Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Abstract Machine Semantics for Quipper

Relatore:
Chiar.mo Prof.
UGO DAL LAGO

Presentata da:
ANDREA COLLEDAN

Sessione III
Anno Accademico 2019-2020

Contents

Introduction

As progress is made in the physical realization of new and more powerful quantum computers, the need for a quantum programming language that goes beyond a mere instruction set for quantum hardware and instead offers high-level features similar to those we are already used to in classical programming becomes more and more apparent. Today, it would be ridiculous if a programmer were to try and code a web application by defining every part of it in terms of logic gates. Similarly, it is unreasonable to expect that a quantum programmer in the future will describe all of its quantum algorithms in terms of elementary unitary transformations. At the time of writing, one of the most promising candidates for a quantum programming language suitable to real-world applications is undoubtedly Quipper [6, 7].

Quipper is a functional programming language for the description of quantum circuits. What sets Quipper apart from the majority of the remaining quantum programming languages is that it is designed with the explicit objective of being practical, scalable and ultimately useful. To this effect, Quipper is implemented as an embedded language in Haskell, so that all of the advanced programming constructs that are available in Haskell are also available when programming in Quipper. The result is a powerful quantum programming language that does not limit the programmer to a gate-by-gate description of quantum computations, but rather treats circuits themselves as data and supports many higher-order operations to combine them together and manipulate them in their entirety. This allows for the implementation of many real-world quantum algorithms that would be practically inexpressible in other programming languages, if anything due to the sheer size of their circuits.

Unfortunately, while Quipper inherits all of the qualities of Haskell, it also inherits its shortcomings. Namely, Quipper lacks linear types, which are critical to quantum programming, and more generally a formal operational semantics. As a consequence, it is difficult to reason rigorously about the behavior of Quipper programs, a fact that constitutes an obstacle to the application of otherwise valuable static analysis techniques to them. For example, the ability to statically infer bounds on the number of qubits required at run time by a Quipper program would be immensely useful in a time where

quantum resources – although increasingly available – are still scarce. Fortunately, a number of research languages exist that formalize significant fragments of Quipper in a type-safe way. In this thesis, we examine one such language, namely Rios and Selinger's Proto-Quipper-M [13], and use its big-step semantics as a starting point to define a new operational semantics for Quipper which is inspired by abstract machines. We then prove that this new semantics is equivalent to the original one. Our hope is that our work will in turn serve as a valuable starting point for future research in the formalization of more advanced Quipper constructs and in the static analysis of Quipper programs.

Contents of the Thesis

In Chapter 1 we first introduce the quantum circuit model of computing and then we present Quipper as a programming language for the description of quantum circuits. We do so informally, by showing a number of increasingly sophisticated Quipper programs that exemplify the most relevant constructs of the language. Finally, after having illustrated Quipper's qualities, we discuss its main shortcomings and how they impact the programming experience.

In Chapter 2 we introduce Proto-Quipper-M, a type-safe formalization of a relevant fragment of Quipper. We start by introducing the categorical model upon which circuit construction in Proto-Quipper-M is built. Then we proceed to present the language itself, with a particular focus on its linear type system, which can prevent at compile time a number of mistakes that would result in a run time exception in Quipper, namely those related to the violation of the no-cloning property of quantum states. To conclude the section, we cover Proto-Quipper-M's big-step semantics and make a first attempt to define an equivalent small-step semantics. We give safety results for the resulting semantics and we assess its limitations, specifically as far as the circuit boxing operation is concerned.

In Chapter 3 we present two incremental upgrades to the small-step semantics defined in the previous chapter. First, we propose a *stacked semantics*, which overcomes the shortcomings of the previous semantics by introducing an explicit stack into the small-step semantics, to keep track of nested boxing operations. Next, we take the stack approach even further and formulate a proposal for a *machine semantics* for Proto-Quipper-M. This semantics is heavily inspired by abstract machines, and particularly by the CEK machine [4], as it models every phase of the evaluation of a Proto-Quipper-M program as a continuation on a stack. Unlike the CEK machine, however, our machine does not employ environments, and rather relies on an abstract substitution function.

Finally, in the more technical Chapter 4 we analyze the three semantics in their relationship with one another, eventually proving that the proposed machine semantics is effectively equivalent to the starting small-step semantics and – as a consequence – to the original Proto-Quipper-M semantics given by Rios and Selinger.

Chapter 1

Quipper: a Quantum Circuit Description Language

Quipper is a programming language for the description of quantum circuits. In order to understand what it means to describe – or construct – a quantum circuit, it is necessary to know what a quantum circuit is in the first place. We therefore begin this chapter with a brief explanation of the quantum circuit model and then proceed with an introduction of Quipper itself.

1.1 What is a Quantum Circuit?

The quantum circuit model is one of the most widespread models for the study of quantum computations. In it, a computation is modelled as a sequence of elementary *quantum logic gates* which are applied to one or more *wires*, hence the name "circuit". Conceptually, every wire represents an individual qubit, while the quantum gates that are applied to a wire represent unitary transformations of the qubit's quantum state. The circuit metaphor allows us to represent quantum computations graphically in a very intuitive manner, as one can see in Figure 1.1

Note that a satisfactory introduction to the remarkable field of quantum mechanics is outside the scope of this thesis. As such, when we describe quantum circuits in more detail in the pages to come, we take for granted that the reader is already somewhat familiar with the fundamental concepts of quantum mechanics (such as superposition, measurement, entanglement, etc.) and the mathematical notation used to reason about them (such as the bra-ket notation, state vectors, unitary matrices, and so on). The reader who is unfamiliar with these essential ideas can find a minimal introduction to the topic in the author's bachelor thesis [3]. Alternatively, for a more thorough introduction to quantum mechanics and their application to computer science, we refer the reader to

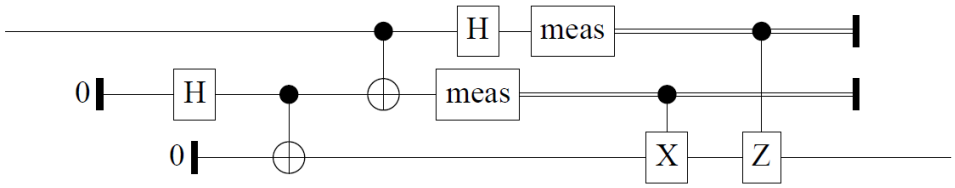


Figure 1.1: A graphical representation of a quantum circuit, where three wires run from left to right. The H, X, Z and $\oplus$ symbols represent each a different quantum gate being applied to the wires, while $\bullet$ denotes a *control qubit* and "meas" denotes the *measurement* of a wire, which results in a bit wire. Of the three wires, the top one is an *input* to the circuit, while the other two are introduced as part of it. Similarly, the first two wires are *discarded* at the end of the circuit and only the bottom wire is returned as an *output*. Because the middle wire is introduced, used and discarded entirely within the circuit, we call it an *ancilla*. This circuit implements the quantum teleportation algorithm.

textbooks such as the ones by Yanofsky and Mannucci [15] and Nielsen and Chuang [10].

1.1.1 Qubits

As we already mentioned, a wire represents a qubit, which is the quantum counterpart of a (classical) bit. Mathematically, a qubit is nothing more than a quantum system which can exist in a superposition of two basis states, which we usually refer to as $|0\rangle$ and $|1\rangle$:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Intuitively, two or more qubits can be considered together in what is commonly called a *quantum register*. In the specific context of quantum circuits, it is perhaps more intuitive to interpret a quantum register as a bundle of wires. If two qubits are respectively in states $|\phi\rangle$ and $|\psi\rangle$, then the state of the register containing both qubits is given by the *tensor product* of $|\phi\rangle$ and $|\psi\rangle$, which is written $|\phi\rangle \otimes |\psi\rangle$, or $|\phi\rangle|\psi\rangle$ or $|\phi\psi\rangle$. The Kronecker product is usually chosen for the tensor product, and it is defined on vectors and matrices alike as follows:

$$A \otimes B = \begin{bmatrix} A_{1,1}B & A_{1,2}B & \dots & A_{1,m}B \\ A_{2,1}B & A_{2,2}B & \dots & A_{2,m}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1}B & A_{n,2}B & \dots & A_{n,m}B \end{bmatrix},$$

where $A_{i,j}B$ represents the product of scalar $A_{i,j}$ with matrix B. Note that if A is a $n \times m$ matrix and B is a $p \times q$ matrix, then $A \otimes B$ is a $np \times mq$ matrix. As an example

of how the composition of qubits works, a quantum register of two qubits can exist in a superposition of the following four basis states:

$$|00\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\begin{bmatrix}1\\0\end{bmatrix}\\0\begin{bmatrix}1\\0\end{bmatrix}\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \quad |01\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\begin{bmatrix}0\\1\end{bmatrix}\\0\begin{bmatrix}0\\1\end{bmatrix}\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix},$$

$$|10\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\begin{bmatrix}1\\0\end{bmatrix}\\1\begin{bmatrix}1\\0\end{bmatrix}\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \quad |11\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\begin{bmatrix}0\\1\end{bmatrix}\\1\begin{bmatrix}0\\1\end{bmatrix}\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}.$$

Whereas we can always take the states of two qubits and tensor them together to get the state of the quantum register containing those two qubits, the reverse operation is not always possible. That is, if $|\epsilon\rangle$ is the state of a two-qubit quantum register, then in general it is not possible to find two single-qubit states $|\phi\rangle$ and $|\psi\rangle$ such that $|\epsilon\rangle = |\phi\rangle \otimes |\psi\rangle$. If we can do so, then we say that $|\epsilon\rangle$ is a *separable* state, whereas if we cannot we say that $|\epsilon\rangle$ is an *entangled* state. In the latter case, we often just say that the two qubits that make up the register are entangled. Informally, this means that the state of one qubit is only describable in terms of the state of the other (e.g. we can say that they are either both $|0\rangle$ or both $|1\rangle$) and not individually. Entanglement is at the base of almost every quantum algorithm.

1.1.2 Quantum Logic Gates

In the case of classical circuits, the fundamental building blocks of any computation on bits are logic gates, such as the NOT gate, the AND gate, the XOR gate, and so on. The operations carried out by these gates are trivial in isolation, but by composing them together we can design circuits that perform advanced arithmetic or implement control flow and memory. The name *quantum logic gate* therefore hints at the existence of similar basic building blocks with which to build quantum circuits that perform arbitrarily complex quantum computations. However, a significant difference between the classical and the quantum case is that in the latter a system can only evolve by means of reversible transformations (unless a measurement is involved). As a result, the basic building blocks that we are looking for must be reversible too. This is not generally the case with logic gates (in fact, out of the aforementioned classical gates only NOT is reversible), which means that we need a completely different set of gates with which to describe quantum circuits. In the following paragraphs, we briefly describe some of the quantum gates that occur most frequently in the literature, describing their effect on qubits both informally and formally, as unitary transformations.

Pauli-X Gate The *Pauli-X gate* is the most basic (i.e. least exotic) quantum logic gate. It is usually denoted by the letter X or, alternatively, the $\oplus$ symbol. Intuitively, this gate is the exact quantum counterpart of the classical NOT gate, as it maps the basis state $|0\rangle$ to the basis state $|1\rangle$ and vice-versa:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle, \quad X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle.$$

Note that because X maps classical states into classical states, it cannot be used to introduce superpositions into a computation.

Pauli-Z Gate The *Pauli-Z gate* is the first truly quantum logic gate that we examine, in that it does not correspond to any classical operation. Its action on a single qubit can be described as follows:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix},$$

$$Z|0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle, \quad Z|1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} = -|1\rangle.$$

Informally, the Z gate *shifts the phase* of a qubit. Phase is a property of all quantum states and plays a relevant role in many quantum algorithms. However, for the sake of this introduction we will not delve into these aspects of quantum computing.

Hadamard Gate The *Hadamard gate* is denoted by the letter H and it is one of the most fundamental quantum logic gates, as it can be used to put a single qubit into a perfect superposition. More formally, it has the following effect:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \quad H|1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix},$$

where the two resulting states are equally perfect superpositions which only differ in phase. Here, *perfect* refers to the fact that after the application of the Hadamard gate a qubit has an equal probability of being measured $|0\rangle$ or $|1\rangle$.

Controlled Gates

In addition to the simple quantum gates that we examined so far, in the literature we often encounter gates that are *controlled* by an additional qubit, which we call a *control qubit*. Intuitively, if the control qubit is in state $|0\rangle$, then the controlled gate does nothing to its inputs, whereas if the control qubit is in state $|1\rangle$ the controlled gate works as usual. Although any gate can be controlled, the most common controlled gate by far is the controlled-NOT, or CNOT gate.

