]\!] \\ = \langle \mathcal{D}_E[\![\mathsf{E}]\!],\ k_0 \mapsto \mathcal{D}_V[\![\{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}]\!] \parallel s \rangle \\ = \langle \mathcal{D}_E[\![\mathsf{E}]\!],\ k_0 \mapsto \{\ \mathcal{D}_E[\![\mathsf{E}]\!],\ (x) \Rightarrow s_0\ \} \parallel s \rangle\end{array}$$

By rule *(proc-0)* we thus have

$$\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \rangle]\!] \to \mathcal{D}_M[\![\langle \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \} \parallel t \rangle]\!]$$

Case $\mathcal{D}[\![t]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![t_0]\!] = s_0 \rightsquigarrow x$:
In this case we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \rangle]\!] \\ = \langle \mathcal{D}_E[\![\mathsf{E}]\!] \parallel \textbf{process}\ k_0(x)\ \{\ \textbf{run}(x)\ \{\ s_0\ \}\ \};\ s \rangle\end{array}$$

and

$$\begin{array}{l}\mathcal{D}_M[\![\langle \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \} \parallel t \rangle]\!] \\ = \langle \mathcal{D}_E[\![\mathsf{E}]\!],\ k_0 \mapsto \mathcal{D}_V[\![\{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}]\!] \parallel s \rangle \\ = \langle \mathcal{D}_E[\![\mathsf{E}]\!],\ k_0 \mapsto \{\ \mathcal{D}_E[\![\mathsf{E}]\!],\ (x) \Rightarrow \textbf{run}(x)\ \{\ s_0\ \}\ \} \parallel s \rangle\end{array}$$

By rule *(proc-0)* we thus have

$$\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \, \rangle \,]\!] \to \mathcal{D}_M[\![\, \langle \, \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \} \parallel t \, \rangle \,]\!]$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow v_0$ where $v_0 \neq x$:
We have

$$\begin{aligned}&\mathcal{D}_M[\![\, \langle \, \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \} \parallel t \, \rangle \,]\!]\\ &= \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}\ ]\!] \parallel s \, \rangle\\ &= \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!] \parallel s \, \rangle\end{aligned}$$

We distinguish whether $v_0 \in \mathsf{FV}(\textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0)$ or not.
If so, we use $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!](v_0) = \mathcal{D}_V[\![\ \mathsf{E}(v_0)\ ]\!] = \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]$ and obtain by rule *(run)* that

$$\begin{aligned}&\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \, \rangle \,]\!]\\ &= \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{run}(v_0)\ \{\ \textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0\ \} \, \rangle\\ &\to \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0 \parallel \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!] \, \rangle\end{aligned}$$

Using rules *(push)* and *(sus)* hence gives us

$$\begin{aligned}&\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \, \rangle \,]\!]\\ &\to^2 \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \} \parallel \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!] \, \rangle\\ &\to \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!] \parallel s \, \rangle\end{aligned}$$

Up to the additional mapping for v_0 in the definition of k_0 this is just what we want and since v_0 is not free in s_0 we can identify the two definitions of k_0 by weakening and hence also the configurations.
If $v_0 \notin \mathsf{FV}(\textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0)$, we have

$$\begin{aligned}&\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \, \rangle \,]\!]\\ &= \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0) \parallel \textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0 \parallel \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!] \, \rangle\end{aligned}$$

Using rules *(push)* and *(sus)* hence gives us

$$\begin{aligned}&\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \, \rangle \,]\!]\\ &\to \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0) \parallel \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \} \parallel \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!] \, \rangle\\ &\to \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!] \parallel s \, \rangle\end{aligned}$$

Apart from the missing binding for v_0 in the environment this is just what we want and since v_0 is not free in s we can identify the two configurations by weakening.

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k$ with $k \neq k_0$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$:
In this case we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\, \langle \, \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \} \parallel t \, \rangle \,]\!]\\ &= \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k_0 \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \, \rangle\\ &= \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \} \parallel s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \, \rangle\end{aligned}$$

We distinguish whether $k \in \mathsf{FV}(\textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s)$ or not.
If so, we use $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k) = \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!] = \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]$ and obtain by rule *(run)* that

$$\begin{aligned}&\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t \, \rangle \,]\!]\\ &= \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{run}(k)\ \{\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ \} \, \rangle\\ &\to \langle \, \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \, \rangle\end{aligned}$$

By rule *(proc-1)* we thus have

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ \to^2 \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

Apart from the binding for k in the environment this is just what we want and since k is not free in s we can identify the two configurations by weakening.
If $k \notin \mathsf{FV}(\textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s)$, we have

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k)\ \|\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

By rule *(proc-1)* we thus have

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ \to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ (x) \Rightarrow s_0\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

Up to the missing mapping for k in the definition of k_0 this is just what we want and since k is not free in s_0 we can identify the two definitions of k_0 by weakening and hence also the configurations.

Case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \rightsquigarrow k$ with $k \neq k_0$ and $\mathcal{D}[\![\ t_0\ ]\!]\ =\ s_0 \rightsquigarrow x$:
In this case we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}\ \|\ t\ \rangle\ ]\!]\\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k_0 \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}\ ]\!]\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \textbf{run}(x)\ \{\ s_0\ \}\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

We distinguish whether $k \in \mathsf{FV}(\textbf{process}\ k_0(x)\ \{\ \textbf{run}(x)\ \{\ s_0\ \}\ \};\ s)$ or not.
If so, we use $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)\ =\ \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!]\ =\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]$ and obtain by rule *(run)* that

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{run}(k)\ \{\ \textbf{process}\ k_0(x)\ \{\ \textbf{run}(x)\ \{\ s_0\ \}\ \};\ s\ \}\ \rangle\\ \to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{process}\ k_0(x)\ \{\ \textbf{run}(x)\ \{\ s_0\ \}\ \};\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

By rule *(proc-1)* we thus have

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ \to^2 \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow \textbf{run}(x)\ \{\ s_0\ \}\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

Apart from the binding for k in the environment this is just what we want and since k is not free in s we can identify the two configurations by weakening.
If $k \notin \mathsf{FV}(\textbf{process}\ k_0(x)\ \{\ \textbf{run}(x)\ \{\ s_0\ \}\ \};\ s)$, we have

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k)\ \|\ \textbf{process}\ k_0(x)\ \{\ \textbf{run}(x)\ \{\ s_0\ \}\ \};\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

By rule *(proc-1)* we thus have

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ \to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ (x) \Rightarrow \textbf{run}(x)\ \{\ s_0\ \}\ \}\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

Up to the missing mapping for k in the definition of k_0 this is just what we want and since k is not free in s_0 we can identify the two definitions of k_0 by weakening and hence also the configurations.

Case $\mathcal{D}[\![\ t\ ]\!] \ =\ s \rightsquigarrow k$ with $k\ \neq\ k_0$ and $\mathcal{D}[\![\ t_0\ ]\!] \ =\ s_0 \rightsquigarrow v_0$ where $v_0\ \neq\ x$:
We have

$$\begin{aligned}
&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}\ \|\ t\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k_0 \mapsto \mathcal{D}_V[\![\ \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}\ ]\!]\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\
&=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle
\end{aligned}$$

We distinguish whether $v_0\ \in \mathsf{FV}(\textbf{val}\ x\ =\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(k)\ \{\ s\ \}\ \};\ s_0)$ or not.
If so, we use $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!](v_0)\ =\ \mathcal{D}_V[\![\ \mathsf{E}(v_0)\ ]\!]\ =\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]$ and obtain by rule *(run)* that

$$\begin{aligned}
&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{run}(v_0)\ \{\ \textbf{val}\ x\ =\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(k)\ \{\ s\ \}\ \};\ s_0\ \}\ \rangle\\
&\rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{val}\ x\ =\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(k)\ \{\ s\ \}\ \};\ s_0\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle
\end{aligned}$$

Using rules *(push)*, *(sus)* and *(run)* hence gives us

$$\begin{aligned}
&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\
&\rightarrow^2 \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(k)\ \{\ s\ \}\ \}\ \|\ \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle\\
&\rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \|\ \textbf{run}(k)\ \{\ s\ \}\ \rangle\\
&\rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle
\end{aligned}$$

Up to the additional mapping for v_0 in the definition of k_0 and the additional binding for k in the environment this is just what we want and since v_0 is not free in s_0 and k is not free in s we can identify the two definitions of k_0 by weakening and hence also the configurations.
If $v_0\ \notin \mathsf{FV}(\textbf{val}\ x\ =\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(k)\ \{\ s\ \}\ \};\ s_0)$, we have

$$\begin{aligned}
&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0)\ \|\ \textbf{val}\ x\ =\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(k)\ \{\ s\ \}\ \};\ s_0\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle
\end{aligned}$$

Using rules *(push)*, *(sus)* and *(run)* hence gives us

$$\begin{aligned}
&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\
&\rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0)\ \|\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(k)\ \{\ s\ \}\ \}\ \|\ \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle\\
&\rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \|\ \textbf{run}(k)\ \{\ s\ \}\ \rangle\\
&\rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ k_0 \mapsto \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \|\ s\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle
\end{aligned}$$

Apart from the missing binding for v_0 and the additional binding for k in the environment this is just what we want and since v_0 and k are not free in s we can identify the two configurations by weakening.

Case $\mathcal{D}[\![\ t\ ]\!] \ =\ s \rightsquigarrow k_0$ and $\mathcal{D}[\![\ t_0\ ]\!] \ =\ s_0 \rightsquigarrow \bullet$:
In this case we obtain

$$\begin{aligned}
&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ \textbf{run}(k_0)\ \{\ s\ \}\ \rangle
\end{aligned}$$

and

$$\begin{aligned}
&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ k_0 \mapsto \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}\ \|\ t\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ s\ \|\ \mathcal{D}_K[\![\ \{\ \mathsf{E},\ (x) \Rightarrow t_0\ \}\ ]\!]\ \rangle\\
&=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ s\ \|\ \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x) \Rightarrow s_0\}\ \rangle
\end{aligned}$$

By rules *(proc-0)* and *(run)* we have

$$\begin{array}{l}\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt } k_0(x) \; \{ \; t_0 \; \}; \; t \, \rangle \,]\!] \\ \to \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; k_0 \mapsto \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; (x) \Rightarrow s_0 \; \} \parallel \textbf{run}(k_0) \; \{ \; s \; \} \, \rangle \\ \to \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; k_0 \mapsto \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; (x) \Rightarrow s_0 \; \} \parallel s \parallel \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; (x) \Rightarrow s_0 \; \} \, \rangle\end{array}$$

Up to the additional binding for k_0 in the environment this is just what we want and since k_0 is not free in s we can identify the two configurations by weakening.