CNOT gate The *CNOT gate* takes as input a *control qubit* and a *target qubit*. Graphically, the two qubits are usually connected by a line and denoted respectively by the $\bullet$ and $\oplus$ symbols, as seen in Figure 1.1. The CNOT gate applies the X gate to the target qubit if the control is in state $|1\rangle$, otherwise it leaves the two qubits unaltered. More formally:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

$$CNOT|00\rangle = |00\rangle, \quad CNOT|01\rangle = |01\rangle,$$

$$CNOT|10\rangle = |11\rangle, \quad CNOT|11\rangle = |10\rangle.$$

In other words, the CNOT gate always maps the two-qubit state $|x\rangle|y\rangle$ to $|x\rangle|x \oplus y\rangle$.

1.1.3 From Gates to Circuits

Now that we know what a quantum logic gate is, we need to define how quantum gates can be assembled together to form a quantum circuit. In the classical case, we know that logic gates can be wired together with a great degree of freedom, to the extent where a Boolean circuit is considered valid as long as it can be represented by an acyclic directed graph. In particular, any gate within a circuit can merge two or more input wires into a single output (*fan-in*) and redirect this output to an arbitrary number of subsequent gates (*fan-out*). The same thing cannot be done in quantum circuits. On one hand, the fan-in of wires is by its own nature non-injective, and therefore irreversible. On the other hand, the fan-out of wires violates the *no-cloning theorem*, an ubiquitous result in quantum physics which asserts that it is impossible to duplicate an arbitrary unknown quantum state. In quantum computing, this constraint is also referred to as the *no-cloning property* of quantum states, and it entails that no quantum gate can create a copy of a qubit wire.

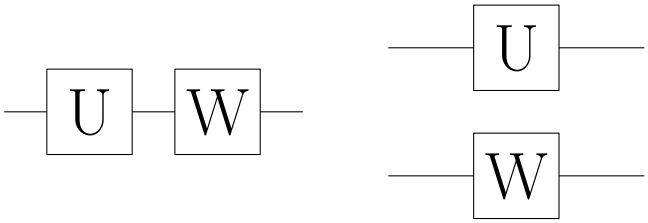


Figure 1.2: Composition of gates U and W in sequence (left) and in parallel (right).

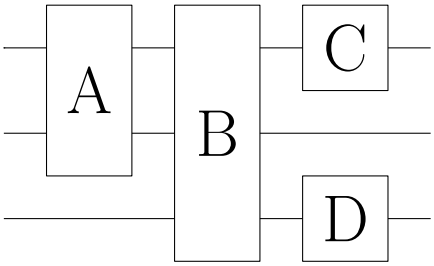


Figure 1.3: Example of a circuit made up of both multi-qubit gates and single-qubit gates, composed both in parallel and in sequence.

Because the wires of a quantum circuit cannot be split or merged, we have that the only way we can compose quantum gates into quantum circuits is either in sequence, on the same wires, or in parallel, on different wires, as seen in figure 1.2. Mathematically, composition in sequence corresponds to a simple matrix multiplication. That is, if the two gates being composed correspond to unitary transformations U and W, then their composition in series is represented by the unitary transformation

$$WU.$$

On the other hand, composition in parallel corresponds to the tensor product. That is, the composition in parallel of U and W corresponds to the unitary transformation

$$U \otimes W.$$

Naturally, this kind of composability also applies to multi-qubit gates and even to entire circuits. In this case, when we compose gates in sequence, we can choose which outputs of which gates become which inputs of which gates. That is, we are not limited to only chaining gates with the same number of inputs and outputs, as can be seen in Figure 1.3.

1.2 Quipper

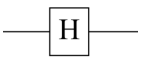
Quipper [6, 7] is a functional programming language for the description of quantum circuits. Unlike other quantum programming languages, Quipper is designed with the goal of being practical and scalable, allowing programmers to leverage the power of higher-order operators to describe quantum algorithms requiring order of trillions of gates. In order to provide this kind of power, Quipper is currently implemented as an embedded programming language in Haskell (that is, as a library and preferred idiom for Haskell), which means that it benefits from all of Haskell's features, including some advanced and experimental GHC extensions. The most noticeable consequence of this embedded approach is that Quipper is actually a purely functional language, in which circuit construction is a side-effect of the execution of a program. Because the host language is Haskell, it comes as no surprise that this side-effect is modelled as a monad.

The most effective way to introduce Quipper is probably by example, so in the following paragraphs we present and comment a number of simple programs that illustrate the essential aspects of Quipper programming. Note that most of the examples are abridged from [6].

1.2.1 Building Quantum Circuits in Quipper

Because Quipper is a circuit description language, the execution of a Quipper program naturally results in a circuit. However, when programming in Quipper it is often easier to think of a program in terms of a sequence of quantum operations being carried out imperatively in real time on a number of qubits. In fact, a quantum circuit is described in Quipper by a monadic function that takes as input qubits (or bits) and returns qubits (or bits). Consider, as a first example, the following minimal function which applies a single Hadamard gate to its input wire:

```
first :: Qubit -> Circ Qubit
first q = do
    q' <- hadamard q
    return q'
```



Let us go through this example line by line. The first line is the type of the function. Here, `Qubit` is the type of qubits, while `Circ` is the type constructor that tells us that this function is monadic and that its side-effect is precisely that of building a circuit "behind the scenes". This function therefore describes a quantum computation which takes a qubit as input and returns a qubit as output. On the third line we have the application of the Hadamard gate to the input wire `q`. The output wire of this newly applied gate is then bound to the variable `q'` for later use. Note that although the no-cloning property prevents us from reusing the same wire twice, it does not prevent us

from reusing the same wire *name* twice, so in this case we could also have written `q <- hadamard q`. Lastly, in line four `q'` is returned as the only output of the entire circuit.

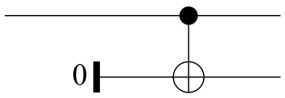
Circuit Parameters

In addition to bits and qubits, a function can also take other types as inputs, which we call *parameters*. In this case the function does not describe a single circuit, but rather a *family* of circuits. As an example, consider the following function that initializes a bit based on a Boolean parameter:

```
plus_minus :: Bool -> Circ Qubit
plus_minus b = do
    q <- qinit b
    r <- hadamard q
    return r
```

In this case, `plus_minus` is not itself a circuit, but `plus_minus False` and `plus_minus True` are. At this point it is appropriate to distinguish between the three phases of the execution of a Quipper program: *compile time*, during which the program is type-checked and compiled, *circuit generation time*, during which the parameters are known and a program is evaluated to obtain a circuit, and eventually *circuit execution time*, in which the actual inputs are given to the circuit, which can then be executed in a quantum simulator. Continuing with our examples, consider the following function, which, given a qubit, uses it as control when applying a CNOT gate to a newly introduced wire:

```
share :: Qubit -> Circ (Qubit, Qubit)
share q = do
    a <- qinit False
    a <- qnot a `controlled` q
    return (q,a)
```

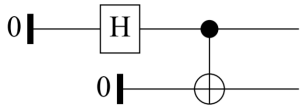


Here we have a new operation, namely `controlled`, which makes the application of any quantum operation (in this case `qnot`, which corresponds to the Pauli-X gate, to wire `a`) dependent on a control qubit, in this case `q`. Notice also how we can make use of generic Haskell constructs when describing quantum computations, such as the tuple `(q,a)` in the return statement.

Assembling Larger Circuits

So far we have only seen the application of elementary gates to wires. In the following example we initialize a pair of qubits in the entangled Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. To do so, we can take advantage of the functions defined so far to write very concise and easy to read code. This is a small demonstration the high-level and combinatorial nature of Quipper as a quantum programming language:

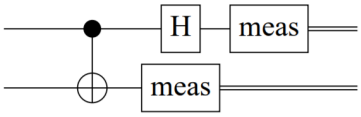
```
bell00 :: Circ (Qubit, Qubit)
bell00 = do
    q <- plus_minus False
    (q,s) <- share q
    return (q,s)
```



Quantum Teleportation Circuit

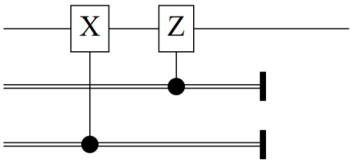
As a last example, we build the quantum teleportation circuit that we showed in Figure 1.1 at the beginning of this chapter. It is clear from the last example that we do not need (and do not want) to define the entire circuit in one shot. Rather, we can express it in terms of its two natural constituents: *Alice* and *Bob*.

```
alice :: Qubit -> Qubit -> Circ (Bit, Bit)
alice q s = do
    s <- qnot s `controlled` q
    q <- hadamard q
    (x,y) <- measure (q,s)
    return (x,y)
```



Note that this circuit performs a measurement. As a results, the two qubit wires `q` and `s` collapse to the classical wires `x` and `y`, of type `Bit`, which are then returned.

```
bob :: Qubit -> (Bit, Bit) -> Circ Qubit
bob q (x,y) = do
    q <- gate_X q `controlled` y
    q <- gate_Z q `controlled` x
    cdiscard (x,y)
    return q
```



Bob's part of the circuit controls the application of gates X and Z with the classical bits `y` and `x`, respectively. There is no fundamental difference between quantum and classical control, as the latter is just a special case of the former. Furthermore, this circuit makes use of the `cdiscard` operation, which is used to terminate and discard any amount of bit wires. The analogous operation on qubit wires is called `qdiscard` and it is not a reversible operation. Now that we have both Alice's and Bob's parts of the quantum teleportation algorithm, we can implement the circuit shown in figure 1.1 as follows:

```
teleport :: Qubit -> Circ Qubit
teleport q = do
    (a,b) <- bell00
    (x,y) <- alice q a
    b <- bob b (x,y)
    return b
```

Boxed Circuits

We conclude this introduction to Quipper's features with *boxed circuits*. Many quantum algorithms rely on multiple repetitions of gate patterns and entire sub-circuits to carry out tasks on arrays of qubits. Intuitively, instead of generating a circuit every time we need it as part of a bigger algorithm, we could generate it once and for all and then reuse it as many times as needed. This is exactly what boxing a circuit does. If we have a circuit building function `f`, we can evaluate it and store the resulting circuit for later use with the `box` operator:

```
let c = box "sub-circuit" f
```

where `"sub-circuit"` is the name given to the boxed sub-circuit. Once `c` has been built, we can use it an arbitrary number of times as part of a bigger circuit, without having to execute `f` ever again.

1.2.2 Quipper's Shortcomings

So far, the fact that Quipper is an embedded language in Haskell has allowed us to write expressive and elegant code which makes use of a number of the host's features. However, the embedding approach comes with some trade-offs and Haskell has some limitations that negatively affect Quipper. First and foremost, Haskell's type system lacks *linear types* and as such it is not expressive enough to enforce the no-cloning property of quantum states at compile time. This means that it is perfectly legal to write a Quipper program that duplicates qubit wires, such as the following one:

```
illegal :: Qubit -> Circ (Qubit, Qubit)
illegal q = do
    q' <- hadamard q
    copy_of_q' <- hadamard q
    return (q',copy_of_q')
```

This ill-formed program raises no error whatsoever at compile time, but fails immediately at circuit generation time, as soon as Quipper's runtime realizes that it violates the no-cloning property of quantum states. Another limitation of Haskell's type system is its lack of *dependent types*, which makes it harder to express and work with circuit families whose members differ in type, such as the family of Quantum Fourier Transform (QFT) circuits. In essence, despite being strongly typed, Quipper is not type-safe.

A second shortcoming is more academic in nature, but nonetheless relevant for this thesis. Because Haskell is a complex programming language with no formal semantics, it follows that Quipper too lacks a formal semantics and the behavior of Quipper programs

is purely determined by the language's implementation. It is therefore hard to reason rigorously about Quipper's features and guarantee that new additions to the language behave well with respect to the existing specification.

Chapter 2

Proto-Quipper-M: a Formalization of Quipper

At the end of the previous chapter, we mentioned that Quipper does not have a formal semantics and that it is not type-safe. In order to still be able to study Quipper in a formal way, the *Proto-Quipper* family of research languages has been introduced over the last years. Each language of this family formalizes a relevant fragment of Quipper in a type-safe way. The most prominent Proto-Quipper instances are Proto-Quipper-S [14], Proto-Quipper-M [13] and Proto-Quipper-D [5]. In particular, Proto-Quipper-M is a lambda-calculus built upon a categorical model, which features a full-fledged linear type system. A linear type system guarantees that certain variables – more technically, *linear resources* – are consumed exactly once. In the case of Proto-Quipper-M, the linear resources are the free wire ends in the circuit being built as a side-effect of the evaluation of a program. This makes it so that, unlike Quipper, Proto-Quipper-M can enforce the no-cloning property of quantum states at compile time rather than at run time. In this chapter we give an overview of Proto-Quipper-M, starting from its categorical model, which actually generalizes the notion of quantum circuit, and reviewing its syntax, type system and semantics.

2.1 Generalizing Quantum Circuits

In Proto-Quipper-M, a quantum circuit is modeled as a morphism in a symmetric monoidal category. To understand what this means, we need to know what a category is and what it means for it to be monoidal and symmetric. Note that category theory is an extensive and notoriously hard to approach area of mathematics and it would be impossible to give a satisfactory introduction to the topic – however minimal – as part of this thesis. For this reason, we focus on just the elements that are strictly necessary in order to understand the quantum circuit model of Proto-Quipper-M. The

interested reader is referred to the excellent works of Riehl [12] and Asperti and Longo [1] for a thorough introduction to the curious world of categories.

2.1.1 Generalized Circuits

Informally, a category consists of a collection of *objects* and *arrows* between those objects. The objects are generic and are treated by the arrows as atomic entities. Furthermore, for every object in a category there is an *identity* arrow which begins and ends at that object, and arrows are *composable* in the sense that for every pair of arrows going from object A to object B and from object B to object C, respectively, there exists a third arrow in the same category that goes from A to C. Because arrows usually represent some sort of "transformation" between objects, they are usually referred to as *morphisms*. More formally, the concept of category can be defined as follows.

Definition 2.1.1 (Category)**.** *A category* $\mathbf{C}$ *consists of*

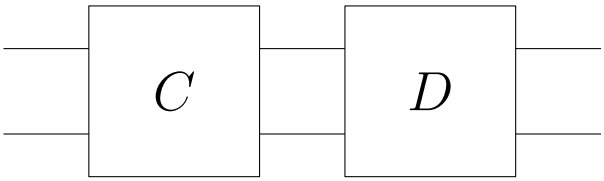
- *A class* $\mathrm{ob}(\mathbf{C})$ *of the* objects *of* $\mathbf{C}$*,*
- *A class* $\hom(\mathbf{C})$ *of the* morphisms *between objects of* $\mathbf{C}$*,*

such that

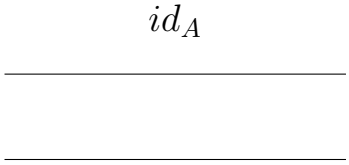
- *Every morphism in* $\hom(\mathbf{C})$ *has a defined* domain *and* codomain*. A morphism* f *with domain* A *and codomain* B *is written as* $f : A \to B$*. The set of morphisms from* A *to* B *is called the* hom-set *of* A *and* B *and it is denoted by* $\mathbf{C}(A, B)$*.*
- *For every pair of morphism* $f : A \to B$ *and* $g : B \to C$ *in* $\hom(\mathbf{C})$ *there exists a composite morphism* $f \circ g : A \to C$ *in* $\hom(\mathbf{C})$*. The composition function* $\circ$ *is associative, that is,*
$$(f \circ g) \circ h = f \circ g \circ h = f \circ (g \circ h).$$
- *Every object* A *in* $\mathrm{ob}(\mathbf{C})$ *has an associated* identity morphism $id_A : A \to A$*. The identity morphisms are left and right identities with respect to composition, that is, for all* $f : A \to B$*:*
$$id_A \circ f = f = f \circ id_B,$$

The most striking quality of the definition of category is that it makes very few assumptions about the nature of objects and morphisms. Thanks to this generality, category theory can be used to reason about a truly enormous variety of mathematical entities, and many well known categories exist that model equally well known areas of mathematics. For example, the category **Set**, whose objects are sets and whose morphisms are functions, or the category **Top**, whose objects are topological spaces and whose morphisms are continuous maps between them, or again the category **Ab**, whose objects are Abelian groups and whose morphisms are group homomorphisms, and so on.

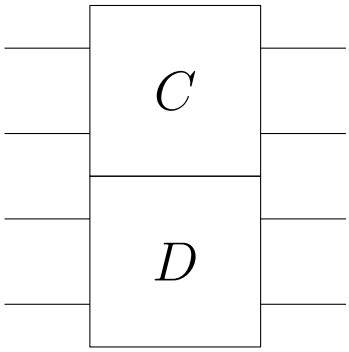
In our specific case, if we recall what we saw in Section 1.1, we can start to see how quantum circuits can be reasoned about in terms of categories. Suppose the objects of a given category $\mathbf{M}$ represent collections of wires. If $A, B \in \text{ob}(\mathbf{M})$ are two such collections, then a circuit C that takes as input the wires in A and outputs the wires in B can be clearly modelled by a morphism $C : A \to B$ in $\mathbf{M}$. Furthermore, if the wires output by a circuit C coincide with the wires taken as input by a circuit D, the two circuits can be composed in series, as shown graphically:



This is naturally modelled by the composition function $\circ$, since circuit composition is also associative. Lastly, for every collection of wires A, the circuit that does nothing to the wires in A and returns them unaltered is a perfectly valid circuit. It is modelled by the identity morphism id_A and appending or prepending it to any other circuit has no effect whatsoever on said circuit:



This is a good starting point for a quantum circuit model. However, there are some aspects of circuits which are not captured satisfactorily (or at all) by the sole notion of category. For example, recall from Section 1.1 that two circuits C and D can always be composed in parallel, as opposed to in series, as shown graphically:



This idea is not captured by the definition of category alone. As a consequence, we must turn to a more specific kind of categories, whose definition depends on two

concepts which we will now introduce briefly: *isomorphisms* and *functors*. Intuitively, an isomorphism is nothing more than an "invertible" morphism. More formally:

Definition 2.1.2 (Isomorphism)**.** *A morphism $f : A \to B$ is said to be an* isomorphism *if there exists $g : B \to A$ such that*

$$f \circ g = id_A, \qquad g \circ f = id_B.$$

We say that two objects A and B are isomorphic*, and we write $A \cong B$, when there exists an isomorphism between A and B.*

Usually, the existence of an isomorphism between two objects entails that the two can be considered equivalent from a certain perspective. Oftentimes, both sides of an isomorphism are given under the same name, and the notation $\gamma : A \cong B$ (where γ can be any name) is used to denote $f : A \to B$ and $g : B \to A$.

While an isomorphism, and more in general a morphism, establishes a mapping between different objects within the same category, a functor is a mapping between entire categories. Because categories are comprised of objects and morphisms, a functor must act on both in such a way as to preserve the basic structure of the category.

Definition 2.1.3 (Functor)**.** *Given categories $\mathbf{C}$ and $\mathbf{D}$, a* functor *$F : \mathbf{C} \to \mathbf{D}$ consists of*

- *A mapping that associates to every object A in $\mathbf{C}$ an object FA in $\mathbf{D}$.*
- *A mapping that associates to every morphism $f : A \to B$ in $\mathbf{C}$ a morphism $Ff : FA \to FB$ in $\mathbf{D}$.*

such that

- *Composition is preserved. That is, for every pair of morphisms f and g in $\mathbf{C}$,*

$$Ff \circ Fg = F(f \circ g).$$

- *Identities are preserved. That is, for every object A in $\mathbf{C}$,*

$$F(id_A) = id_{FA}$$

Given two categories $\mathbf{C}$ and $\mathbf{D}$, we can obtain their product $\mathbf{C} \times \mathbf{D}$, which is nothing more than the category whose objects are pairs (A, B), where A is an object of $\mathbf{C}$ and B is an object of $\mathbf{D}$, and whose morphisms are of the form $f \times g : (A, B) \to (C, D)$, where $f : A \to C$ is a morphism of $\mathbf{C}$ and $g : B \to D$ is a morphism of $\mathbf{D}$. In this case, the composition of morphisms is defined point-wise. A functor whose domain is a product category is called a *bifunctor*, if the product involves exactly two categories, or a *multifunctor* in general. Intuitively, a multifunctor is nothing more than a functor with more than one argument. We can now give the definition of *monoidal category*.

Definition 2.1.4 (Monoidal Category)**.** *A category* $\mathbf{C}$ *is said to be* monoidal *if it is equipped with:*

- *A bifunctor* $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$*, called* tensor product*,*
- *An object* I*, called* identity object*,*
- *Three natural isomorphism which guarantee that*
  - $\otimes$ *is associative: for all* A, B, C *in* $\mathbf{C}$ *there exists an isomorphism* $\alpha_{A,B,C} : A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$*, called* associator*, which is natural in* A, B *and* C*,*
  - I *is a left identity for* $\otimes$*: for all* A *in* $\mathbf{C}$ *there exists a natural isomorphism* $\lambda_A : I \otimes A \cong A$*, called* left unitor*,*
  - I *is a right identity for* $\otimes$*: for all* A *in* $\mathbf{C}$ *there exists a natural isomorphism* $\rho_A : A \otimes I \cong A$*, called* right unitor*,*

  and such that the following diagrams commute:
  - *For all* A, B, C, D *in* $\mathbf{C}$*:*

$$\begin{array}{ccc} A \otimes (B \otimes (C \otimes D)) & \xrightarrow{id_A \otimes \alpha_{B,C,D}} & A \otimes ((B \otimes C) \otimes D) \\ \downarrow{\scriptstyle \alpha_{A,B,C\otimes D}} & & \\ (A \otimes B) \otimes (C \otimes D) & & \downarrow{\scriptstyle \alpha_{A,B\otimes C,D}} \\ \downarrow{\scriptstyle \alpha_{A\otimes B,C,D}} & & \\ ((A \otimes B) \otimes C) \otimes D & \xleftarrow[\alpha_{A,B,C} \otimes id_D]{} & (A \otimes (B \otimes C)) \otimes D \end{array}$$

  - *For all* A, B *in* $\mathbf{C}$*:*

$$\begin{array}{ccc} A \otimes (I \otimes B) & \xrightarrow{\alpha_{A,I,B}} & (A \otimes I) \otimes B \\ & {\scriptstyle id_A \otimes \lambda_B} \searrow \quad \swarrow {\scriptstyle \rho_A \otimes id_B} & \\ & A \otimes B & \end{array}$$

If the category $\mathbf{M}$ that we are using to model quantum circuits is monoidal, then we can model circuit composition in parallel by the tensor product $\otimes$. Note that, being a functor, $\otimes$ can be applied to both morphisms (circuits) *and* objects (collections of wires), which means that whenever we have two collections of wires A and B, we can put them together into a single collection $A \otimes B$. To this effect, the identity object I represents the empty collection of wires. There is one last property that we would like to reflect in our categorical model, and it is that the wires of a quantum circuit can be rearranged freely (together with all the gates that act on them, naturally), without altering the

fundamental nature of the circuit itself. A monoidal category is not enough to model this property, so we turn to *symmetric monoidal categories*.

Definition 2.1.5 (Symmetric Monoidal Category)**.** *A monoidal category* $\mathbf{C}$ *is said to be* symmetric *when it is equipped, for all* A, B *in* $\mathbf{C}$*, with an isomorphism*

$$\gamma_{A,B} : A \otimes B \cong B \otimes A,$$

which is natural in both A *and* B *and such that the following diagrams commute*

- *For all* A *in* $\mathbf{C}$*:*

$$\begin{array}{ccc} A \otimes I & \xrightarrow{\gamma_{A,I}} & I \otimes A \\ & {\scriptstyle \rho_A} \searrow \quad \swarrow {\scriptstyle \lambda_A} & \\ & A & \end{array}$$

- *For all* A, B, C *in* $\mathbf{C}$*:*

$$\begin{array}{ccc} (A \otimes B) \otimes C & \xrightarrow{\gamma_{A,B} \otimes id_C} & (B \otimes A) \otimes C \\ \downarrow {\scriptstyle \alpha_{A,B,C}} & & \downarrow {\scriptstyle \alpha_{B,A,C}} \\ A \otimes (B \otimes C) & & B \otimes (A \otimes C) \\ \downarrow {\scriptstyle \gamma_{A,B \otimes C}} & & \downarrow {\scriptstyle id_B \otimes \gamma_{A,C}} \\ (B \otimes C) \otimes A & \xrightarrow{\alpha_{B,C,A}} & B \otimes (C \otimes A) \end{array}$$

- *For all* A, B *in* $\mathbf{C}$*:*

$$\begin{array}{c} A \otimes B \\ {\scriptstyle \gamma_{B,A}} \circlearrowleft {\scriptstyle \gamma_{A,B}} \\ B \otimes A \end{array}$$

This definition reflects the irrelevance of the order of the wires in a circuit precisely in the existence of the isomorphism γ. In conclusion, we can see that a symmetric monoidal category $\mathbf{M}$ is a good mathematical model for quantum circuits. As we mentioned earlier, Proto-Quipper-M is a language designed specifically for describing morphisms in a symmetric monoidal category, which we call *generalized circuits* from now on.

Definition 2.1.6 (Generalized Circuit)**.** *Let* $\mathbf{M}$ *be a symmetric monoidal category. We call the morphisms of* $\mathbf{M}$ generalized circuits.