Case $\mathcal{D}[\![\; t \;]\!] \; = \; s \rightsquigarrow k_0$ and $\mathcal{D}[\![\; t_0 \;]\!] \; = \; s_0 \rightsquigarrow x$:
In this case we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt } k_0(x) \; \{ \; t_0 \; \}; \; t \, \rangle \,]\!] \\ = \; \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel \textbf{process } k_0(x) \; \{ \; \textbf{run}(x) \; \{ \; s_0 \; \} \; \}; \; \textbf{run}(k_0) \; \{ \; s \; \} \, \rangle\end{array}$$

and

$$\begin{array}{l}\mathcal{D}_M[\![\, \langle \, \mathsf{E}, \; k_0 \mapsto \{ \; \mathsf{E}, \; (x) \Rightarrow t_0 \; \} \parallel t \, \rangle \,]\!] \\ = \; \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel s \parallel \mathcal{D}_K[\![\; \{ \; \mathsf{E}, \; (x) \Rightarrow t_0 \; \} \;]\!] \, \rangle \\ = \; \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel s \parallel \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; (x) \Rightarrow \textbf{run}(x) \; \{ \; s_0 \; \} \; \} \, \rangle\end{array}$$

By rules *(proc-0)* and *(run)* we have

$$\begin{array}{l}\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt } k_0(x) \; \{ \; t_0 \; \}; \; t \, \rangle \,]\!] \\ \to \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; k_0 \mapsto \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; (x) \Rightarrow \textbf{run}(x) \; \{ \; s_0 \; \} \; \} \parallel \textbf{run}(k_0) \; \{ \; s \; \} \, \rangle \\ \to \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; k_0 \mapsto \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; (x) \Rightarrow \textbf{run}(k) \; \{ \; s_0 \; \} \; \} \parallel s \parallel \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; (x) \Rightarrow \textbf{run}(x) \; \{ \; s_0 \; \} \; \} \, \rangle\end{array}$$

Up to the additional binding for k_0 in the environment this is just what we want and since k_0 is not free in s we can identify the two configurations by weakening.

Case $\mathcal{D}[\![\; t \;]\!] \; = \; s \rightsquigarrow k_0$ and $\mathcal{D}[\![\; t_0 \;]\!] \; = \; s_0 \rightsquigarrow v_0$ with $v_0 \; \neq \; x$:
We have

$$\begin{array}{l}\mathcal{D}_M[\![\, \langle \, \mathsf{E}, \; k_0 \mapsto \{ \; \mathsf{E}, \; (x) \Rightarrow t_0 \; \} \parallel t \, \rangle \,]\!] \\ = \; \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel s \parallel \mathcal{D}_K[\![\; \{ \; \mathsf{E}, \; (x) \Rightarrow t_0 \; \} \;]\!] \, \rangle \\ = \; \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel s \parallel \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!].\mathsf{rm}(v_0), \; (x) \Rightarrow s_0 \; \} \; :: \; \mathcal{D}_K[\![\; \mathsf{E}(v_0) \;]\!] \, \rangle\end{array}$$

We distinguish whether $v_0 \; \in \mathsf{FV}(\textbf{val } x \; = \; s; \; s_0)$ or not.
If so, we use $\mathcal{D}_E[\![\; \mathsf{E} \;]\!](v_0) \; = \; \mathcal{D}_V[\![\; \mathsf{E}(v_0) \;]\!] \; = \; \mathcal{D}_K[\![\; \mathsf{E}(v_0) \;]\!]$ and obtain by rule *(run)* that

$$\begin{array}{l}\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt } k_0(x) \; \{ \; t_0 \; \}; \; t \, \rangle \,]\!] \\ = \; \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel \textbf{run}(v_0) \; \{ \; \textbf{val } x \; = \; s; \; s_0 \; \} \, \rangle \\ \to \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel \textbf{val } x \; = \; s; \; s_0 \parallel \mathcal{D}_K[\![\; \mathsf{E}(v_0) \;]\!] \, \rangle\end{array}$$

By rule *(push)* we thus have

$$\begin{array}{l}\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt } k_0(x) \; \{ \; t_0 \; \}; \; t \, \rangle \,]\!] \\ \to^2 \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!] \parallel s \parallel \{ \; \mathcal{D}_E[\![\, \mathsf{E} \,]\!], \; (x) \Rightarrow s_0 \; \} \; :: \; \mathcal{D}_K[\![\; \mathsf{E}(v_0) \;]\!] \, \rangle\end{array}$$

Up to the additional binding for v_0 in the definition of the first stack frame this is just what we want and since v_0 is not free in s_0 we can identify the two configurations by weakening.
If $v_0 \; \notin \mathsf{FV}(\textbf{val } x \; = \; s; \; s_0)$, we have

$$\begin{array}{l}\mathcal{D}_M[\![\, \langle \, \mathsf{E} \parallel \textbf{cnt } k_0(x) \; \{ \; t_0 \; \}; \; t \, \rangle \,]\!] \\ = \; \langle \, \mathcal{D}_E[\![\, \mathsf{E} \,]\!].\mathsf{rm}(v_0) \parallel \textbf{val } x \; = \; s; \; s_0 \parallel \mathcal{D}_K[\![\; \mathsf{E}(v_0) \;]\!] \, \rangle\end{array}$$

By rule *(push)* we thus have

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ \rangle\ ]\!]\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0)\ \|\ s\ \|\ \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(v_0),\ (x) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(v_0)\ ]\!]\ \rangle\end{aligned}$$

Up to the missing binding for v_0 in the environment this is just what we want and since v_0 is not free in s we can identify the two configurations by weakening.

case *(iftruv)*
We have

$$\langle\ \mathsf{E},\ v \mapsto \mathsf{true}\ \|\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle \to \langle\ \mathsf{E},\ v \mapsto \mathsf{true}\ \|\ t_1\ \rangle$$

We distinguish cases according to the translation of t_1 and t_2.

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \rightsquigarrow k$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \rightsquigarrow k$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ =\ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true}\ \|\ t_1\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true}\ \|\ s_1\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle$$

By rule *(iftruv-1)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true}\ \|\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true}\ \|\ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true}\ \|\ s_1\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \rightsquigarrow \bullet$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ =\ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \bullet$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true}\ \|\ t_1\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{true}\ \|\ s_1\ \rangle$$

By rule *(iftruv-0)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true}\ \|\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{true}\ \|\ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{true}\ \|\ s_1\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \rightsquigarrow k$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ =\ \textbf{if}\ v\ \textbf{then}\ \textbf{suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2 \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true}\ \|\ t_1\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{true}\ \|\ s_1\ \rangle$$

By rules *(sus)* and *(iftruv-1)* we obtain using $\mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] = \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!] = \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)$ that

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if } v \textbf{ then } t_1 \textbf{ else } t_2\ \rangle\ ]\!]\\ &= \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true} \parallel \textbf{if } v \textbf{ then suspend } \{\ k \Rightarrow s_1\ \} \textbf{ else } s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true} \parallel \textbf{suspend } \{\ k \Rightarrow s_1\ \} \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true},\ k \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k) \parallel s_1\ \rangle\end{aligned}$$

which is exactly what we want since $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k) = \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]$.

Case $\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow k$ and $\mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow \bullet$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if } v \textbf{ then } t_1 \textbf{ else } t_2\ ]\!] = \textbf{if } v \textbf{ then } s_1 \textbf{ else suspend } \{\ k \Rightarrow s_2\ \} \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel t_1\ \rangle\ ]\!] = \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true} \parallel s_1 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle$$

By rule *(iftruv-1)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if } v \textbf{ then } t_1 \textbf{ else } t_2\ \rangle\ ]\!]\\ &= \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true} \parallel \textbf{if } v \textbf{ then } s_1 \textbf{ else suspend } \{\ k \Rightarrow s_2\ \} \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{true} \parallel s_1 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow k_1$ and $\mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow k_2$ and $k_1 \neq k_2$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if } v \textbf{ then } t_1 \textbf{ else } t_2\ ]\!] = \textbf{if } v \textbf{ then suspend } \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \} \textbf{ else } s_2 \rightsquigarrow k_2$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel t_1\ \rangle\ ]\!] = \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_1),\ v \mapsto \mathsf{true} \parallel s_1 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k_1)\ ]\!]\ \rangle$$

By rules *(iftruv-1)*, *(sus)* and *(run)* we obtain using $\mathcal{D}_K[\![\ \mathsf{E}(k_i)\ ]\!] = \mathcal{D}_V[\![\ \mathsf{E}(k_i)\ ]\!] = \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k_i)$ for $i = 1,\ 2$ that

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{true} \parallel \textbf{if } v \textbf{ then } t_1 \textbf{ else } t_2\ \rangle\ ]\!]\\ &= \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2),\ v \mapsto \mathsf{true} \parallel \textbf{if } v \textbf{ then suspend } \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \} \textbf{ else } s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k_2)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2),\ v \mapsto \mathsf{true} \parallel \textbf{suspend } \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \} \parallel \mathcal{D}_K[\![\ \mathsf{E}(k_2)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2),\ v \mapsto \mathsf{true},\ k_2 \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k_2) \parallel \textbf{run}(k_1)\ \{\ s_1\ \}\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{true} \parallel s_1 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k_1)\ ]\!]\ \rangle\end{aligned}$$

since $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2),\ k_2 \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k_2) = \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]$. Up to the additional binding for k_1 in the environment this is just what we want and since k_1 is not free in s_1 we can identify the two configurations by weakening.

case *(iftrub)*
We have

$$\langle\ \mathsf{E} \parallel \textbf{if } \mathsf{true} \textbf{ then } t_1 \textbf{ else } t_2\ \rangle \to \langle\ \mathsf{E} \parallel t_1\ \rangle$$

We distinguish cases according to the translation of t_1 and t_2.

Case $\mathcal{D}[\![\ t_1\ ]\!] \ =\ s_1 \rightsquigarrow k$ and $\mathcal{D}[\![\ t_2\ ]\!] \ =\ s_2 \rightsquigarrow k$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] \ =\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel t_1\ \rangle\ ]\!] \ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel s_1 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle$$

By rule *(iftrub-1)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel s_1 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!] \ =\ s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!] \ =\ s_2 \rightsquigarrow \bullet$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] \ =\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \bullet$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel t_1\ \rangle\ ]\!] \ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel s_1\ \rangle$$

By rule *(iftrub-0)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel s_1\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!] \ =\ s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!] \ =\ s_2 \rightsquigarrow k$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] \ =\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ \textbf{suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2 \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel t_1\ \rangle\ ]\!] \ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel s_1\ \rangle$$

By rules *(sus)* and *(iftrub-1)* we obtain using $\mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \ =\ \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!] \ =\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)$ that

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel \textbf{if}\ \mathsf{true}\ \textbf{then}\ \textbf{suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel \textbf{suspend}\ \{\ k \Rightarrow s_1\ \} \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k) \parallel s_1\ \rangle\end{aligned}$$

which is exactly what we want since $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k) \ =\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]$.