Note that because this definition of circuit is so general, Proto-Quipper-M can be used to describe *any* instance of a symmetric monoidal category. This includes different quantum circuit representations (such as DAGs or unitary matrices), as well as other entities which are not necessarily quantum circuits, or circuits at all. In this respect, Proto-Quipper-M is more general than Quipper.

2.1.2 Generalized Labelled Circuits

Although the definition that we just gave is by itself sufficient to characterize a quantum circuit categorically, it would be practical – almost *necessary*, from a programming point of view – to have a way to identify and pick individual wires out of a collection, rather than treating the latter as an atomic object. We therefore introduce *labels*, which behave as a pointers to individual free wire ends, and we associate a *wire type* to each one of them. In the case of quantum circuits, the types of wires are likely to be either *bit* or *qubit*, but for the sake of generality we assume that wire types come from an arbitrary set $\mathcal{W}$, which is a parameter of the model of the language.

Definition 2.1.7 (Wire Types)**.** *Let* $\mathbf{M}$ *be a given symmetric monoidal category, and let* $\mathcal{W}$ *be a set equipped with an interpretation function*

$$[\![\cdot]\!] : \mathcal{W} \to \mathrm{ob}(\mathbf{M}),$$

that is, a mapping from the elements of $\mathcal{W}$ *to the objects of* $\mathbf{M}$*. We call the elements of* $\mathcal{W}$ wire types*.*

Wires can be considered individually or in bundles. In the second case, we assume that wire labels can be ordered and we refer to the resulting collection of mappings from labels to wire types as a *label context*.

Definition 2.1.8 (Label Context)**.** *Let* $\mathcal{L}$ *be a fixed countably infinite set of* label names*, which we assume to be totally ordered. A* label context Q *is a function of the form*

$$Q : \mathcal{L} \to \mathcal{W}.$$

Such a function that maps label names $\ell_1, \ell_2, \ldots, \ell_n$ *respectively to wire types* $\alpha_1, \alpha_2, \ldots, \alpha_n$ *can be written as follows:*

$$\ell_1 : \alpha_1, \ell_2 : \alpha_2, \ldots, \ell_n : \alpha_n.$$

The interpretation of a label context $Q = \ell_1 : \alpha_1, \ell_2 : \alpha_2, \ldots, \ell_n : \alpha_n$ *is the following object of* $\mathbf{M}$*:*

$$[\![Q]\!] = [\![\alpha_1]\!] \otimes [\![\alpha_2]\!] \otimes \cdots \otimes [\![\alpha_n]\!],$$

where $\ell_1 < \ell_2 < \cdots < \ell_n$*. In the case where* $Q = \emptyset$*, we have* $[\![\emptyset]\!] = I$*.*

At this point, by instantiating the generic objects of $\mathbf{M}$ with label contexts, we get the category $\mathbf{M}_{\mathcal{L}}$, which is a truly suitable model for a quantum circuit description language. Note that $\mathbf{M}$ and $\mathbf{M}_{\mathcal{L}}$ are essentially the same category, the only difference between the two being that $\mathbf{M}_{\mathcal{L}}$ is imbued with a labelling structure that allows us to identify individual wires and their type. To reflect this quality, we call the morphisms of $\mathbf{M}_{\mathcal{L}}$ *generalized labelled circuits*.

Definition 2.1.9 (Generalized Labelled Circuit)**.** *Let* $\mathbf{M}$ *be a given symmetric monoidal category. Let* $\mathbf{M}_{\mathcal{L}}$ *be a category in which*

- *The objects are label contexts,*
- *A morphism* $f : Q \to R$ *is a morphism* $g : [\![Q]\!] \to [\![R]\!]$ *in* $\mathbf{M}$*.*

We call the morphisms of $\mathbf{M}_{\mathcal{L}}$ generalized labelled circuits*, or just* labelled circuits*.*

2.2 Proto-Quipper-M's Syntax

We are now ready to start examining Proto-Quipper-M, starting from its syntax. Although the original Proto-Quipper-M specification given by Rios and Selinger [13] is quite rich, in this thesis we only consider a minimal fragment of the language, for the sake of simplicity. Our fragment can be described by the following grammar:

$$\begin{aligned} M, N ::= {} & x \mid \ell \mid \lambda x.M \mid MN \mid \langle M, N\rangle \mid \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N \\ & \mid \mathsf{lift}\,M \mid \mathsf{force}\,M \mid \mathsf{box}_T\,M \mid \mathsf{apply}(M, N) \mid (\vec{\ell}, C, \vec{\ell'}), \end{aligned}$$

where x ranges over variables names, ℓ ranges over the label names in $\mathcal{L}$ and C is a labelled circuit, that is, a morphism in $\mathbf{M}_{\mathcal{L}}$. The ultimate goal of the evaluation of a Proto-Quipper-M program is – unsurprisingly – the construction of a quantum circuit as a side-effect. Here, "side-effect" means that a circuit does not actually appear in the term that builds it, but rather lives "behind the scenes" and any changes made to it are, in a way, imperative in nature. For this reason, we often refer to the circuit being built by a program as the *underlying* circuit. That being said, it comes as no surprise that the most essential programming constructs of Proto-Quipper-M are those that allow to work with circuits. In particular, a term of the form $(\vec{\ell}, C, \vec{\ell'})$ is called a *boxed circuit* and allows to treat quantum circuits as data: it corresponds to a labelled quantum circuit $C : Q \to Q'$ which exposes input labels $\vec{\ell}$ and output labels $\vec{\ell'}$ as an interface (where $\vec{\ell}$ and $\vec{\ell'}$ are all and only the labels occurring in Q and Q', respectively). New boxed circuits can be introduced via the box_T construct, which corresponds to Quipper's box operator, while existing boxed circuits can be used by the apply operator, which models circuit application. Informally, with apply we can attach a boxed circuit to one or more exposed wires of the underlying circuit, whereas with box_T we can take a circuit-building

function, execute it in a sandboxed environment (with new labels created on-the-fly), box the resulting circuit and obtain it as a result. Note that we often employ $\vec{\ell}$ as syntactic sugar to denote an arbitrary tuple of labels. More formally:

$$\vec{\ell}, \vec{k} ::= \ell \mid \langle \vec{\ell}, \vec{k} \rangle.$$

Values are a subset of terms and they can be defined by the following grammar:

$$V, W ::= \ell \mid \lambda x.M \mid \langle V, W \rangle \mid \mathsf{lift}\, M \mid (\vec{\ell}, C, \vec{\ell'}).$$

Note that the original work by Rios and Selinger includes constants in the language. In particular, it assumes that for every quantum gate there exists a function constant that applies the corresponding gate to the underlying circuit, like in Quipper. For simplicity, we decided to omit constants from the language. Instead, we assume that these constants exist as specific morphisms in the $\mathbf{M}_{\mathcal{L}}$ category. For example, we assume that a morphism $H : (\ell : \mathsf{Qubit}) \to (\ell' : \mathsf{Qubit})$ exists and represents the circuit where the Hadamard gate is applied to a single qubit wire ℓ to obtain an output qubit wire ℓ'. We can therefore refer to the Hadamard gate within our language with a term of the form (ℓ, H, ℓ'). We now proceed to give some standard preliminary definitions which will be used in the coming sections.

Definition 2.2.1 (Free Labels)**.** *The set of* free labels *of a term M, denoted as $FL(M)$, is defined as follows:*

$$\begin{aligned} FL(x) = FL((\vec{\ell}, C, \vec{\ell'})) &= \emptyset \\ FL(\ell) &= \{\ell\} \\ FL(\lambda x.N) = FL(\mathsf{lift}\, N) = FL(\mathsf{force}\, N) = FL(\mathsf{box}_T\, N) &= FL(N) \\ FL(NP) = FL(\langle N, P \rangle) = FL(\mathsf{let}\, \langle x, y \rangle = N\, \mathsf{in}\, P) = FL(\mathsf{apply}(N, P)) &= FL(N) \cup FL(P). \end{aligned}$$

Notice how all of the labels occurring in a term are free. This is because labels are not bound by the constructs of the language, but rather by the underlying circuit. We will consider this aspect in more detail in the coming sections, and especially in Section 2.5.4.

Definition 2.2.2 (Free Variables)**.** *The set of* free variables *of a term M, denoted as $FV(M)$, is defined as follows:*

$$\begin{aligned} FV(\ell) = FV((\vec{\ell}, C, \vec{\ell'})) &= \emptyset \\ FV(x) &= \{x\} \\ FV(\lambda x.N) &= FV(N) \setminus \{x\} \\ FV(\mathsf{lift}\, N) = FV(\mathsf{force}\, N) = FV(\mathsf{box}_T\, N) &= FV(N) \\ FV(\mathsf{let}\, \langle x, y \rangle = N\, \mathsf{in}\, P) &= FV(N) \cup (FV(P) \setminus \{x, y\}) \\ FV(NP) = FV(\langle N, P \rangle) = FV(\mathsf{apply}(N, P)) &= FV(N) \cup FV(P). \end{aligned}$$

Definition 2.2.3 (Capture-avoiding Substitution)**.** *Let M and N be terms such that none of the variables occurring free in N occur in M, free or bound. We define the* substitution of N for x in M*, or $M[N/x]$, as follows:*

$$
\begin{aligned}
x[N/x] &= N \\
y[N/x] &= y \\
\ell[N/x] &= \ell \\
(\lambda x.L)[N/x] &= \lambda x.L \\
(\lambda y.L)[N/x] &= \lambda x.L[N/x] \\
(LP)[N/x] &= L[N/x]P[N/x] \\
\langle L, P\rangle[N/x] &= \langle L[N/x], P[N/x]\rangle \\
(\mathsf{let}\ \langle x, y\rangle = L\ \mathsf{in}\ P)[N/x] &= \mathsf{let}\ \langle x, y\rangle = L[N/x]\ \mathsf{in}\ P \\
(\mathsf{let}\ \langle y, x\rangle = L\ \mathsf{in}\ P)[N/x] &= \mathsf{let}\ \langle y, x\rangle = L[N/x]\ \mathsf{in}\ P \\
(\mathsf{let}\ \langle y, z\rangle = L\ \mathsf{in}\ P)[N/x] &= \mathsf{let}\ \langle y, z\rangle = L[N/x]\ \mathsf{in}\ P[N/x] \\
(\mathsf{lift}\ L)[N/x] &= \mathsf{lift}\ L[N/x] \\
(\mathsf{force}\ L)[N/x] &= \mathsf{force}\ L[N/x] \\
(\mathsf{box}_T\ L)[N/x] &= \mathsf{box}_T\ L[N/x] \\
(\mathsf{apply}(L, P))[N/x] &= \mathsf{apply}(L[N/x], P[N/x]) \\
(\vec{\ell}, C, \vec{\ell'})[N/x] &= (\vec{\ell}, C, \vec{\ell'}).
\end{aligned}
$$

2.3 Type System

As we mentioned in the previous section, Proto-Quipper-M is endowed with a linear type system that ensures that quantum states are never used more than once. In fact, there are two kinds of types in Proto-Quipper-M: *parameter types* and *linear types*. As the name suggests, parameter types refer to parameters, which are not subjected to linearity constraints and can be used any number of times. Any type that is not a parameter type is a linear type. A variable of linear type is also referred to as a *linear resource* and, once introduced, can (and must) be consumed exactly once. Among linear types, we distinguish the *simple M-types*, that is, the types of tuples of labels. For simplicity, we often refer to these as just *M-types*. Ultimately, types can be described by the following grammar:

$$\begin{array}{rll} \text{Types} & A, B & ::= \alpha \mid A \otimes B \mid A \multimap B \mid !A \mid \mathsf{Circ}(T, U), \\ \text{Parameter types} & P, R & ::= P \otimes R \mid !A \mid \mathsf{Circ}(T, U), \\ \text{Simple M-types} & T, U & ::= \alpha \mid T \otimes U, \end{array}$$

where α comes from the set $\mathcal{W}$ of wire types. We note that $A \multimap B$ is the type of linear abstractions from A to B, while $\mathsf{Circ}(T, U)$ is the type of circuits from M-type T to M-type U.

We now define the notion of *typing context*. In Proto-Quipper-M, a typing context can contain both parameter variables and linear variables. However, it is often useful to distinguish the case in which a typing context only contains parameter variables from the case in which it contains both kinds of variables. We therefore call a typing context a *parameter context*, and denote it by Φ, if it contains exclusively parameter types, whereas we call it a *generic context*, and denote it by Γ, if it contains parameter *and* linear types alike. Not that this distinction is in no way formal. In fact, a parameter variable may appear on one occasion in Φ and on another in Γ in two rule applications within the same type derivation. Whereas variables are assigned a type by a typing context, labels are assigned a type by the very label contexts that we saw in Section 2.1.2. If a generic context Γ and a label context Q turn M into a term of type A, then we write the following typing judgement:

$$\Gamma; Q \vdash M : A.$$

Typing judgements can be obtained by the following typing rules:

$$\frac{}{\Phi, x : A; \emptyset \vdash x : A}\, var \qquad \frac{}{\Phi; \ell : \alpha \vdash \ell : \alpha}\, labels$$

$$\frac{\Gamma, x : A; Q \vdash M : B}{\Gamma; Q \vdash \lambda x.M : A \multimap B}\, abs \qquad \frac{\Phi, \Gamma_1; Q_1 \vdash M : A \multimap B \quad \Phi, \Gamma_2; Q_2 \vdash N : A}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash MN : B}\, app$$

$$\frac{\Phi, \Gamma_1; Q_1 \vdash M : A \quad \Phi, \Gamma_2; Q_2 \vdash N : B}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash \langle M, N \rangle : A \otimes B}\, tuple$$

$$\frac{\Phi, \Gamma_1; Q_1 \vdash M : A \otimes B \quad \Phi, \Gamma_2, x : A, y : B; Q_2 \vdash N : C}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash \mathsf{let}\, \langle x, y \rangle = M\, \mathsf{in}\, N : C}\, let$$

$$\frac{\Phi; \emptyset \vdash M : A}{\Phi; \emptyset \vdash \mathsf{lift}\, M : !A}\, lift \qquad \frac{\Gamma; Q \vdash M : !A}{\Gamma; Q \vdash \mathsf{force}\, M : A}\, force \qquad \frac{\Gamma; Q \vdash M : !(T \multimap U)}{\Gamma; Q \vdash \mathsf{box}_T\, M : \mathsf{Circ}(T, U)}\, box$$

$$\frac{\Phi, \Gamma_1; Q_1 \vdash M : \mathsf{Circ}(T, U) \quad \Phi, \Gamma_2; Q_2 \vdash N : T}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash \mathsf{apply}(M, N) : U}\, apply$$

$$\frac{\emptyset; Q_1 \vdash \vec{\ell} : T \quad \emptyset; Q_2 \vdash \vec{\ell'} : U \quad C \in \mathbf{M}_{\mathcal{L}}(Q_1, Q_2)}{\Phi; \emptyset \vdash (\vec{\ell}, C, \vec{\ell'}) : \mathsf{Circ}(T, U)}\, circ$$

where we assume that Γ_1 and Γ_2 (as well as Q_1 and Q_2) are always disjoint and Γ_1, Γ_2 denotes the union of contexts Γ_1 and Γ_2. Note how the requirement that Γ_1 and Γ_2 be disjoint guarantees that a linear variable cannot be used more than once in a term, while the fact that the *var* rule successfully derives $\Phi, \Gamma; \emptyset \vdash x : A$ exclusively if x is the only linear variable in Γ guarantees that no linear variable goes unused. Together, these two principles guarantee that every linear variable is used *exactly* once, which is precisely the definition of *linearity*. It is easy to see that this kind of constraint holds for labels too, and in this case the linearity property coincides with the no-cloning property of quantum states. We now prove some preliminary results on type derivations and the relationship between types and terms.

Lemma 2.3.1 (Generation of Typing Judgements)**.** *The following hold:*

1. *If $\Phi, \Gamma; Q \vdash \ell : A$ then $\Gamma = \emptyset$ and there exists $\alpha \in \mathcal{W}$ such that $A \equiv \alpha$ and $Q = \ell : \alpha$.*
2. *If $\Gamma; Q \vdash \lambda x.M : C$ then there exist A and B such that $\Gamma, x : A; Q \vdash M : B$ and $C \equiv A \multimap B$.*
3. *If $\Phi, \Gamma; Q \vdash MN : B$ then there exists A, as well as Γ_1, Γ_2 and Q_1, Q_2, such that $\Phi, \Gamma_1; Q_1 \vdash M : A \multimap B$ and $\Phi, \Gamma_2; Q_2 \vdash N : A$, where $\Gamma_1, \Gamma_2 = \Gamma$ and $Q_1, Q_2 = Q$.*
4. *If $\Phi, \Gamma; Q \vdash \langle M, N \rangle : C$ then there exist A and B, as well as Γ_1, Γ_2 and Q_1, Q_2, such that $\Phi, \Gamma_1; Q_1 \vdash M : A$, $\Phi, \Gamma_2; Q_2 \vdash N : B$ and $C \equiv A \otimes B$, where $\Gamma_1, \Gamma_2 = \Gamma$ and $Q_1, Q_2 = Q$.*
5. *If $\Phi, \Gamma; Q \vdash \mathsf{let}\, \langle x, y \rangle = M \,\mathsf{in}\, N : C$ then there exist A and B, as well as Γ_1, Γ_2 and Q_1, Q_2, such that $\Phi, \Gamma_1; Q_1 \vdash M : A \otimes B$ and $\Phi, \Gamma_2, x : A, y : B; Q_2 \vdash N : C$, where $\Gamma_1, \Gamma_2 = \Gamma$ and $Q_1, Q_2 = Q$.*
6. *If $\Gamma; Q \vdash \mathsf{force}\, M : A$, then $\Gamma; Q \vdash M : !A$.*
7. *If $\Gamma; Q \vdash \mathsf{box}_T\, M : W$ then there exist T and U such that $\Gamma; Q \vdash M : !(T \multimap U)$ and $W \equiv \mathsf{Circ}(T, U)$.*
8. *If $\Phi, \Gamma; Q \vdash \mathsf{lift}\, M : C$ then $\Gamma = Q = \emptyset$ and there exists A such that $\Phi; \emptyset \vdash M : A$ and $C \equiv !A$*

9. *If* $\Phi, \Gamma; Q \vdash \mathsf{apply}(M, N) : C$ *then there exist* T *and* U*, as well as* Γ_1, Γ_2 *and* Q_1, Q_2*, such that* $\Phi, \Gamma_1; Q_1 \vdash M : \mathsf{Circ}(T, U)$*,* $\Phi, \Gamma_2; Q_2 \vdash N : T$ *and* $C \equiv U$*, where* $\Gamma_1, \Gamma_2 = \Gamma$ *and* $Q_1, Q_2 = Q$*.*

10. *If* $\Phi, \Gamma; Q \vdash (\vec{\ell}, C, \vec{\ell'}) : C$ *then* $\Gamma = Q = \emptyset$ *and there exist* T *and* U*, as well as* Q_1, Q_2*, such that* $\emptyset; Q_1 \vdash \vec{\ell} : T$*,* $\emptyset; Q_2 \vdash \vec{\ell'} : U$*,* $C \in \mathbf{M}_{\mathcal{L}}(Q_1, Q_2)$ *and* $C \equiv \mathsf{Circ}(T, U)$*.*

Proof. The claims all follow directly from the principle of inversion, since the rule system for types is syntax-directed. □

Lemma 2.3.2 (Generation of Values)**.** *Suppose* V *is a value. Then the following hold:*

1. *If* $\Phi, \Gamma; Q \vdash V : T$ *for some simple M-type* T*, then* $\Gamma = \emptyset$ *and* $V \equiv \vec{\ell}$ *for some* $\vec{\ell}$*.*

2. *If* $\Phi, \Gamma; Q \vdash V : A \multimap B$ *for some* A, B*, then* $V \equiv \lambda x.N$ *for some* x *and* N*.*

3. *If* $\Phi, \Gamma; Q \vdash V : A \otimes B$ *for some* A, B*, then* $V \equiv \langle V_1, V_2 \rangle$ *for some values* V_1, V_2*.*

4. *If* $\Phi, \Gamma; Q \vdash V : !A$ *for some* A*, then* $\Gamma = Q = \emptyset$ *and* $V \equiv \mathsf{lift}\, N$ *for some* N*.*

5. *If* $\Phi, \Gamma; Q \vdash V : \mathsf{Circ}(T, U)$ *for some simple M-types* T *and* U*, then* $\Gamma = Q = \emptyset$ *and* $V \equiv (\vec{\ell}, D, \vec{\ell'})$ *for some* $\vec{\ell}, \vec{\ell'}$ *and* D*.*

Proof. The claim follows immediately from the grammar for values and the rule system for types. □

Lemma 2.3.3 (Type of Values)**.** *Given a typing judgement* $\Phi, \Gamma, Q \vdash V : A$*, where* V *is a value, then either one of the following holds:*

- $\Gamma = Q = \emptyset$*.*
- A *is a linear type.*

Proof. By induction on the form of V:

- Case $V \equiv \ell$. In this case, by Lemma 2.3.1 we get $\Phi; \ell : \alpha \vdash \ell : \alpha$ and conclude that A is a linear type.
- Case $V \equiv \lambda x.N$. In this case, by Lemma 2.3.1 we get $\Phi, \Gamma; Q \vdash \lambda x.N : B \multimap C$ and conclude that A is a linear type.
- Case $V \equiv \langle V_1, V_2 \rangle$. In this case, by Lemma 2.3.1 we get $\Phi, \Gamma; Q \vdash \langle V_1, V_2 \rangle : B \otimes C$, $\Phi, \Gamma_1; Q_1 \vdash V_1 : B$ and $\Phi, \Gamma_2; Q_2 \vdash V_2 : C$, for some Γ_1, Γ_2 and Q_1, Q_2 such that $\Gamma_1, \Gamma_2 = \Gamma$ and $Q_1, Q_2 = Q$. By inductive hypothesis we know that either B is a linear type or $\Gamma_1 = Q_1 = \emptyset$. In the former case, we conclude that $B \otimes C$ is also a linear type. In the latter case, we know by inductive hypothesis that either C is a linear type or $\Gamma_2 = Q_2 = \emptyset$. In the former case, we conclude that $B \otimes C$ is also a linear type, while in the latter case we conclude that $\Gamma_1, \Gamma_2 = \Gamma = \emptyset$ and $Q_1, Q_2 = Q = \emptyset$.

- Case $V \equiv \mathsf{lift}\, N$. In this case, by Lemma 2.3.1 we get $\Phi; \emptyset \vdash \mathsf{lift}\, N : !B$ and conclude that $\Gamma = Q = \emptyset$.
- Case $V \equiv (\vec{\ell}, C, \vec{\ell'})$. In this case, by Lemma 2.3.1 we get $\Phi; \emptyset \vdash (\vec{\ell}, C, \vec{\ell'}) : \mathsf{Circ}(T, U)$ and conclude that $\Gamma = Q = \emptyset$.

□

Naturally, we have that substitution behaves well with respect to types. Specifically, by substituting a value of type A for a variable of type A in a term M, we do not alter the type of M. We prove this separately for parameter types and linear types at first and then join the two results.

Lemma 2.3.4 (Parameter Substitution)**.** *Let $\Phi = \Phi', x : R$. If $\Phi, \Gamma; Q \vdash M : B$ and $\Phi'; \emptyset \vdash V : R$, where V is a value, then $\Phi', \Gamma; Q \vdash M[V/x] : B$.*

Proof. By induction on the derivation of $\Phi, \Gamma; Q \vdash M : B$.