Case $\mathcal{D}[\![\ t_1\ ]\!] \ =\ s_1 \rightsquigarrow k$ and $\mathcal{D}[\![\ t_2\ ]\!] \ =\ s_2 \rightsquigarrow \bullet$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] \ =\ \textbf{if}\ \mathsf{true}\ \textbf{then}\ s_1\ \textbf{else}\ \textbf{suspend}\ \{\ k \Rightarrow s_2\ \} \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\langle \mathsf{E} \parallel t_1 \rangle]\!] \;=\; \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k) \parallel s_1 \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle$$

By rule *(iftrub-1)* we obtain

$$\begin{array}{l} \mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{if } \mathsf{true} \textbf{ then } t_1 \textbf{ else } t_2 \rangle]\!] \\ \;=\; \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k) \parallel \textbf{if } \mathsf{true} \textbf{ then } s_1 \textbf{ else suspend } \{ \, k \Rightarrow s_2 \, \} \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle \\ \to \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k) \parallel s_1 \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle \end{array}$$

which is exactly what we want.

Case $\mathcal{D}[\![t_1]\!] \;=\; s_1 \rightsquigarrow k_1$ and $\mathcal{D}[\![t_2]\!] \;=\; s_2 \rightsquigarrow k_2$ and $k_1 \;\neq\; k_2$:
In this case we have

$$\mathcal{D}[\![\textbf{if } \mathsf{true} \textbf{ then } t_1 \textbf{ else } t_2]\!] \;=\; \textbf{if } \mathsf{true} \textbf{ then suspend } \{ \, k_2 \Rightarrow \textbf{run}(k_1) \; \{ \, s_1 \, \} \, \} \textbf{ else } s_2 \rightsquigarrow k_2$$

and

$$\mathcal{D}_M[\![\langle \mathsf{E} \parallel t_1 \rangle]\!] \;=\; \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k_1) \parallel s_1 \parallel \mathcal{D}_K[\![\mathsf{E}(k_1)]\!] \rangle$$

By rules *(iftrub-1)*, *(sus)* and *(run)* we obtain using $\mathcal{D}_K[\![\mathsf{E}(k_i)]\!] \;=\; \mathcal{D}_V[\![\mathsf{E}(k_i)]\!] \;=\; \mathcal{D}_E[\![\mathsf{E}]\!](k_i)$ for $i \;=\; 1,\; 2$ that

$$\begin{array}{l} \mathcal{D}_M[\![\langle \mathsf{E} \parallel \textbf{if } \mathsf{true} \textbf{ then } t_1 \textbf{ else } t_2 \rangle]\!] \\ \;=\; \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k_2) \parallel \textbf{if } \mathsf{true} \textbf{ then suspend } \{ \, k_2 \Rightarrow \textbf{run}(k_1) \; \{ \, s_1 \, \} \, \} \textbf{ else } s_2 \parallel \mathcal{D}_K[\![\mathsf{E}(k_2)]\!] \rangle \\ \to \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k_2) \parallel \textbf{suspend } \{ \, k_2 \Rightarrow \textbf{run}(k_1) \; \{ \, s_1 \, \} \, \} \parallel \mathcal{D}_K[\![\mathsf{E}(k_2)]\!] \rangle \\ \to \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k_2), \; k_2 \mapsto \mathcal{D}_E[\![\mathsf{E}]\!](k_2) \parallel \textbf{run}(k_1) \; \{ \, s_1 \, \} \rangle \\ \to \langle \mathcal{D}_E[\![\mathsf{E}]\!] \parallel s_1 \parallel \mathcal{D}_K[\![\mathsf{E}(k_1)]\!] \rangle \end{array}$$

since $\mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k_2), \; k_2 \mapsto \mathcal{D}_E[\![\mathsf{E}]\!](k_2) \;=\; \mathcal{D}_E[\![\mathsf{E}]\!]$. Up to the additional binding for k_1 in the environment this is just what we want and since k_1 is not free in s_1 we can identify the two configurations by weakening.

case *(ifflsv)*
We have

$$\langle \mathsf{E}, \; v \mapsto \mathsf{false} \parallel \textbf{if } v \textbf{ then } t_1 \textbf{ else } t_2 \rangle \to \langle \mathsf{E}, \; v \mapsto \mathsf{false} \parallel t_2 \rangle$$

We distinguish cases according to the translation of t_1 and t_2.

Case $\mathcal{D}[\![t_1]\!] \;=\; s_1 \rightsquigarrow k$ and $\mathcal{D}[\![t_2]\!] \;=\; s_2 \rightsquigarrow k$:
In this case we have

$$\mathcal{D}[\![\textbf{if } v \textbf{ then } t_1 \textbf{ else } t_2]\!] \;=\; \textbf{if } v \textbf{ then } s_1 \textbf{ else } s_2 \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\langle \mathsf{E}, \; v \mapsto \mathsf{false} \parallel t_2 \rangle]\!] \;=\; \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k), \; v \mapsto \mathsf{false} \parallel s_2 \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle$$

By rule *(ifflsv-1)* we obtain

$$\begin{array}{l} \mathcal{D}_M[\![\langle \mathsf{E}, \; v \mapsto \mathsf{false} \parallel \textbf{if } v \textbf{ then } t_1 \textbf{ else } t_2 \rangle]\!] \\ \;=\; \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k), \; v \mapsto \mathsf{false} \parallel \textbf{if } v \textbf{ then } s_1 \textbf{ else } s_2 \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle \\ \to \langle \mathcal{D}_E[\![\mathsf{E}]\!].\mathsf{rm}(k), \; v \mapsto \mathsf{false} \parallel s_2 \parallel \mathcal{D}_K[\![\mathsf{E}(k)]\!] \rangle \end{array}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!] \ = \ s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!] \ = \ s_2 \rightsquigarrow \bullet$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] \ = \ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \bullet$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel t_2\ \rangle\ ]\!] \ = \ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{false} \parallel s_2\ \rangle$$

By rule *(ifflsv-0)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &= \ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{false} \parallel s_2\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!] \ = \ s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!] \ = \ s_2 \rightsquigarrow k$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] \ = \ \textbf{if}\ v\ \textbf{then}\ \textbf{suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2 \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel t_2\ \rangle\ ]\!] \ = \ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{false} \parallel s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle$$

By rule *(ifflsv-1)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &= \ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ \textbf{suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{false} \parallel s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!] \ = \ s_1 \rightsquigarrow k$ and $\mathcal{D}[\![\ t_2\ ]\!] \ = \ s_2 \rightsquigarrow \bullet$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] \ = \ \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ \textbf{suspend}\ \{\ k \Rightarrow s_2\ \} \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel t_2\ \rangle\ ]\!] \ = \ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ v \mapsto \mathsf{false} \parallel s_2\ \rangle$$

By rules *(sus)* and *(ifflsv-1)* we obtain using $\mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!] \ = \ \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!] \ = \ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)$ that

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &= \ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ \textbf{suspend}\ \{\ k \Rightarrow s_2\ \} \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{false} \parallel \textbf{suspend}\ \{\ k \Rightarrow s_2\ \} \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ v \mapsto \mathsf{false},\ k \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k) \parallel s_2\ \rangle\end{aligned}$$

which is exactly what we want since $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k) \ = \ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]$.

Case $\mathcal{D}[\![\ t_1\ ]\!] \ = \ s_1 \rightsquigarrow k_1$ and $\mathcal{D}[\![\ t_2\ ]\!] \ = \ s_2 \rightsquigarrow k_2$ and $k_1 \ \neq \ k_2$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!] \ = \ \textbf{if}\ v\ \textbf{then}\ \textbf{suspend}\ \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ \textbf{else}\ s_2 \rightsquigarrow k_2$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel t_2\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2),\ v \mapsto \mathsf{false} \parallel s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k_2)\ ]\!]\ \rangle$$

By rule *(ifflsv-1)* we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E},\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!] \\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2),\ v \mapsto \mathsf{false} \parallel \textbf{if}\ v\ \textbf{then}\ \textbf{suspend}\ \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ \textbf{else}\ s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k_2)\ ]\!]\ \rangle \\ \rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2),\ v \mapsto \mathsf{false} \parallel s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k_2)\ ]\!]\ \rangle\end{array}$$

which is exactly what we want.

case *(ifflsb)*
We have

$$\langle\ \mathsf{E} \parallel \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle \rightarrow \langle\ \mathsf{E} \parallel t_2\ \rangle$$

We distinguish cases according to the translation of t_1 and t_2.

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \rightsquigarrow k$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \rightsquigarrow k$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ =\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel t_2\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle$$

By rule *(ifflsb-1)* we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!] \\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle \\ \rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k) \parallel s_2 \parallel \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{array}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \rightsquigarrow \bullet$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ =\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \bullet$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel t_2\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel s_2\ \rangle$$

By rule *(ifflsb-0)* we obtain

$$\begin{array}{l}\mathcal{D}_M[\![\ \langle\ \mathsf{E} \parallel \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!] \\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \rangle \\ \rightarrow \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!] \parallel s_2\ \rangle\end{array}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \rightsquigarrow k$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ =\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ \textbf{suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2 \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ t_2\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k)\ \|\ s_2\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle$$

By rule *(ifflsb-1)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k)\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k)\ \|\ s_2\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\end{aligned}$$

which is exactly what we want.

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \rightsquigarrow k$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \rightsquigarrow \bullet$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ =\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else suspend}\ \{\ k \Rightarrow s_2\ \} \rightsquigarrow k$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ t_2\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ s_2\ \rangle$$

By rules *(sus)* and *(ifflsb-1)* we obtain using $\mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ =\ \mathcal{D}_V[\![\ \mathsf{E}(k)\ ]\!]\ =\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)$ that

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k)\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ s_1\ \textbf{else suspend}\ \{\ k \Rightarrow s_2\ \}\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k)\ \|\ \textbf{suspend}\ \{\ k \Rightarrow s_2\ \}\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)\ \|\ s_2\ \rangle\end{aligned}$$

which is exactly what we want since $\mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ k \mapsto \mathcal{D}_E[\![\ \mathsf{E}\ ]\!](k)\ =\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]$.

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \rightsquigarrow k_1$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \rightsquigarrow k_2$ and $k_1\ \neq\ k_2$:
In this case we have

$$\mathcal{D}[\![\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ =\ \textbf{if}\ \mathsf{false}\ \textbf{then suspend}\ \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ \textbf{else}\ s_2 \rightsquigarrow k_2$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ t_2\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2)\ \|\ s_2\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k_2)\ ]\!]\ \rangle$$

By rule *(ifflsb-1)* we obtain

$$\begin{aligned}&\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \rangle\ ]\!]\\ &=\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2)\ \|\ \textbf{if}\ \mathsf{false}\ \textbf{then suspend}\ \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ \textbf{else}\ s_2\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k_2)\ ]\!]\ \rangle\\ &\to \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k_2)\ \|\ s_2\ \|\ \mathcal{D}_K[\![\ \mathsf{E}(k_2)\ ]\!]\ \rangle\end{aligned}$$

which is exactly what we want.

◀

We again obtain the corollary of evaluation as for the CPS translation above. The DS-machine needs at most four times as many steps as the CPS-machine.

Proof. Note that $\mathcal{D}[\![\ \textbf{exit}\ e\ ]\!]\ =\ \textbf{exit}\ e \rightsquigarrow \bullet$ and thus

$$\mathcal{D}_K[\![\ \texttt{done}\ ]\!]\ =\ \{\ \mathcal{D}_E[\![\ \bullet\ ]\!],\ (x\ :\ \tau) \Rightarrow \textbf{exit}\ x\ \}\ =\ \texttt{done}$$

Hence, we have

$$\mathcal{D}_M[\![\ \langle\ k \mapsto \texttt{done}\ \|\ t\ \rangle\ ]\!]\ =\ \langle\ \bullet\ \|\ s\ \|\ \texttt{done}\ \rangle$$

and

$$\mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ \textbf{exit}\ e\ \rangle\ ]\!]\ =\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ \textbf{exit}\ e\ \rangle$$

The claim now follows by Corollary 13. ◀

D.2 Right Inverse

We now prove that $\mathcal{D}[\![\ \cdot\ ]\!]$ is the right inverse of $\mathcal{C}[\![\ \cdot\ ]\!]$ (Theorem 15).

Proof. Induction over the typing derivation and case distinctions according to the definition of $\mathcal{D}[\![\ \cdot\ ]\!]$. We use in various cases that $\mathcal{D}[\![\ \cdot\ ]\!]$ does not introduce fresh variables.

case App

Given $\Gamma \vdash f(e \mid k)$ we distinguish whether $k \in \mathsf{FV}(f(e))$ or not.
If so, we have $\mathcal{D}[\![\ f(e \mid k)\ ]\!] = \textbf{run}(k)\ \{\ f(e)\ \} \rightsquigarrow \bullet$. Since $\mathcal{C}[\![\ \textbf{run}(k)\ \{\ s\ \}\ ]\!]_\bullet = \mathcal{C}[\![\ s\ ]\!]_k$ we obtain $\mathcal{C}[\![\ \mathcal{D}[\![\ f(e \mid k)\ ]\!]\ ]\!]_\bullet = \mathcal{C}[\![\ f(e)\ ]\!]_k = f(e \mid k)$.
If not, we have $\mathcal{D}[\![\ f(e \mid k)\ ]\!] = f(e) \rightsquigarrow k$ and obtain $\mathcal{C}[\![\ \mathcal{D}[\![\ f(e \mid k)\ ]\!]\ ]\!]_k = \mathcal{C}[\![\ f(e)\ ]\!]_k = f(e \mid k)$.

case Jmp

Given $\Gamma \vdash k(e)$ we have $\mathcal{D}[\![\ k(e)\ ]\!] = \textbf{ret}\ e \rightsquigarrow k$ and obtain $\mathcal{C}[\![\ \mathcal{D}[\![\ k(e)\ ]\!]\ ]\!]_k = \mathcal{C}[\![\ \textbf{ret}\ e\ ]\!]_k = k(e)$.

case Let

Given $\Gamma \vdash \textbf{let}\ f(x : \tau \mid k_0 : \neg\,\tau_0)\ \{\ t_0\ \};\ t$ we have $\Gamma,\ x : \tau,\ k_0 : \neg\,\tau_0 \vdash t_0$ and $\Gamma,\ f : \tau \to \tau_0 \vdash t$.
We distinguish whether $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ or $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k$. In each case we need to distinguish three subcases, $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow k_0$, $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow v_0$ with $v_0 \neq k_0$.

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow k_0$ (note that this means that $k_0 \notin \mathsf{FV}(s_0)$):
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_\bullet = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_{k_0} = t_0$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \textbf{def}\ f(x)\ \{\ s_0\ \};\ s\ ]\!]_\bullet \\
&= \textbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \};\ \mathcal{C}[\![\ s\ ]\!]_\bullet && (k_0 \text{ is fresh}) \\
&= \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_\bullet = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_\bullet = t_0$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s\ ]\!]_\bullet \\
&= \textbf{let}\ f(x \mid k_1)\ \{\ \mathcal{C}[\![\ \textbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ ]\!]_{k_1}\ \};\ \mathcal{C}[\![\ s\ ]\!]_\bullet && (k_1 \text{ is fresh}) \\
&= \textbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \};\ t && (\alpha\text{-renaming}) \\
&= \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow v_0$ where $v_0 \neq k_0$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_\bullet = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_{v_0} = t_0$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \textbf{def}\ f(x)\ \{\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s\ ]\!]_\bullet \\
&= \textbf{let}\ f(x \mid k_1)\ \{\ \mathcal{C}[\![\ \textbf{suspend}\ \{\ k_0 \Rightarrow \textbf{run}(v_0)\ \{\ s_0\ \}\ \}\ ]\!]_{k_1}\ \};\ \mathcal{C}[\![\ s\ ]\!]_\bullet && (k_1 \text{ is fresh}) \\
&= \textbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ \textbf{run}(v_0)\ \{\ s_0\ \}\ ]\!]_\bullet\ \};\ t && (\alpha\text{-renaming } (v_0 \neq k_0)) \\
&= \textbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{v_0}\ \};\ t \\
&= \textbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow k_0$ (note that this means that $k_0 \notin \mathsf{FV}(s_0)$):
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_k = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_{k_0} = t_0$. We distinguish

whether $k \in \mathsf{FV}(\mathbf{def}\ f(x)\ \{\ s_0\ \};\ s)$ or not.
If so, we have $\mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!] = \mathbf{run}(k)\ \{\ \mathbf{def}\ f(x)\ \{\ s_0\ \};\ s\ \} \rightsquigarrow \bullet$.
Since $\mathcal{C}[\![\ \mathbf{run}(k)\ \{\ s\ \}\ ]\!]_\bullet = \mathcal{C}[\![\ s\ ]\!]_k$ we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \mathbf{def}\ f(x)\ \{\ s_0\ \};\ s\ ]\!]_k \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \};\ \mathcal{C}[\![\ s\ ]\!]_k \qquad (k_0 \text{ is fresh}) \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

If not, we have $\mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!] = \mathbf{def}\ f(x)\ \{\ s_0\ \};\ s \rightsquigarrow k$ and we likewise obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_k \\
&= \mathcal{C}[\![\ \mathbf{def}\ f(x)\ \{\ s_0\ \};\ s\ ]\!]_k \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_k = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_\bullet = t_0$. We distinguish whether $k \in \mathsf{FV}(\mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s)$ or not.
If so, we have $\mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!] = \mathbf{run}(k)\ \{\ \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s\ \} \rightsquigarrow \bullet$.
Since $\mathcal{C}[\![\ \mathbf{run}(k)\ \{\ s\ \}\ ]\!]_\bullet = \mathcal{C}[\![\ s\ ]\!]_k$ we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s\ ]\!]_k \\
&= \mathbf{let}\ f(x \mid k_1)\ \{\ \mathcal{C}[\![\ \mathbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ ]\!]_{k_1}\ \};\ \mathcal{C}[\![\ s\ ]\!]_k \qquad (k_1 \text{ is fresh}) \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \};\ t \qquad (\alpha\text{-renaming}) \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

If not, we have $\mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!] = \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s \rightsquigarrow k$ and we likewise obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_k \\
&= \mathcal{C}[\![\ \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow s_0\ \}\ \};\ s\ ]\!]_k \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow v_0$ where $v_0 \neq k_0$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_k = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_{v_0} = t_0$. We distinguish whether $k \in \mathsf{FV}(\mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow \mathbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s)$ or not.
If so, we have
$\mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!] = \mathbf{run}(k)\ \{\ \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow \mathbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s\ \} \rightsquigarrow \bullet$.
Since $\mathcal{C}[\![\ \mathbf{run}(k)\ \{\ s\ \}\ ]\!]_\bullet = \mathcal{C}[\![\ s\ ]\!]_k$ we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow \mathbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s\ ]\!]_k \\
&= \mathbf{let}\ f(x \mid k_1)\ \{\ \mathcal{C}[\![\ \mathbf{suspend}\ \{\ k_0 \Rightarrow \mathbf{run}(v_0)\ \{\ s_0\ \}\ \}\ ]\!]_{k_1}\ \};\ \mathcal{C}[\![\ s\ ]\!]_k \qquad (k_1 \text{ is fresh}) \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ \mathbf{run}(v_0)\ \{\ s_0\ \}\ ]\!]_\bullet\ \};\ t \qquad (\alpha\text{-renaming } (v_0 \neq k_0)) \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{v_0}\ \};\ t \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

If not, we have $\mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!] = \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow \mathbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s \rightsquigarrow k$ and we likewise obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_k \\
&= \mathcal{C}[\![\ \mathbf{def}\ f(x)\ \{\ \mathbf{suspend}\ \{\ k_0 \Rightarrow \mathbf{run}(v_0)\ \{\ s_0\ \}\ \}\ \};\ s\ ]\!]_k \\
&= \mathbf{let}\ f(x \mid k_0)\ \{\ t_0\ \};\ t
\end{aligned}$$

case $\textsc{Cnt}$

Given $\Gamma \vdash \textbf{cnt}\ k_0(x : \tau)\ \{\ t_0\ \};\ t$ we have $\Gamma,\ x : \tau \vdash t_0$ and $\Gamma,\ k_0 : \neg\,\tau \vdash t$. We distinguish whether $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$, $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k$ with $k \neq k_0$ or $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k_0$. In each case we need to distinguish several subcases for $\mathcal{D}[\![\ t_0\ ]\!]$.