- Case of *var*. Suppose $M \equiv y$. If $x \not\equiv y$, we have $y[V/x] = y$ and the claim is trivially true. Otherwise, if $x \equiv y$, then $y[V/x] = x[V/x] = V$ and $\Phi'; \emptyset \vdash V : R$ by hypothesis.
- Case of *labels*. Suppose $M \equiv \vec{\ell}$. In this case $\vec{\ell}[V/x] = \ell'$ and the claim is trivially true.
- Case of *abs*. Suppose $M \equiv \lambda y.N$. By Lemma 2.3.1 we know that $\Phi, \Gamma, y : C; Q \vdash N : D$ for some C and D such that $B \equiv C \multimap D$. If $x \equiv y$ then by the definition of capture-free substitution we have $(\lambda x.N)[V/x] = \lambda x.N$ and the claim is trivially true. Otherwise, if $x \not\equiv y$, we have $(\lambda y.N)[V/x] = \lambda y.(N[V/x])$. In this case, by inductive hypothesis we get $\Phi', \Gamma, y : C; Q \vdash N[V/x] : D$ and conclude $\Phi', \Gamma; Q \vdash \lambda y.(N[V/x]) : C \multimap D$ by the *abs* rule.
- Case of *app*. Suppose $M \equiv NP$. In this case, $(NP)[V/x] = (N[V/x])(P[V/x])$. By Lemma 2.3.1 we know that $\Phi, \Gamma_1; Q_1 \vdash N : C \multimap B$ and $\Phi, \Gamma_2; Q_2 \vdash P : C$, for some C and for $\Gamma_1, \Gamma_2, Q_1, Q_2$ such that $\Gamma = \Gamma_1, \Gamma_2$ and $Q = Q_1, Q_2$. By inductive hypothesis we get $\Phi', \Gamma_1; Q_1 \vdash N[V/x] : C \multimap B$ and $\Phi', \Gamma_2; Q_2 \vdash P[V/x] : C$ and conclude $\Phi', \Gamma; Q \vdash (N[V/x])(P[V/x]) : B$ by the *app* rule.
- Case of *tuple*. Suppose $M \equiv \langle N, P \rangle$. In this case, $\langle NP \rangle[V/x] = \langle N[V/x], P[V/x] \rangle$. By Lemma 2.3.1 we know that $\Phi, \Gamma_1; Q_1 \vdash N : C$ and $\Phi, \Gamma_2; Q_2 \vdash P : D$, for some C and D such that $B \equiv C \otimes D$ and for $\Gamma_1, \Gamma_2, Q_1, Q_2$ such that $\Gamma = \Gamma_1, \Gamma_2$ and $Q = Q_1, Q_2$. By inductive hypothesis we get $\Phi', \Gamma_1; Q_1 \vdash N[V/x] : C$ and $\Phi', \Gamma_2; Q_2 \vdash P[V/x] : D$ and conclude $\Phi', \Gamma; Q \vdash \langle N[V/x], P[V/x] \rangle : C \otimes D$ by the *tuple* rule.
- Case of *let*. Suppose $M \equiv \mathsf{let}\, \langle y, z \rangle = N \,\mathsf{in}\, P$. By Lemma 2.3.1 we know that $\Phi, \Gamma_1; Q_1 \vdash N : C \otimes D$ and $\Phi, \Gamma_2, y : C, z : D; Q_2 \vdash P : B$, for some C and D and for

$\Gamma_1, \Gamma_2, Q_1, Q_2$ such that $\Gamma = \Gamma_1, \Gamma_2$ and $Q = Q_1, Q_2$. If $x \equiv y$ or $x \equiv z$ then by the definition of capture-free substitution we have $(\mathsf{let}\,\langle y, z\rangle = N\,\mathsf{in}\,P)[V/x] = \mathsf{let}\,\langle y, z\rangle = N[V/x]\,\mathsf{in}\,P$. In this case, by inductive hypothesis we get $\Phi', \Gamma_1; Q_1 \vdash N[V/x] : C \otimes D$ and conclude $\Phi', \Gamma; Q \vdash \mathsf{let}\,\langle y, z\rangle = N[V/x]\,\mathsf{in}\,P : B$ by the *tuple* rule. Otherwise, if $x \not\equiv y$ and $x \not\equiv z$, we have $(\mathsf{let}\,\langle y, z\rangle = N\,\mathsf{in}\,P)[V/x] = \mathsf{let}\,\langle y, z\rangle = N[V/x]\,\mathsf{in}\,P[V/x]$. In this case, by inductive hypothesis we get both $\Phi', \Gamma_1; Q_1 \vdash N[V/x] : C \otimes D$ and $\Phi', \Gamma_2, y : C, z : D; Q_2 \vdash P[V/x] : B$ and conclude $\Phi', \Gamma; Q \vdash \mathsf{let}\,\langle y, z\rangle = N[V/x]\,\mathsf{in}\,P[V/x] : B$ by the *let* rule.

- Case of *lift*. Suppose $M \equiv \mathsf{lift}\,N$. In this case, $(\mathsf{lift}\,N)[V/x] = \mathsf{lift}(N[V/x])$ and $\Gamma = Q = \emptyset$ by Lemma 2.3.1. By the same lemma we know that $\Phi; \emptyset \vdash N : C$ for some C such that $B \equiv !C$. By inductive hypothesis we get $\Phi'; \emptyset \vdash N[V/x] : C$ and conclude $\Phi', \emptyset; Q' \vdash \mathsf{lift}(N[V/x]) : !C$ by the *lift* rule.

- Case of *force*. Suppose $M \equiv \mathsf{force}\,N$. In this case, $(\mathsf{force}\,N)[V/x] = \mathsf{force}(N[V/x])$. By Lemma 2.3.1 we know that $\Phi, \Gamma; Q \vdash N : !B$. By inductive hypothesis we get $\Phi', \Gamma; Q \vdash N[V/x] : !B$ and conclude $\Phi', \Gamma; Q \vdash \mathsf{force}(N[V/x]) : B$ by the *force* rule.

- Case of *box*. Suppose $M \equiv \mathsf{box}_T\,N$. In this case, $(\mathsf{box}_T\,N)[V/x] = \mathsf{box}_T(N[V/x])$. By Lemma 2.3.1 we know that $\Phi, \Gamma; Q \vdash N : !(T \multimap U)$ for some T, U such that $B \equiv \mathsf{Circ}(T, U)$. By inductive hypothesis we get $\Phi', \Gamma; Q \vdash N[V/x] : !(T \multimap U)$ and conclude $\Phi', \Gamma; Q \vdash \mathsf{box}_T(N[V/x]) : \mathsf{Circ}(T, U)$ by the *box* rule.

- Case of *apply*. Suppose $M \equiv \mathsf{apply}(N, P)$. In this case, $\mathsf{apply}(N, P)[V/x] = \mathsf{apply}(N[V/x], P[V/x])$. By Lemma 2.3.1 we know that $\Phi, \Gamma_1; Q_1 \vdash N : \mathsf{Circ}(T, U)$ and $\Phi, \Gamma_2; Q_2 \vdash P : T$, for some T, U such that $B \equiv U$ and for $Q_1, Q_2, \Gamma_1, \Gamma_2$ such that $B \equiv U$ and $\Gamma = \Gamma_1, \Gamma_2$ and $Q = Q_1, Q_2$. By inductive hypothesis we get $\Phi', \Gamma_1; Q_1 \vdash N[V/x] : \mathsf{Circ}(T, U)$ and $\Phi', \Gamma_2; Q_2 \vdash P[V/x] : T$ and conclude $\Phi', \Gamma; Q \vdash \mathsf{apply}(N[V/x], P[V/x]) : U$ by the *apply* rule.

- Case of *circ*. Suppose $M \equiv (\vec{\ell}, D, \vec{\ell'})$. In this case $(\vec{\ell}, D, \vec{\ell'})[N/x] = (\vec{\ell}, D, \vec{\ell'})$ and the claim is trivially true.

□

Lemma 2.3.5 (Linear Substitution)**.** *If $\Phi, \Gamma, x : A; Q \vdash M : B$ and $\Phi, \Gamma'; Q' \vdash V : A$, where A is a linear type and V is a value, then $\Phi, \Gamma, \Gamma'; Q, Q' \vdash M[V/x] : B$.*

Proof. By induction on the derivation of $\Phi, \Gamma, x : A; Q \vdash M : B$.

- Case of *var*. Suppose $M \equiv y$. Here necessarily $y \equiv x$ (i.e. x occurs free exactly once in M), since Γ contains exclusively x and therefore cannot possibly assign a type to any $y \not\equiv x$. In this case $x[V/x] = V$ and $\Phi, \Gamma'; Q' \vdash V : A$ by hypothesis.

- Case of *labels*. This case is impossible since it would entail $\Gamma = \emptyset$ by Lemma 2.3.1.

- Case of *abs*. Suppose $M \equiv \lambda y.N$. In this case, $(\lambda y.N)[V/x] = \lambda y.(L[V/x])$. By Lemma 2.3.1 we know that $\Phi, \Gamma, y : C; Q \vdash N : D$, for some C, D such that $B \equiv C \multimap D$. By inductive hypothesis we get $\Phi, \Gamma, y : C, \Gamma'; Q, Q' \vdash N[V/x] : D$ and conclude $\Phi, \Gamma, \Gamma'; Q, Q' \vdash \lambda y.(N[V/x]) : C \multimap D$ by the *abs* rule.

- Case of *app*. Suppose $M \equiv NP$. By Lemma 2.3.1 we know that $\Phi, \Gamma_1; Q_1 \vdash N : C \multimap B$ and $\Phi, \Gamma_2; Q_2 \vdash P : C$, for some C and for $\Gamma_1, \Gamma_2, Q_1, Q_2$ such that $\Gamma, x : A = \Gamma_1, \Gamma_2$ and $Q = Q_1, Q_2$. Because Γ_1 and Γ_2 are disjoint, we have either $x \in \Gamma_1$ or $x \in \Gamma_2$. Let us assume, without loss of generality, that $x \in \Gamma_1$ and thus $\Gamma_1 = \Gamma_1', x : A$ for some Γ_1'. In this case, $(NP)[V/x] = (N[V/x])P$. By the results of Lemma 2.3.1 and the inductive hypothesis we get $\Phi, \Gamma_1', \Gamma'; Q_1, Q' \vdash N[V/x] : C \multimap B$ and conclude $\Phi, \Gamma, \Gamma'; Q, Q' \vdash (N[V/x])P : B$ by the *app* rule.

- Case of *tuple*. Suppose $M \equiv \langle N, P \rangle$. By Lemma 2.3.1 we know that $\Phi, \Gamma_1; Q_1 \vdash N : C$ and $\Phi, \Gamma_2; Q_2 \vdash P : D$, for some C and D such that $B \equiv C \otimes D$ and for $\Gamma_1, \Gamma_2, Q_1, Q_2$ such that $\Gamma = \Gamma_1, \Gamma_2$ and $Q = Q_1, Q_2$. Because Γ_1 and Γ_2 are disjoint, we have either $x \in \Gamma_1$ or $x \in \Gamma_2$. Let us assume, without loss of generality, that $x \in \Gamma_1$ and thus $\Gamma_1 = \Gamma_1', x : A$ for some Γ_1'. In this case, $\langle NP \rangle[V/x] = \langle N[V/x], P \rangle$. By inductive hypothesis we get $\Phi, \Gamma_1', \Gamma'; Q_1, Q' \vdash N[V/x] : C$ and conclude $\Phi, \Gamma, \Gamma'; Q, Q' \vdash \langle N[V/x], P \rangle : C \otimes D$ by the *tuple* rule.

- Case of *let*. Suppose $M \equiv \mathsf{let}\, \langle y, z \rangle = N \,\mathsf{in}\, P$. By Lemma 2.3.1 we know that $\Phi, \Gamma_1; Q_1 \vdash N : C \otimes D$ and $\Phi, \Gamma_2, y : C, z : D; Q_2 \vdash P : B$, for some C and D and for $\Gamma_1, \Gamma_2, Q_1, Q_2$ such that $\Gamma = \Gamma_1, \Gamma_2$ and $Q = Q_1, Q_2$. Because Γ_1 and Γ_2 are disjoint, we have either $x \in \Gamma_1$ or $x \in \Gamma_2$. Let us assume, without loss of generality, that $x \in \Gamma_1$ and thus $\Gamma_1 = \Gamma_1', x : A$ for some Γ_1'. In this case, $(\mathsf{let}\, \langle y, z \rangle = N \,\mathsf{in}\, P)[V/x] = \mathsf{let}\, \langle y, z \rangle = N[V/x] \,\mathsf{in}\, P$. By inductive hypothesis we get $\Phi, \Gamma_1', \Gamma'; Q_1, Q' \vdash N[V/x] : C \otimes D$ and conclude $\Phi, \Gamma, \Gamma'; Q, Q' \vdash \mathsf{let}\, \langle y, z \rangle = N[V/x] \,\mathsf{in}\, P : B$ by the *let* rule.

- Case of *lift*. This case is impossible since it would entail $\Gamma = \emptyset$ by Lemma 2.3.1.

- Case of *force*. Suppose $M \equiv \mathsf{force}\, N$. In this case, $(\mathsf{force}\, N)[V/x] = \mathsf{force}(N[V/x])$. By Lemma 2.3.1 we know that $\Phi, \Gamma, x : A; Q \vdash N : !B$. By inductive hypothesis we get $\Phi, \Gamma, \Gamma'; Q, Q' \vdash N[V/x] : !B$ and conclude $\Phi, \Gamma, \Gamma'; Q, Q' \vdash \mathsf{force}(N[V/x]) : B$

- Case of *box*. Suppose $M \equiv \mathsf{box}_T\, N$. In this case, $(\mathsf{box}_T\, N)[V/x] = \mathsf{box}_T(N[V/x])$. By Lemma 2.3.1 we know that $\Phi, \Gamma, x : A; Q \vdash N : !(T \multimap U)$ for some T, U such that $B \equiv \mathsf{Circ}(T, U)$. By inductive hypothesis we get $\Phi, \Gamma, \Gamma'; Q, Q' \vdash N[V/x] : !(T \multimap U)$ and conclude $\Phi, \Gamma, \Gamma'; Q, Q' \vdash \mathsf{box}_T(N[V/x]) : \mathsf{Circ}(T, U)$ by the *box* rule.