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_\bullet = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_\bullet = t_0$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet\\
&= \mathcal{C}[\![\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ ]\!]_\bullet\\
&= \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \};\ \mathcal{C}[\![\ s\ ]\!]_\bullet\\
&= \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow x$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_\bullet = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_x = t_0$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet\\
&= \mathcal{C}[\![\ \textbf{process}\ k_0(x)\ \{\ \textbf{run}(x)\ \{\ s_0\ \}\ \};\ s\ ]\!]_\bullet\\
&= \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ \textbf{run}(x)\ \{\ s_0\ \}\ ]\!]_\bullet\ \};\ \mathcal{C}[\![\ s\ ]\!]_\bullet\\
&= \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_x\ \};\ t\\
&= \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow v_0$ where $v_0 \neq x$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_\bullet = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_{v_0} = t_0$. We distinguish whether $v_0 \in \mathsf{FV}(\textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0)$ or not.
If so, we have $\mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!] = \textbf{run}(v_0)(\textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0) \rightsquigarrow \bullet$.
Since $\mathcal{C}[\![\ \textbf{run}(k)\ \{\ s\ \}\ ]\!]_\bullet = \mathcal{C}[\![\ s\ ]\!]_k$ we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet\\
&= \mathcal{C}[\![\ \textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0\ ]\!]_{v_0}\\
&= \textbf{cnt}\ k_1(x) = \mathcal{C}[\![\ s_0\ ]\!]_{v_0};\ \mathcal{C}[\![\ \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \}\ ]\!]_{k_1} && (k_1 \text{ is fresh})\\
&= \textbf{cnt}\ k_0(x) = t_0;\ \mathcal{C}[\![\ s\ ]\!]_\bullet && (\alpha\text{-renaming})\\
&= \textbf{cnt}\ k_0(x) = t_0;\ t
\end{aligned}$$

If not, we have $\mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!] = \textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0 \rightsquigarrow v_0$.
and we likewise obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_{v_0}\\
&= \mathcal{C}[\![\ \textbf{val}\ x = \textbf{suspend}\ \{\ k_0 \Rightarrow s\ \};\ s_0\ ]\!]_{v_0}\\
&= \textbf{cnt}\ k_0(x) = t_0;\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k$ with $k \neq k_0$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_k = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_\bullet = t_0$. We distinguish whether $k \in \mathsf{FV}(\textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s)$ or not.
If so, we have $\mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!] = \textbf{run}(k)\ \{\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ \} \rightsquigarrow \bullet$.
Since $\mathcal{C}[\![\ \textbf{run}(k)\ \{\ s\ \}\ ]\!]_\bullet = \mathcal{C}[\![\ s\ ]\!]_k$ we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet\\
&= \mathcal{C}[\![\ \textbf{process}\ k_0(x)\ \{\ s_0\ \};\ s\ ]\!]_k\\
&= \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \};\ \mathcal{C}[\![\ s\ ]\!]_k\\
&= \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t
\end{aligned}$$

If not, we have $\mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ =\ \textsf{\textbf{process}}\ k_0(x)\ \{\ s_0\ \};\ s \rightsquigarrow k$ and we likewise obtain

$$\begin{aligned}&\mathcal{C}⟦\ \mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ ⟧_k\\ &= \mathcal{C}⟦\ \textsf{\textbf{process}}\ k_0(x)\ \{\ s_0\ \};\ s\ ⟧_k\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\end{aligned}$$

Case $\mathcal{D}⟦\ t\ ⟧\ =\ s \rightsquigarrow k$ with $k\ \neq\ k_0$ and $\mathcal{D}⟦\ t_0\ ⟧\ =\ s_0 \rightsquigarrow x$:
By the induction hypothesis we have $\mathcal{C}⟦\ s\ ⟧_k\ =\ t$ and $\mathcal{C}⟦\ s_0\ ⟧_x\ =\ t_0$. We distinguish whether $k\ \in \mathsf{FV}(\textsf{\textbf{process}}\ k_0(x)\ \{\ \textsf{\textbf{run}}(x)\ \{\ s_0\ \}\ \};\ s)$ or not.
If so, we have $\mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ =\ \textsf{\textbf{run}}(k)\ \{\ \textsf{\textbf{process}}\ k_0(x)\ \{\ \textsf{\textbf{run}}(x)\ \{\ s_0\ \}\ \};\ s\ \} \rightsquigarrow \bullet$.
Since $\mathcal{C}⟦\ \textsf{\textbf{run}}(k)\ \{\ s\ \}\ ⟧_\bullet\ =\ \mathcal{C}⟦\ s\ ⟧_k$ we obtain

$$\begin{aligned}&\mathcal{C}⟦\ \mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ ⟧_\bullet\\ &= \mathcal{C}⟦\ \textsf{\textbf{process}}\ k_0(x)\ \{\ \textsf{\textbf{run}}(x)\ \{\ s_0\ \}\ \};\ s\ ⟧_k\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ \{\ \mathcal{C}⟦\ \textsf{\textbf{run}}(x)\ \{\ s_0\ \}\ ⟧_\bullet\ \};\ \mathcal{C}⟦\ s\ ⟧_k\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ \{\ \mathcal{C}⟦\ s_0\ ⟧_x\ \};\ t\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\end{aligned}$$

If not, we have $\mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ =\ \textsf{\textbf{process}}\ k_0(x)\ \{\ \textsf{\textbf{run}}(x)\ \{\ s_0\ \}\ \};\ s \rightsquigarrow k$ and we likewise obtain

$$\begin{aligned}&\mathcal{C}⟦\ \mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ ⟧_k\\ &= \mathcal{C}⟦\ \textsf{\textbf{process}}\ k_0(x)\ \{\ \textsf{\textbf{run}}(x)\ \{\ s_0\ \}\ \};\ s\ ⟧_k\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\end{aligned}$$

Case $\mathcal{D}⟦\ t\ ⟧\ =\ s \rightsquigarrow k$ with $k\ \neq\ k_0$ and $\mathcal{D}⟦\ t_0\ ⟧\ =\ s_0 \rightsquigarrow v_0$ where $v_0\ \neq\ x$:
By the induction hypothesis we have $\mathcal{C}⟦\ s\ ⟧_k\ =\ t$ and $\mathcal{C}⟦\ s_0\ ⟧_{v_0}\ =\ t_0$. We distinguish whether $v_0\ \in \mathsf{FV}(\textsf{\textbf{val}}\ x\ =\ \textsf{\textbf{suspend}}\ \{\ k_0 \Rightarrow \textsf{\textbf{run}}(k)\ \{\ s\ \}\ \};\ s_0)$ or not.
If so, we have $\mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ =\ \textsf{\textbf{run}}(v_0)(\textsf{\textbf{val}}\ x\ =\ \textsf{\textbf{suspend}}\ \{\ k_0 \Rightarrow \textsf{\textbf{run}}(k)\ \{\ s\ \}\ \};\ s_0) \rightsquigarrow \bullet$.
Since $\mathcal{C}⟦\ \textsf{\textbf{run}}(k)\ \{\ s\ \}\ ⟧_\bullet\ =\ \mathcal{C}⟦\ s\ ⟧_k$ we obtain

$$\begin{aligned}&\mathcal{C}⟦\ \mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ ⟧_\bullet\\ &= \mathcal{C}⟦\ \textsf{\textbf{val}}\ x\ =\ \textsf{\textbf{suspend}}\ \{\ k_0 \Rightarrow \textsf{\textbf{run}}(k)\ \{\ s\ \}\ \};\ s_0\ ⟧_{v_0}\\ &= \textsf{\textbf{cnt}}\ k_1(x)\ =\ \mathcal{C}⟦\ s_0\ ⟧_{v_0};\ \mathcal{C}⟦\ \textsf{\textbf{suspend}}\ \{\ k_0 \Rightarrow \textsf{\textbf{run}}(k)\ \{\ s\ \}\ \}\ ⟧_{k_1} && (k_1 \text{ is fresh})\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ =\ t_0;\ \mathcal{C}⟦\ \textsf{\textbf{run}}(k)\ \{\ s\ \}\ ⟧_\bullet && (\alpha\text{-renaming } (k\ \neq\ k_0))\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ =\ t_0;\ \mathcal{C}⟦\ s\ ⟧_k\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ =\ t_0;\ t\end{aligned}$$

If not, we have $\mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ =\ \textsf{\textbf{val}}\ x\ =\ \textsf{\textbf{suspend}}\ \{\ k_0 \Rightarrow \textsf{\textbf{run}}(k)\ \{\ s\ \}\ \};\ s_0 \rightsquigarrow v_0$.
and we likewise obtain

$$\begin{aligned}&\mathcal{C}⟦\ \mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ ⟧_{v_0}\\ &= \mathcal{C}⟦\ \textsf{\textbf{val}}\ x\ =\ \textsf{\textbf{suspend}}\ \{\ k_0 \Rightarrow \textsf{\textbf{run}}(k)\ \{\ s\ \}\ \};\ s_0\ ⟧_{v_0}\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ =\ t_0;\ t\end{aligned}$$

Case $\mathcal{D}⟦\ t\ ⟧\ =\ s \rightsquigarrow k_0$ and $\mathcal{D}⟦\ t_0\ ⟧\ =\ s_0 \rightsquigarrow \bullet$:
By the induction hypothesis we have $\mathcal{C}⟦\ s\ ⟧_{k_0}\ =\ t$ and $\mathcal{C}⟦\ s_0\ ⟧_\bullet\ =\ t_0$. Hence we obtain

$$\begin{aligned}&\mathcal{C}⟦\ \mathcal{D}⟦\ \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\ ⟧\ ⟧_\bullet\\ &= \mathcal{C}⟦\ \textsf{\textbf{process}}\ k_0(x)\ \{\ s_0\ \};\ \textsf{\textbf{run}}(k_0)\ \{\ s\ \}\ ⟧_\bullet\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ \{\ \mathcal{C}⟦\ s_0\ ⟧_\bullet\ \};\ \mathcal{C}⟦\ \textsf{\textbf{run}}(k_0)\ \{\ s\ \}\ ⟧_\bullet\\ &= \textsf{\textbf{cnt}}\ k_0(x)\ \{\ t_0\ \};\ t\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k_0$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow x$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_{k_0} = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_x = t_0$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \textbf{process}\ k_0(x)\ \{\ \textbf{run}(x)\ \{\ s_0\ \}\ \};\ \textbf{run}(k_0)\ \{\ s\ \}\ ]\!]_\bullet \\
&= \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ \textbf{run}(x)\ \{\ s_0\ \}\ ]\!]_\bullet\ \};\ \mathcal{C}[\![\ \textbf{run}(k_0)\ \{\ s\ \}\ ]\!]_\bullet \\
&= \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_x\ \};\ \mathcal{C}[\![\ s\ ]\!]_{k_0} \\
&= \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t
\end{aligned}$$

Case $\mathcal{D}[\![\ t\ ]\!] = s \rightsquigarrow k_0$ and $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow v_0$ with $v_0 \neq x$:
By the induction hypothesis we have $\mathcal{C}[\![\ s\ ]\!]_{k_0} = t$ and $\mathcal{C}[\![\ s_0\ ]\!]_{v_0} = t_0$. We distinguish whether $v_0 \in \mathsf{FV}(\textbf{val}\ x = s;\ s_0)$ or not.
If so, we have $\mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!] = \textbf{run}(v_0)\ \{\ \textbf{val}\ x = s;\ s_0\ \} \rightsquigarrow \bullet$.
Since $\mathcal{C}[\![\ \textbf{run}(k)\ \{\ s\ \}\ ]\!]_\bullet = \mathcal{C}[\![\ s\ ]\!]_k$ and since $k_0 \notin \mathsf{FV}(s)$ we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \textbf{val}\ x = s;\ s_0\ ]\!]_{v_0} \\
&= \textbf{cnt}\ k_0(x)\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{v_0}\ \};\ \mathcal{C}[\![\ s\ ]\!]_{k_0} \qquad (k_0 \text{ is fresh}) \\
&= \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t
\end{aligned}$$

If not, we have $\mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!] = \textbf{val}\ x = s;\ s_0 \rightsquigarrow v_0$ and we likewise obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t\ ]\!]\ ]\!]_{v_0} \\
&= \mathcal{C}[\![\ \textbf{val}\ x = s;\ s_0\ ]\!]_{v_0} \\
&= \textbf{cnt}\ k_0(x)\ \{\ t_0\ \};\ t
\end{aligned}$$

case Ext
Given $\Gamma \vdash \textbf{exit}\ e$ we have $\mathcal{D}[\![\ \textbf{exit}\ e\ ]\!] = \textbf{exit}\ e \rightsquigarrow \bullet$. Hence we obtain $\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{exit}\ e\ ]\!]\ ]\!]_\bullet = \mathcal{C}[\![\ \textbf{exit}\ e\ ]\!]_\bullet = \textbf{exit}\ e$.

case If
Given $\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2$ we have $\Gamma \vdash t_1$ and $\Gamma \vdash t_2$. We distinguish cases according to the definition of $\mathcal{D}[\![\ \cdot\ ]\!]$.

Case $\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow k$ and $\mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow k$:
By the induction hypothesis we have $\mathcal{C}[\![\ s_i\ ]\!]_k = t_i$ for $i = 1, 2$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ ]\!]_k \\
&= \mathcal{C}[\![\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ ]\!]_k \\
&= \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_k\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_k \\
&= \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2
\end{aligned}$$

Case $\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow \bullet$:
By the induction hypothesis we have $\mathcal{C}[\![\ s_i\ ]\!]_\bullet = t_i$ for $i = 1, 2$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ ]\!]_\bullet \\
&= \mathcal{C}[\![\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ ]\!]_\bullet \\
&= \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_\bullet\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_\bullet \\
&= \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2
\end{aligned}$$

Case $\mathcal{D}[\![\ t_1\ ]\!] = s_1 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\ t_2\ ]\!] = s_2 \rightsquigarrow k$:

By the induction hypothesis we have $\mathcal{C}[\![\ s_1\ ]\!]_\bullet\ =\ t_1$ and $\mathcal{C}[\![\ s_2\ ]\!]_k\ =\ t_2$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ ]\!]_k\\
&=\ \mathcal{C}[\![\ \textbf{if}\ e\ \textbf{then suspend}\ \{\ k \Rightarrow s_1\ \}\ \textbf{else}\ s_2\ ]\!]_k\\
&=\ \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ \textbf{suspend}\ \{\ k \Rightarrow s_1\ \}\ ]\!]_k\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_k\\
&=\ \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_\bullet\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_k\\
&=\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2
\end{aligned}$$

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \leadsto k$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \leadsto \bullet$:
By the induction hypothesis we have $\mathcal{C}[\![\ s_1\ ]\!]_k\ =\ t_1$ and $\mathcal{C}[\![\ s_2\ ]\!]_\bullet\ =\ t_2$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ ]\!]_k\\
&=\ \mathcal{C}[\![\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else suspend}\ \{\ k \Rightarrow s_2\ \}\ ]\!]_k\\
&=\ \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_k\ \textbf{else}\ \mathcal{C}[\![\ \textbf{suspend}\ \{\ k \Rightarrow s_2\ \}\ ]\!]_k\\
&=\ \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_k\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_\bullet\\
&=\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2
\end{aligned}$$

Case $\mathcal{D}[\![\ t_1\ ]\!]\ =\ s_1 \leadsto k_1$ and $\mathcal{D}[\![\ t_2\ ]\!]\ =\ s_2 \leadsto k_2$:
By the induction hypothesis we have $\mathcal{C}[\![\ s_i\ ]\!]_{k_i}\ =\ t_i$ for $i\ =\ 1,\ 2$. Hence we obtain

$$\begin{aligned}
&\mathcal{C}[\![\ \mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ ]\!]\ ]\!]_{k_2}\\
&=\ \mathcal{C}[\![\ \textbf{if}\ e\ \textbf{then suspend}\ \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ \textbf{else}\ s_2\ ]\!]_{k_2}\\
&=\ \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ \textbf{suspend}\ \{\ k_2 \Rightarrow \textbf{run}(k_1)\ \{\ s_1\ \}\ \}\ ]\!]_{k_2}\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_{k_2}\\
&=\ \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ \textbf{run}(k_1)\ \{\ s_1\ \}\ ]\!]_\bullet\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_{k_2}\\
&=\ \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_{k_1}\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_{k_2}\\
&=\ \textbf{if}\ e\ \textbf{then}\ t_1\ \textbf{else}\ t_2
\end{aligned}$$

◀

Next, we can show that the right inverse property can be extended to machines (Theorem 16).

Proof. For each statement we distinguish cases according to the definition of the corresponding direct-style translation. We use in various cases that the direct-style translations do not introduce fresh variables.

Configurations
Given $\vdash\ \langle\ \mathsf{E}\ \|\ t\ \rangle$ we distinguish whether $\mathcal{D}[\![\ t\ ]\!]\ =\ s \leadsto \bullet$ or $\mathcal{D}[\![\ t\ ]\!]\ =\ s \leadsto k$.

case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \leadsto \bullet$
In this case we have $\mathcal{C}[\![\ s\ ]\!]_\bullet\ =\ t$ by Theorem 15. Using the statement for environments we thus obtain

$$\begin{aligned}
&\mathcal{C}_M[\![\ \mathcal{D}_M[\![\ \langle\ \mathsf{E}\ \|\ t\ \rangle\ ]\!]\ ]\!]\\
&=\ \mathcal{C}_M[\![\ \langle\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ \|\ s\ \rangle\ ]\!]\\
&=\ \langle\ \mathcal{C}_E[\![\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ ]\!]\ \|\ \mathcal{C}[\![\ s\ ]\!]_\bullet\ \rangle\\
&=\ \langle\ \mathsf{E}\ \|\ t\ \rangle
\end{aligned}$$

case $\mathcal{D}[\![\ t\ ]\!]\ =\ s \leadsto k$
In this case we have $\mathcal{C}[\![\ s\ ]\!]_k\ =\ t$ by Theorem 15. Note that $k\ \notin \mathsf{FV}(s)$ and also that $k\ \notin \mathsf{dom}(\mathsf{E.rm}(k))$. Using the statement for environments and frames and

$\mathcal{D}_E[\![\; \mathsf{E} \;]\!].\mathsf{rm}(k) \;=\; \mathcal{D}_E[\![\; \mathsf{E}.\mathsf{rm}(k) \;]\!]$ we obtain

$$\begin{aligned}
&\mathcal{C}_M[\![\; \mathcal{D}_M[\![\; \langle \; \mathsf{E} \parallel t \; \rangle \;]\!] \;]\!] \\
&= \mathcal{C}_M[\![\; \langle \; \mathcal{D}_E[\![\; \mathsf{E} \;]\!].\mathsf{rm}(k) \parallel s \parallel \mathcal{D}_K[\![\; \mathsf{E}(k) \;]\!] \; \rangle \;]\!] \\
&= \langle \; \mathcal{C}_E[\![\; \mathcal{D}_E[\![\; \mathsf{E}.\mathsf{rm}(k) \;]\!] \;]\!], \; k \mapsto \mathcal{C}_K[\![\; \mathcal{D}_K[\![\; \mathsf{E}(k) \;]\!] \;]\!] \parallel \mathcal{C}[\![\; s \;]\!]_k \; \rangle \qquad (k \text{ is fresh}) \\
&= \langle \; \mathsf{E}.\mathsf{rm}(k), \; k \mapsto \mathsf{E}(k) \parallel t \; \rangle \\
&= \langle \; \mathsf{E} \parallel t \; \rangle
\end{aligned}$$

Values

Given $\vdash_{val} \; \mathsf{V}$ we distinguish whether V is an integer, a closure or a frame.

case $\mathsf{V} \;=\; 19$

This case is immediate: $\mathcal{C}_V[\![\; \mathcal{D}_V[\![\; 19 \;]\!] \;]\!] \;=\; \mathcal{C}_V[\![\; 19 \;]\!] \;=\; 19$.

case $\mathsf{V} \;=\; \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow t_0 \; \}$ (note that this means that $k_0 \;\notin \mathsf{dom}(\mathsf{E})$)

We need to distinguish three subcases, $\mathcal{D}[\![\; t_0 \;]\!] \;=\; s_0 \rightsquigarrow k_0$, $\mathcal{D}[\![\; t_0 \;]\!] \;=\; s_0 \rightsquigarrow \bullet$ and $\mathcal{D}[\![\; t_0 \;]\!] \;=\; s_0 \rightsquigarrow v_0$ with $v_0 \;\neq\; k_0$. These are analogous to the corresponding cases for Let in theorem 15.

Case $\mathcal{D}[\![\; t_0 \;]\!] \;=\; s_0 \rightsquigarrow k_0$ (note that this means that $k_0 \;\notin \mathsf{FV}(s_0)$):

By Theorem 15 we have $\mathcal{C}[\![\; s_0 \;]\!]_{k_0} \;=\; t_0$. With the statement for environments we thus obtain

$$\begin{aligned}
&\mathcal{C}_V[\![\; \mathcal{D}_V[\![\; \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow t_0 \; \} \;]\!] \;]\!] \\
&= \mathcal{C}_V[\![\; \{ \; \mathcal{D}_E[\![\; \mathsf{E} \;]\!], \; (x \; : \; \tau) \Rightarrow s_0 \; \} \;]\!] \\
&= \{ \; \mathcal{C}_E[\![\; \mathcal{D}_E[\![\; \mathsf{E} \;]\!] \;]\!], \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow \mathcal{C}[\![\; s_0 \;]\!]_{k_0} \; \} \qquad (k_0 \text{ is fresh}) \\
&= \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow s_0 \; \}
\end{aligned}$$

Case $\mathcal{D}[\![\; t_0 \;]\!] \;=\; s_0 \rightsquigarrow \bullet$:

By Theorem 15 we have and $\mathcal{C}[\![\; s_0 \;]\!]_\bullet \;=\; t_0$. With the statement for environments we thus obtain

$$\begin{aligned}
&\mathcal{C}_V[\![\; \mathcal{D}_V[\![\; \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow t_0 \; \} \;]\!] \;]\!]_\bullet \\
&= \mathcal{C}_V[\![\; \{ \; \mathcal{D}_E[\![\; \mathsf{E} \;]\!], \; (x \; : \; \tau) \Rightarrow \textbf{suspend} \; \{ \; k_0 \Rightarrow s_0 \; \} \; \} \;]\!] \\
&= \{ \; \mathcal{C}_E[\![\; \mathcal{D}_E[\![\; \mathsf{E} \;]\!] \;]\!], \; (x \; : \; \tau \mid k_1 \; : \; \neg \; \tau_0) \Rightarrow \mathcal{C}[\![\; \textbf{suspend} \; \{ \; k_0 \Rightarrow s_0 \; \} \; \} \;]\!]_{k_1} \; \} \qquad (k_1 \text{ is fresh}) \\
&= \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow \mathcal{C}[\![\; s_0 \;]\!]_\bullet \; \} \qquad (\alpha\text{-renaming}) \\
&= \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow t_0 \; \}
\end{aligned}$$