- Case of *apply*. Suppose $M \equiv \mathsf{apply}(N, P)$. By Lemma 2.3.1 we know that $\Phi, \Gamma_1, x : A; Q_1 \vdash N : \mathsf{Circ}(T, U)$ and $\Phi, \Gamma_2; Q_2 \vdash P : T$, for some T, U such that $B \equiv U$

and for $\Gamma_1, \Gamma_2, Q_1, Q_2$ such that $\Gamma, x : A = \Gamma_1, \Gamma_2$ and $Q = Q_1, Q_2$. Because Γ_1 and Γ_2 are disjoint, we have either $x \in \Gamma_1$ or $x \in \Gamma_2$. Let us assume, without loss of generality, that $x \in \Gamma_1$ and thus $\Gamma_1 = \Gamma_1', x : A$ for some Γ_1'. In this case, $\mathsf{apply}(N, P)[V/x] = \mathsf{apply}(N[V/x], P)$. By the results of Lemma 2.3.1 and the inductive hypothesis we get $\Phi, \Gamma_1', \Gamma'; Q_1, Q' \vdash N[V/x] : \mathsf{Circ}(T, U)$ and conclude $\Phi, \Gamma, \Gamma'; Q, Q' \vdash \mathsf{apply}(N[V/x], P) : U$ by the *apply* rule.

- Case of *circ*. This case is impossible since it would entail $\Gamma = \emptyset$ by Lemma 2.3.1.

□

Theorem 2.3.6 (Substitution)**.** *If $\Phi, \Gamma, x : A; Q \vdash M : B$ and $\Phi, \Gamma'; Q' \vdash V : B$, where V is a value, then*

$$\Phi, \Gamma, \Gamma'; Q, Q' \vdash M[V/x] : B.$$

Proof. The claim follows immediately from Lemma 2.3.3 and lemmata 2.3.4 and 2.3.5. □

2.4 Big-step Operational Semantics

In this section we review the operational semantics given by Rios and Selinger for Proto-Quipper-M, which, as the title suggests, are big-step. As we mentioned earlier, the evaluation of a Proto-Quipper-M program is intimately related to the circuit that the program is designed to build. Because of this, the operational semantics of the language is not defined on terms alone, but rather jointly on terms and circuits. To this effect, we give the definition of *configuration*.

Definition 2.4.1 (Configuration)**.** *A* configuration *is a pair (C, M), where C is a circuit and M is a term.*

Intuitively, C is the circuit being built as a side-effect of the evaluation of M. We now proceed to give the following definitions and functions, which will be essential throughout the rest of this thesis.

Definition 2.4.2 (Equivalent Circuit)**.** *Let $C : Q_1 \to Q_1'$ and $D : Q_2 \to Q_2'$ be two labelled circuits and let $(\vec{\ell}, C, \vec{\ell'})$ and $(\vec{k}, D, \vec{k'})$ be the corresponding boxed circuits. We say that $(\vec{\ell}, C, \vec{\ell'})$ and $(\vec{k}, D, \vec{k'})$ are* equivalent *and we write $(\vec{\ell}, C, \vec{\ell'}) \cong (\vec{k}, D, \vec{k'})$ when they only differ by a renaming of labels, that is when $C = D$ in $\mathbf{M}$.*

Definition 2.4.3 (freshlabels)**.** *Given a term M and a simple M-type T, we define the function* freshlabels *as follows:*

$$\mathsf{freshlabels}(M, T) = (Q, \vec{\ell}),$$

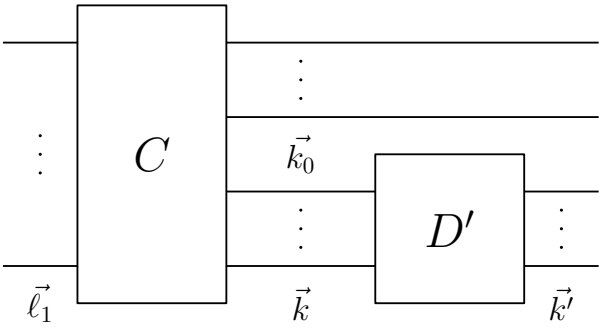
such that the labels in $\vec{\ell}$ do not occur in M and $\emptyset; Q \vdash \vec{\ell} : T$.

Definition 2.4.4 (append)**.** *Let $C : Q_1 \to Q_1'$ and $D : Q_2 \to Q_2'$ be two labelled circuits and let $(\vec{\ell}_1, C, \vec{\ell}_1')$ and $(\vec{\ell}_2, D, \vec{\ell}_2')$ be the corresponding boxed circuits. Let $\vec{k}$ be a subset of the labels which occur in $\vec{\ell}_1'$. We define the function* append *as follows:*

$$\mathsf{append}(C, \vec{k}, \vec{\ell}_2, D, \vec{\ell}_2') = (C', \vec{k'}),$$

where C' is the circuit obtained by attaching the inputs of D' to the matching outputs of C, for $(\vec{k}, D', \vec{k'}) \cong (\vec{\ell}_2, D, \vec{\ell}_2')$. More formally, assume, without loss of generality, that Q_1' is the concatenation of Q_{11}' and Q_{12}', where Q_{12}' contains all and only the labels in $\vec{k}$. Then we have

$$C' = C \circ (id_{Q_{11}'} \otimes D').$$



Now we have all the prerequisites for the definition of an operational semantics. We define $\Downarrow$ as a binary relation over configurations. Informally, $(C, M) \Downarrow (D, V)$ means that the evaluation of M with an underlying circuit C eventually results in value V and in the construction of circuit D.

$$\frac{}{(C, x) \Downarrow \text{Error}} \qquad \frac{}{(C, \vec{\ell}) \Downarrow (C, \vec{\ell})} \qquad \frac{}{(C, \lambda x.M) \Downarrow (C, \lambda x.M)}$$

$$\frac{(C, M) \Downarrow (C_1, \lambda x.P) \quad (C_1, N) \Downarrow (C_2, V) \quad (C_2, P[V/x]) \Downarrow (C_3, W)}{(C, MN) \Downarrow (C_3, W)}$$

$$\frac{(C, M) \Downarrow \text{Otherwise}}{(C, MN) \Downarrow \text{Error}} \qquad \frac{(C, M) \Downarrow (C_1, \langle V_1, V_2 \rangle) \quad (C_1, N[V_1/x][V_2/y]) \Downarrow (C_3, W)}{(C, \mathsf{let}\ \langle x, y \rangle = M\ \mathsf{in}\ N) \Downarrow (C_3, W)}$$

$$\frac{(C, M) \Downarrow \text{Otherwise}}{(C, \mathsf{let}\ \langle x, y \rangle = M\ \mathsf{in}\ N) \Downarrow \text{Error}} \qquad \frac{(C, M) \Downarrow (C_1, V) \quad (C_1, N) \Downarrow (C_2, W)}{(C, \langle M, N \rangle) \Downarrow (C_2, \langle V, W \rangle)}$$

$$\frac{}{(C, \mathsf{lift}\ M) \Downarrow (C, \mathsf{lift}\ M)} \qquad \frac{(C, M) \Downarrow (C_1, \mathsf{lift}\ N) \quad (C_1, N) \Downarrow (C_2, V)}{(C, \mathsf{force}\ M) \Downarrow (C_2, V)}$$

$$\frac{(C, M) \Downarrow \text{Otherwise}}{(C, \mathsf{force}\, M) \Downarrow \text{Error}}$$

$$\frac{(C, M) \Downarrow (C_1, \mathsf{lift}\, N) \quad (Q, \vec{\ell}) = \mathsf{freshlabels}(N, T) \quad (id_Q, N\vec{\ell}) \Downarrow (D, \vec{\ell'})}{(C, \mathsf{box}_T\, M) \Downarrow (C_1, (\vec{\ell}, D, \vec{\ell'}))}$$

$$\frac{(C, M) \Downarrow \text{Otherwise}}{(C, \mathsf{box}_T\, M) \Downarrow \text{Error}} \qquad \frac{(C, M) \Downarrow \text{Otherwise}}{(C, \mathsf{apply}(M, N)) \Downarrow \text{Error}} \qquad \frac{}{(C, (\vec{\ell}, D, \vec{\ell'})) \Downarrow (C, (\vec{\ell}, D, \vec{\ell'}))}$$

$$\frac{(C, M) \Downarrow (C_1, (\vec{\ell}, D, \vec{\ell'})) \quad (C_1, N) \Downarrow (C_2, \vec{k}) \quad (C_3, \vec{k'}) = \mathsf{append}(C_2, \vec{k}, \vec{\ell}, D, \vec{\ell'})}{(C, \mathsf{apply}(M, N)) \Downarrow (C_3, \vec{k'})}$$

where the notation "$(C, M) \Downarrow$ Otherwise" is shorthand for $(C, M) \Downarrow (D, V)$ where V does not match the explicit form required by any other rule that evaluates the same configuration.

2.5 Small-step Operational Semantics

It is now time to take the first step towards a machine semantics for Proto-Quipper-M. In this section, we extrapolate an equivalent small-step semantics from the big-step semantics that we just saw, and we examine its properties and its limitations. For the sake of simplicity, from now on we will assume that the terms in the configurations we work with do not contain free variables.

Definition 2.5.1 (Small-step Configuration)**.** *A* small-step configuration *is a pair of the form (C, M), where C is a circuit and M is a term with no free variables.*

We then define a binary reduction relation $\to$ on configurations. Informally, $(C, M) \to (D, N)$ means that M evaluates to N in one step, in a way that updates the underlying circuit from C to D. In order to mimic the behavior of the big-step semantics, we design two sets of rules. The rules in the first set each resemble one of the main rules of the original big-step semantics and operate directly on redexes.

$$\frac{}{(C, (\lambda x.M)V) \to (C, M[V/x])}\,\beta\text{-reduction}$$

$$\frac{}{(C, \mathsf{let}\, \langle x, y\rangle = \langle V, W\rangle\, \mathsf{in}\, M) \to (C, M[V/x][W/y])}\,let$$

$$\frac{}{(C, \mathsf{force}(\mathsf{lift}\, M)) \to (C, M)}\,force$$

$$\frac{(Q, \vec{\ell}) = \mathsf{freshlabels}(M, T) \quad (id_Q, M\vec{\ell}) \to \ldots \to (D, \vec{\ell'})}{(C, \mathsf{box}_T(\mathsf{lift}\, M)) \to (C, (\vec{\ell}, D, \vec{\ell'}))}\, box$$

$$\frac{(C', \vec{k'}) = \mathsf{append}(C, \vec{k}, \vec{\ell}, D, \vec{\ell'})}{(C, \mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})) \to (C', \vec{k'})}\, apply$$

where the notation $(C, M) \to \ldots \to (C', M')$ is shorthand for

$$\begin{aligned}(C, M) \equiv (C_1, M_1), (C_1, M_1) \to (C_2, M_2), \ldots,\\ (C_{n-1}, M_{n-1}) \to (C_n, M_n), (C_n, M_n) \equiv (C', M'),\end{aligned}$$

for some $n > 0$. That is, a reduction sequence of finite length from (C, M) to (C', M'). We employ a finite, but arbitrary number of premises instead of a single premise with the transitive and reflexive closure $\to^*$ of the reduction relation $\to$ in order to make future proofs by induction easier. We now examine the second set of rules, which we call *contextual rules*. Each one of these rules allows for the intermediate evaluation of an immediate sub-term within a term.

$$\frac{(C, M) \to (C', M')}{(C, MN) \to (C', M'N)}\, ctx\text{-}app\text{-}left \qquad \frac{(C, M) \to (C', M')}{(C, VM) \to (C', VM')}\, ctx\text{-}app\text{-}right$$

$$\frac{(C, M) \to (C', M')}{(C, \langle M, N\rangle) \to (C', \langle M', N\rangle)}\, ctx\text{-}tuple\text{-}left \qquad \frac{(C, M) \to (C', M')}{(C, \langle V, M\rangle) \to (C', \langle V, M'\rangle)}\, ctx\text{-}tuple\text{-}right$$

$$\frac{(C, M) \to (C', M')}{(C, \mathsf{let}\, \langle x, y\rangle = M\, \mathsf{in}\, N) \to (C', \mathsf{let}\, \langle x, y\rangle = M'\, \mathsf{in}\, N)}\, ctx\text{-}let$$

$$\frac{(C, M) \to (C', M')}{(C, \mathsf{force}\, M) \to (C', \mathsf{force}\, M')}\, ctx\text{-}force \qquad \frac{(C, M) \to (C', M')}{(C, \mathsf{box}_T\, M) \to (C', \mathsf{box}_T\, M')}\, ctx\text{-}box$$

$$\frac{(C, M) \to (C', M')}{(C, \mathsf{apply}(M, N)) \to (C', \mathsf{apply}(M', N))}\, ctx\text{-}apply\text{-}left$$

$$\frac{(C, M) \to (C', M')}{(C, \mathsf{apply}(V, M)) \to (C', \mathsf{apply}(V, M'))}\, ctx\text{-}apply\text{-}right$$

The way we introduced it, the reduction relation $\to$ is deterministic, as we prove in the following results.