Case $\mathcal{D}[\![\; t_0 \;]\!] \;=\; s_0 \rightsquigarrow v_0$ where $v_0 \;\neq\; k_0$:

By Theorem 15 we have $\mathcal{C}[\![\; s_0 \;]\!]_{v_0} \;=\; t_0$. With the statement for environments we thus obtain

$$\begin{aligned}
&\mathcal{C}_V[\![\; \mathcal{D}_V[\![\; \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow t_0 \; \} \;]\!] \;]\!]_\bullet \\
&= \mathcal{C}_V[\![\; \{ \; \mathcal{D}_E[\![\; \mathsf{E} \;]\!], \; (x \; : \; \tau) \Rightarrow \textbf{suspend} \; \{ \; k_0 \Rightarrow \textbf{run}(v_0) \; \{ \; s_0 \; \} \; \} \; \} \;]\!] \\
&= \{ \; \mathcal{C}_E[\![\; \mathcal{D}_E[\![\; \mathsf{E} \;]\!] \;]\!], \; (x \; : \; \tau \mid k_1 \; : \; \neg \; \tau_0) \Rightarrow \mathcal{C}[\![\; \textbf{suspend} \; \{ \; k_0 \Rightarrow \textbf{run}(v_0) \; \{ \; s_0 \; \} \; \} \;]\!]_{k_1} \; \} \quad (k_1 \text{ is fresh}) \\
&= \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow \mathcal{C}[\![\; \textbf{run}(v_0) \; \{ \; s_0 \; \} \;]\!]_\bullet \; \} \qquad (\alpha\text{-renaming } (v_0 \;\neq\; k_0)) \\
&= \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow \mathcal{C}[\![\; s_0 \;]\!]_{v_0} \; \} \\
&= \{ \; \mathsf{E}, \; (x \; : \; \tau \mid k_0 \; : \; \neg \; \tau_0) \Rightarrow t_0 \; \}
\end{aligned}$$

case $\mathsf{V} \;=\; \{ \; \mathsf{E}, \; (x \; : \; \tau) \Rightarrow t_0 \; \}$

In this case the result follows from the statement for frames since

$$\begin{aligned}
&\mathcal{C}_V[\![\; \mathcal{D}_V[\![\; \{ \; \mathsf{E}, \; (x \; : \; \tau) \Rightarrow t_0 \; \} \;]\!] \;]\!] \\
&= \mathcal{C}_K[\![\; \mathcal{D}_K[\![\; \{ \; \mathsf{E}, \; (x \; : \; \tau) \Rightarrow t_0 \; \} \;]\!] \;]\!] \\
&= \{ \; \mathsf{E}, \; (x \; : \; \tau) \Rightarrow t_0 \; \}
\end{aligned}$$

Environments

This follows from the statement for values. Given $\mathsf{E} \vdash_{env} \Gamma$ we distinguish whether E is empty or not.
If so, we have $\mathcal{C}_E[\![\ \mathcal{D}_E[\![\ \bullet\ ]\!]\ ]\!] = \mathcal{C}_E[\![\ \bullet\ ]\!] = \bullet$.
If not, we obtain using the induction hypothesis

$$\begin{aligned}
&\mathcal{C}_E[\![\ \mathcal{D}_E[\![\ \mathsf{E},\ x \mapsto \mathsf{V}\ ]\!]\ ]\!] \\
&= \mathcal{C}_E[\![\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ ,\ x \mapsto \mathcal{D}_V[\![\ \mathsf{V}\ ]\!]\ ]\!] \\
&= \mathcal{C}_E[\![\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ ]\!]\ ,\ x \mapsto \mathcal{C}_V[\![\ \mathcal{D}_V[\![\ \mathsf{V}\ ]\!]\ ]\!] \\
&= \mathsf{E},\ x \mapsto \mathsf{V}
\end{aligned}$$

Frames

For $\vdash_{val} \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \} : \neg\,\tau$ we distinguish whether $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$, $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow x$ or $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow k$ with $k \neq x$.

case $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow \bullet$

By theorem 15 we have $\mathcal{C}[\![\ s_0\ ]\!]_\bullet = t_0$. With the statement for environments we thus obtain

$$\begin{aligned}
&\mathcal{C}_K[\![\ \mathcal{D}_K[\![\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ ]\!]\ ]\!] \\
&= \mathcal{C}_K[\![\ \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x : \tau) \Rightarrow s_0\ \}\ ]\!] \\
&= \{\ \mathcal{C}_E[\![\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ ]\!],\ (x : \tau) \Rightarrow \mathcal{C}[\![\ s_0\ ]\!]_\bullet\ \} \\
&= \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}
\end{aligned}$$

case $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow x$

By theorem 15 we have $\mathcal{C}[\![\ s_0\ ]\!]_x = t_0$. With the statement for environments we thus obtain

$$\begin{aligned}
&\mathcal{C}_K[\![\ \mathcal{D}_K[\![\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ ]\!]\ ]\!] \\
&= \mathcal{C}_K[\![\ \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!],\ (x : \tau) \Rightarrow \textbf{run}(x)\ \{\ s_0\ \}\ \}\ ]\!] \\
&= \{\ \mathcal{C}_E[\![\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!]\ ]\!],\ (x : \tau) \Rightarrow \mathcal{C}[\![\ \textbf{run}(x)\ \{\ s_0\ \}\ ]\!]_\bullet\ \} \\
&= \{\ \mathsf{E},\ (x : \tau) \Rightarrow \mathcal{C}[\![\ s_0\ ]\!]_x\ \} \\
&= \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}
\end{aligned}$$

case $\mathcal{D}[\![\ t_0\ ]\!] = s_0 \rightsquigarrow k$ (note that this means that $k \notin \mathsf{FV}(s_0)$) with $k \neq x$

By theorem 15 we have $\mathcal{C}[\![\ s_0\ ]\!]_k = t_0$. With the statement for environments and the induction hypothesis we thus obtain (similar to the second case for configurations)

$$\begin{aligned}
&\mathcal{C}_K[\![\ \mathcal{D}_K[\![\ \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \}\ ]\!]\ ]\!] \\
&= \mathcal{C}_K[\![\ \{\ \mathcal{D}_E[\![\ \mathsf{E}\ ]\!].\mathsf{rm}(k),\ (x : \tau) \Rightarrow s_0\ \}\ ::\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ ]\!] \\
&= \{\ \mathcal{C}_E[\![\ \mathcal{D}_E[\![\ \mathsf{E}.\mathsf{rm}(k)\ ]\!]\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathcal{D}_K[\![\ \mathsf{E}(k)\ ]\!]\ ]\!],\ (x : \tau) \Rightarrow \mathcal{C}[\![\ s_0\ ]\!]_k\ \} \qquad (k \text{ is fresh}) \\
&= \{\ \mathsf{E}.\mathsf{rm}(k),\ k \mapsto \mathsf{E}(k),\ (x : \tau) \Rightarrow s_0\ \} \\
&= \{\ \mathsf{E},\ (x : \tau) \Rightarrow s_0\ \}
\end{aligned}$$

◀

D.3 Restricted Left Inverse

We first prove that $\mathcal{D}[\![\ \cdot\ ]\!]$ is the left inverse of $\mathcal{C}[\![\ \cdot\ ]\!]$ on the pure fragment of λ_{D} (Theorem 17).

Proof. Induction over the typing derivation.

case Sequence

Given $\Gamma \vdash\ \textbf{val}\ x_0 = s_0;\ s : \tau$ we have $\Gamma \vdash\ s_0 : \tau_0$ and $\Gamma,\ x : \tau_0 \vdash\ s : \tau$.
The induction hypothesis thus yields $\mathcal{D}[\![\ \mathcal{C}[\![\ s_0\ ]\!]_k\ ]\!] = s_0 \rightsquigarrow k$ and $\mathcal{D}[\![\ \mathcal{C}[\![\ s\ ]\!]_k\ ]\!] = s \rightsquigarrow k$

for fresh k.
Hence, for fresh k we obtain

$$\begin{aligned}
&\mathcal{D}[\![\ \mathcal{C}[\![\ \textbf{val}\ x_0\ =\ s_0;\ s\ ]\!]_k\ ]\!] \\
&=\ \mathcal{D}[\![\ \textbf{cnt}\ k_0(x_0)\ =\ \{\ \mathcal{C}[\![\ s\ ]\!]_k\ \};\ \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ ]\!] \\
&=\ \textbf{val}\ x_0\ =\ s_0;\ s \rightsquigarrow k && (k\ \neq\ x_0)\ (k \text{ not free})
\end{aligned}$$

case Return
Given $\Gamma \vdash\ \textbf{ret}\ e\ :\ \tau$ we have for fresh k

$$\begin{aligned}
&\mathcal{D}[\![\ \mathcal{C}[\![\ \textbf{ret}\ e\ ]\!]_k\ ]\!] \\
&=\ \mathcal{D}[\![\ k(e)\ ]\!] \\
&=\ \textbf{ret}\ e \rightsquigarrow k
\end{aligned}$$

case Define
Given $\Gamma \vdash\ \textbf{def}\ f(x)\ \{\ s_0\ \};\ s\ :\ \tau$ we have $\Gamma,\ x\ :\ \tau \vdash\ s_0\ :\ \tau_0$ and $\Gamma,\ f\ :\ \tau \rightarrow \tau_0 \vdash\ s\ :\ \tau$.
The induction hypothesis thus yields $\mathcal{D}[\![\ \mathcal{C}[\![\ s_0\ ]\!]_k\ ]\!]\ =\ s_0 \rightsquigarrow k$ and $\mathcal{D}[\![\ \mathcal{C}[\![\ s\ ]\!]_k\ ]\!]\ =\ s \rightsquigarrow k$ for fresh k. Hence, for fresh k we obtain

$$\begin{aligned}
&\mathcal{D}[\![\ \mathcal{C}[\![\ \textbf{def}\ f(x)\ \{\ s_0\ \};\ s\ ]\!]_k\ ]\!] \\
&=\ \mathcal{D}[\![\ \textbf{let}\ f(x \mid k_0)\ =\ \{\ \mathcal{C}[\![\ s_0\ ]\!]_{k_0}\ \};\ \mathcal{C}[\![\ s\ ]\!]_k\ ]\!] \\
&=\ \textbf{def}\ f(x)\ \{\ s_0\ \};\ s \rightsquigarrow k && (k \text{ not free})
\end{aligned}$$

case Call
Given $\Gamma \vdash\ f(e)\ :\ \tau$ we have for fresh k

$$\begin{aligned}
&\mathcal{D}[\![\ \mathcal{C}[\![\ f(e)\ ]\!]_k\ ]\!] \\
&=\ \mathcal{D}[\![\ f(e \mid k)\ ]\!] \\
&=\ f(e) \rightsquigarrow k && (k \text{ not free})
\end{aligned}$$

case If
Given $\Gamma \vdash\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ :\ \tau$ we have $\Gamma \vdash\ s_1\ :\ \tau$ and $\Gamma \vdash\ s_2\ :\ \tau$.
The induction hypothesis thus yields $\mathcal{D}[\![\ \mathcal{C}[\![\ s_i\ ]\!]_k\ ]\!]\ =\ s_i \rightsquigarrow k$ for fresh k for $i\ =\ 1,\ 2$.
Hence, for fresh k we obtain

$$\begin{aligned}
&\mathcal{D}[\![\ \mathcal{C}[\![\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ ]\!]_k\ ]\!] \\
&=\ \mathcal{D}[\![\ \textbf{if}\ e\ \textbf{then}\ \mathcal{C}[\![\ s_1\ ]\!]_k\ \textbf{else}\ \mathcal{C}[\![\ s_2\ ]\!]_k\ ]\!] \\
&=\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow k
\end{aligned}$$

◀

Now we can show that the restricted left inverse property can be extended to machines (Theorem 18).

Proof. For each statement we distinguish cases according to the definition of the corresponding CPS translation.

Configurations

Given $\vdash\ \langle\ \mathsf{E} \parallel s \parallel \mathsf{K}\ \rangle$ we have for fresh k that $\mathcal{D}[\![\ \mathcal{C}[\![\ s\ ]\!]_k\ ]\!]\ =\ s \rightsquigarrow k$ by Theorem 17. Hence for fresh k we use the statements for environments and stacks to obtain

$$\begin{aligned}
&\mathcal{D}_M[\![\ \mathcal{C}_M[\![\ \langle\ \mathsf{E} \parallel s \parallel \mathsf{K}\ \rangle\ ]\!]\ ]\!] \\
&=\ \mathcal{D}_M[\![\ \langle\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!],\ k \mapsto \mathcal{C}_K[\![\ \mathsf{K}\ ]\!] \parallel \mathcal{C}[\![\ s\ ]\!]_k\ \rangle\ ]\!] && (k\ \notin \mathsf{E}) \\
&=\ \langle\ \mathcal{D}_E[\![\ \mathcal{C}_E[\![\ \mathsf{E}\ ]\!]\ ]\!] \parallel s \parallel \mathcal{D}_K[\![\ \mathcal{C}_K[\![\ \mathsf{K}\ ]\!]\ ]\!]\ \rangle \\
&=\ \langle\ \mathsf{E} \parallel s \parallel \mathsf{K}\ \rangle
\end{aligned}$$

Values
Given $\vdash_{val}$ V we distinguish whether V is an integer or a closure.

case $\mathsf{V} = 19$
This case is immediate: $\mathcal{D}_V\llbracket\, \mathcal{C}_V\llbracket\, 19 \,\rrbracket \,\rrbracket = \mathcal{D}_V\llbracket\, 19 \,\rrbracket = 19$.

case $\mathsf{V} = \{\ \mathsf{E},\ (x : \tau) \Rightarrow s\ \}$
By Theorem 15 we have $\mathcal{D}\llbracket\, \mathcal{C}\llbracket\, s \,\rrbracket_k \,\rrbracket = s \rightsquigarrow k$ for fresh k. With the statement for environments we thus obtain

$$\begin{aligned}
&\mathcal{D}_V\llbracket\, \mathcal{C}_V\llbracket\, \{\ \mathsf{E},\ (x : \tau) \Rightarrow s\ \} \,\rrbracket \,\rrbracket \\
&= \mathcal{D}_V\llbracket\, \{\ \mathcal{C}_E\llbracket\, \mathsf{E} \,\rrbracket,\ (x : \tau \mid k : \neg\, \tau_0) \Rightarrow \mathcal{C}\llbracket\, s \,\rrbracket_k\ \} \,\rrbracket \\
&= \{\ \mathcal{D}_E\llbracket\, \mathcal{C}_E\llbracket\, \mathsf{E} \,\rrbracket \,\rrbracket,\ (x : \tau) \Rightarrow s\ \} \\
&= \{\ \mathsf{E},\ (x : \tau) \Rightarrow s\ \}
\end{aligned}$$

Environments
This follows from the statement for values. Given $\mathsf{E} \vdash_{env} \Gamma$ we distinguish whether E is empty or not.
If so, we have $\mathcal{D}_E\llbracket\, \mathcal{C}_E\llbracket\, \bullet \,\rrbracket \,\rrbracket = \mathcal{D}_E\llbracket\, \bullet \,\rrbracket = \bullet$.
If not, we obtain using the induction hypothesis

$$\begin{aligned}
&\mathcal{D}_E\llbracket\, \mathcal{C}_E\llbracket\, \mathsf{E},\ x \mapsto \mathsf{V} \,\rrbracket \,\rrbracket \\
&= \mathcal{D}_E\llbracket\, \mathcal{C}_E\llbracket\, \mathsf{E} \,\rrbracket\ ,\ x \mapsto \mathcal{C}_V\llbracket\, \mathsf{V} \,\rrbracket \,\rrbracket \\
&= \mathcal{D}_E\llbracket\, \mathcal{C}_E\llbracket\, \mathsf{E} \,\rrbracket \,\rrbracket\ ,\ x \mapsto \mathcal{D}_V\llbracket\, \mathcal{C}_V\llbracket\, \mathsf{V} \,\rrbracket \,\rrbracket \\
&= \mathsf{E},\ x \mapsto \mathsf{V}
\end{aligned}$$

Stacks
For $\tau \vdash_{stk}$ `done` this is immediate

$$\begin{aligned}
&\mathcal{D}_K\llbracket\, \mathcal{C}_K\llbracket\, \texttt{done} \,\rrbracket \,\rrbracket \\
&= \mathcal{D}_K\llbracket\, \texttt{done} \,\rrbracket \\
&= \texttt{done}
\end{aligned}$$

as we have seen in previous proofs already.
For $\tau \vdash_{stk} \{\ \mathsf{E},\ (x : \tau) \Rightarrow s\ \} :: \mathsf{K}$ we obtain by theorem 17 that $\mathcal{D}\llbracket\, \mathcal{C}\llbracket\, s \,\rrbracket_k \,\rrbracket = s \rightsquigarrow k$ for fresh k. With the statement for environments and stacks we thus obtain

$$\begin{aligned}
&\mathcal{D}_K\llbracket\, \mathcal{C}_K\llbracket\, \{\ \mathsf{E},\ (x : \tau) \Rightarrow s\ \} :: \mathsf{K} \,\rrbracket \,\rrbracket \\
&= \mathcal{D}_K\llbracket\, \{\ \mathcal{C}_E\llbracket\, \mathsf{E} \,\rrbracket,\ k \mapsto \mathcal{C}_K\llbracket\, \mathsf{K} \,\rrbracket,\ (x : \tau) \Rightarrow \mathcal{C}\llbracket\, s \,\rrbracket_k\ \} \,\rrbracket \\
&= \{\ \mathcal{D}_E\llbracket\, \mathcal{C}_E\llbracket\, \mathsf{E} \,\rrbracket \,\rrbracket,\ (x : \tau) \Rightarrow s\ \} :: \mathcal{D}_K\llbracket\, \mathcal{C}_K\llbracket\, \mathsf{K} \,\rrbracket \,\rrbracket \qquad (k \notin \mathsf{E})\ (k \neq x) \\
&= \{\ \mathsf{E},\ (x : \tau) \Rightarrow t_0\ \} :: \mathsf{K}
\end{aligned}$$

◀

D.4 Semantics Reflection

Next, we prove the corollaries about reflection of machine steps (19, 20 and 21).

D.4.1 DS Translation

Let us first look at the DS translation. If there is a step in the DS-machine between translated machine configurations then there must be a step in the CPS-machine between the original states.

Proof. Note that by Theorem 7 we have $\vdash \mathcal{D}_M[\![\mathsf{M}]\!]$. Thus, Theorems 16 and 8 yield

$$\mathsf{M} = \mathcal{C}_M[\![\mathcal{D}_M[\![\mathsf{M}]\!]]\!] \to^{?} \mathcal{C}_M[\![\mathcal{D}_M[\![\mathsf{M}']\!]]\!] = \mathsf{M}'$$

However, $\mathsf{M} \to^0 \mathsf{M}'$ means $\mathsf{M} = \mathsf{M}'$ which would imply $\mathcal{D}_M[\![\mathsf{M}]\!] = \mathcal{D}_M[\![\mathsf{M}']\!]$ contradicting the assumption that there is a step between these configurations. Hence, we have $\mathsf{M} \to \mathsf{M}'$. ◀

D.4.2 CPS Translation

For the CPS translation we only have such a result for pure fragment of λ_{D}. We first prove the corollary stating that evaluation in the CPS-machine is closed on the image of the pure fragment and in lockstep with the DS-machine.

Proof. Since the pure fragment of λ_{D} is closed under evaluation, $\mathsf{pure}(\mathcal{D}_M[\![\mathsf{M}']\!])$ follows from the first statement. By Theorem 7 we have $\vdash \mathcal{C}_M[\![\mathsf{M}]\!]$. Thus, Theorems 18 and 12 yield

$$\mathsf{M} = \mathcal{D}_M[\![\mathcal{C}_M[\![\mathsf{M}]\!]]\!] \to^{[1-4]} \mathcal{D}_M[\![\mathsf{M}']\!]$$

Hence, there is M'' such that $\mathsf{M} \to \mathsf{M}'' \to^{[0-3]} \mathcal{D}_M[\![\mathsf{M}']\!]$. Note that since the pure fragment is closed we have $\mathsf{pure}(\mathsf{M}'')$ and preservation for the pure fragment yields $\vdash \mathsf{M}''$. Theorem 9 implies $\mathcal{C}_M[\![\mathsf{M}]\!] \to \mathcal{C}_M[\![\mathsf{M}'']\!]$ and since the machine steps deterministically (see theorem 4) we have $\mathcal{C}_M[\![\mathsf{M}'']\!] = \mathsf{M}'$. Hence, $\mathsf{M}'' = \mathcal{D}_M[\![\mathcal{C}_M[\![\mathsf{M}'']\!]]\!] = \mathcal{D}_M[\![\mathsf{M}']\!]$ by theorem 18. ◀

Now step-reflection is immediate.

Proof. By corollary 20 we have $\mathsf{M} \to \mathcal{D}_M[\![\mathcal{C}_M[\![\mathsf{M}']\!]]\!]$ and theorem 18 yields the result. ◀