Lemma 2.5.1. *Every small-step configuration (C, M) can be reduced by at most one rule of the small-step operational semantics.*

Proof. We proceed by cases on M:

- Case $M \equiv x$. This case is impossible, since by the definition of small-step configuration M must contain no free variables.
- Case $M \equiv \vec{\ell}$. In this case $\vec{\ell}$ is a value and $(C, \vec{\ell})$ cannot be reduced by any rule.
- Case $M \equiv \lambda x.N$. In this case $\lambda x.N$ is a value and $(C, \lambda x.N)$ cannot be reduced by any rule.
- Case $M \equiv NP$. In this case NP is matched by the *β-reduction*, *ctx-app-left* and *ctx-app-right* rules. However, *β-reduction* requires that both N and P be values to be applied, while *ctx-app-left* requires that N be reducible (and therefore not a value) and *ctx-app-right* requires that N be a value and P be reducible (and therefore not a value). Because these conditions mutually exclude each other, we conclude that (C, NP) can be reduced by at most one rule.
- Case $M \equiv \langle N, P \rangle$. In this case $\langle N, P \rangle$ is matched by the *ctx-tuple-left* and *ctx-tuple-right* rules. However, the latter requires that N be a value to be applied, while the former requires that it be reducible (and therefore not a value). Because these conditions mutually exclude each other, we conclude that $(C, \langle N, P \rangle)$ can be reduced by at most one rule.
- Case $M \equiv \mathsf{let}\, \langle x, y \rangle = N \,\mathsf{in}\, P$. In this case $\mathsf{let}\, \langle x, y \rangle = N \,\mathsf{in}\, P$ is matched by the *let* and *ctx-let* rules. However, the former requires that N be a value to be applied, while the latter requires that it be reducible (and therefore not a value). Because these conditions mutually exclude each other, we conclude that $(C, \mathsf{let}\, \langle x, y \rangle = N \,\mathsf{in}\, P)$ can be reduced by at most one rule.
- Case $M \equiv \mathsf{lift}\, N$. In this case $\mathsf{lift}\, N$ is a value and $(C, \mathsf{lift}\, N)$ cannot be reduced by any rule.
- Case $M \equiv \mathsf{force}\, N$. In this case $\mathsf{force}\, N$ is matched by the *force* and *ctx-force* rules. However, the former requires that N be a value to be applied, while the latter requires that it be reducible (and therefore not a value). Because these conditions mutually exclude each other, we conclude that $(C, \mathsf{force}\, N)$ can be reduced by at most one rule.

- Case $M \equiv \mathsf{box}_T\, N$. In this case $\mathsf{box}_T\, N$ is matched by the *box* and *ctx-box* rules. However, the former requires that N be a value to be applied, while the latter requires that it be reducible (and therefore not a value). Because these conditions mutually exclude each other, we conclude that $(C, \mathsf{box}_T\, N)$ can be reduced by at most one rule.
- Case $M \equiv \mathsf{apply}(N, P)$. In this case $\mathsf{apply}(N, P)$ is matched by the *apply*, *ctx-apply-left* and *ctx-apply-right* rules. However, *apply* requires that both N and P be values to be applied, while *ctx-apply-left* requires that N be reducible (and therefore not a value) and *ctx-apply-right* requires that N be a value and P be reducible (and therefore not a value). Because these conditions mutually exclude each other, we conclude that $(C, \mathsf{apply}(N, P))$ can be reduced by at most one rule.
- Case $M \equiv (\vec{\ell}, D, \vec{\ell'})$. In this case $(\vec{\ell}, D, \vec{\ell'}$ is a value and $(C, (\vec{\ell}, D, \vec{\ell'})$ cannot be reduced by any rule.

□

Proposition 2.5.2 (Determinism of Small-step Semantics)**.** *The reduction relation* $\to$ *is deterministic. That is, if* $(C, M) \to (D, N)$*, then for every configuration* (D', N') *such that* $(C, M) \to (D', N')$ *we have* $D = D'$ *and* $N \equiv N'$*.*

Proof. We already known by Lemma 2.5.1 that at most one rule can be applied to reduce any given configuration. What is left to do is prove that each rule is deterministic by itself. The proof is trivial by induction on the derivation of $(C, M) \to (D, N)$. □

2.5.1 Evaluation Contexts

The contextual rules are defined in a recursive fashion, which means that redexes can be reduced at an arbitrary depth within a term through multiple rule applications. In order to be able to reason about all the valid positions where a reduction may occur within a term, we introduce the notion of *evaluation context*.

Definition 2.5.2 (Evaluation Context)**.** *An evaluation context defines where, within a term, we can reduce a sub-term. Formally, an evaluation context is a function defined by the following grammar:*

$$\begin{aligned} E, F ::= {} & [\cdot] \mid EM \mid VE \mid \langle E, M\rangle \mid \langle V, E\rangle \mid \mathsf{let}\, \langle x, y\rangle = E\, \mathsf{in}\, N \\ & \mid \mathsf{force}\, E \mid \mathsf{box}_T\, E \mid \mathsf{apply}(E, M) \mid \mathsf{apply}(V, E), \end{aligned}$$

and the following semantics:

$$
\begin{aligned}
[\cdot][M] &= M \\
(EN)[M] &= (E[M])N \\
(VE)[M] &= V(E[M]) \\
\langle E, N\rangle[M] &= \langle E[M], N\rangle \\
\langle V, E\rangle[M] &= \langle V, E[M]\rangle \\
(\mathsf{let}\,\langle x, y\rangle = E\,\mathsf{in}\,N)[M] &= \mathsf{let}\,\langle x, y\rangle = E[M]\,\mathsf{in}\,N \\
(\mathsf{force}\,E)[M] &= \mathsf{force}\,E[M] \\
(\mathsf{box}_T\,E)[M] &= \mathsf{box}_T\,E[M] \\
\mathsf{apply}(E, N)[M] &= \mathsf{apply}(E[M], N) \\
\mathsf{apply}(V, E)[M] &= \mathsf{apply}(V, E[M]).
\end{aligned}
$$

An evaluation context is de-facto a function from terms to terms. However, it is perhaps more intuitively to understand an evaluation context as an incomplete term with a hole in which we can stick different sub-terms. The way the grammar for evaluation contexts is designed guarantees that whenever a term is reducible on its own, then it is also reducible when we stick it in an evaluation context, and vice-versa. We prove this fundamental property of evaluation contexts in the following theorem.

Theorem 2.5.3 (Fundamental Theorem of Evaluation Contexts)**.** *For every evaluation context E, we have that $(C, M) \to (C', M')$ if and only if $(C, E[M]) \to (C', E[M'])$.*

Proof. We first prove that if $(C, M) \to (C', M')$, then $(C, E[M]) \to (C', E[M'])$. We proceed by induction on the form of E:

- Case $E \equiv [\cdot]$. In this case $E[M] \equiv M$ and the claim is trivially true.
- Case $E \equiv FN$. In this case $E[M] \equiv (F[M])N$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, (F[M])N) \to (C', (F[M'])N)$ by the *ctx-app-left* rule.
- Case $E \equiv VF$. In this case $E[M] \equiv V(F[M])$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, V(F[M])) \to (C', V(F[M']))$ by the *ctx-app-right* rule.
- Case $E \equiv \langle F, N\rangle$. In this case $E[M] \equiv \langle F[M], N\rangle$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, \langle F[M], N\rangle) \to (C', \langle F[M'], N\rangle)$ by the *ctx-tuple-left* rule.
- Case $E \equiv \langle V, F\rangle$. In this case $E[M] \equiv \langle V, F[M]\rangle$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, \langle V, F[M]\rangle) \to (C', \langle V, F[M]\rangle)$ by the *ctx-tuple-right* rule.

- Case $E \equiv \mathsf{let}\ \langle x, y\rangle = F\ \mathsf{in}\ N$. In this case $E[M] = \mathsf{let}\ \langle x, y\rangle = F[M]\ \mathsf{in}\ N$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, \mathsf{let}\ \langle x, y\rangle = F[M]\ \mathsf{in}\ N) \to (C', \mathsf{let}\ \langle x, y\rangle = F[M']\ \mathsf{in}\ N)$ by the *ctx-let* rule.

- Case $E \equiv \mathsf{force}\, F$. In this case $E[M] = \mathsf{force}\, F[M]$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, \mathsf{force}\, F[M]) \to (C', \mathsf{force}\, F[M'])$ by the *ctx-force* rule.

- Case $E \equiv \mathsf{box}_T\, F$. In this case $E[M] = \mathsf{box}_T\, F[M]$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, \mathsf{box}_T\, F[M]) \to (C', \mathsf{box}_T\, F[M'])$ by the *ctx-box* rule.

- Case $E \equiv \mathsf{apply}(F, N)$. In this case $E[M] \equiv \mathsf{apply}(F[M], N)$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, \mathsf{apply}(F[M], N)) \to (C', \mathsf{apply}(F[M'], N))$ by the *ctx-apply-left* rule.

- Case $E \equiv \mathsf{apply}(V, F)$. In this case $E[M] \equiv \mathsf{apply}(V, F[M])$. By inductive hypothesis we know that $(C, F[M]) \to (C', F[M'])$, so we conclude $(C, \mathsf{apply}(V, F[M])) \to (C', \mathsf{apply}(V, F[M']))$ by the *ctx-apply-right* rule

Now we prove that if $(C, E[M]) \to (C', E[M'])$, then $(C, M) \to (C', M')$. We again proceed by induction on the form of E:

- Case $E \equiv [\cdot]$. In this case $E[M] \equiv M$ and the claim is trivially true.

- Case $E \equiv FN$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, (F[M])N) \to (C', (F[M'])N)}\, \textit{ctx-app-left}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

- Case $E \equiv VF$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, V(F[M])) \to (C', V(F[M']))}\, \textit{ctx-app-right}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

- Case $E \equiv \langle F, N\rangle$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, \langle F[M], N\rangle) \to (C', \langle F[M'], N\rangle)}\, \textit{ctx-tuple-left}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

- Case $E \equiv \langle V, F\rangle$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, \langle V, F[M]\rangle) \to (C', \langle V, F[M']\rangle)}\, \textit{ctx-tuple-right}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

- Case $E \equiv \mathsf{let}\, \langle x, y\rangle = F\, \mathsf{in}\, N$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, \mathsf{let}\, \langle x, y\rangle = F[M]\, \mathsf{in}\, N) \to (C', \mathsf{let}\, \langle x, y\rangle = F[M']\, \mathsf{in}\, N)}\, \textit{ctx-let}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

- Case $E \equiv \mathsf{force}\, F$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, \mathsf{force}\, F[M]) \to (C', \mathsf{force}\, F[M'])}\, \textit{ctx-force}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

- Case $E \equiv \mathsf{box}_T\, F$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, \mathsf{box}_T\, F[M]) \to (C', \mathsf{box}_T\, F[M'])}\, \textit{ctx-box}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

- Case $E \equiv \mathsf{apply}(F, N)$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, \mathsf{apply}(F[M], N)) \to (C', \mathsf{apply}(F[M'], N))}\, \textit{ctx-apply-left}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

- Case $E \equiv \mathsf{apply}(V, F)$. In this case we have

$$\frac{(C, F[M]) \to (C', F[M'])}{(C, \mathsf{apply}(V, F[M])) \to (C', \mathsf{apply}(V, F[M']))}\, \textit{ctx-apply-left}$$

  and by inductive hypothesis on F we immediately conclude $(C, M) \to (C', M')$.

□

Corollary 2.5.3.1. *Suppose we have a term M of the form $E[N]$ for some E. Then (C, M) is reducible if and only if (C, N) is reducible.*

Corollary 2.5.3.2. *Suppose we have a term M of the form $E[N]$ for some E. Then (C, M) is irreducible if and only if (C, N) is irreducible.*

In the light of the Fundamental Theorem of Evaluation Contexts, we can easily understand that if we have a term of the form $E[M]$, where M is a redex, then $E[M] \not\equiv F[N]$ for any other evaluation context F and redex N, otherwise we could choose to reduce M or N, leading to different configurations and breaking determinism. In fact, we prove that this result does not only hold for redexes, but more generally for *proto-redexes*. Intuitively, a proto-redex is a redex without the constraints on the form of the values that make it up. More formally, we give the following definition.

Definition 2.5.3 (Proto-redex)**.** *A term is said to be a* proto-redex *when it can be generated by the following grammar:*

$$P ::= VW \mid \mathsf{let}\ \langle x, y\rangle = V\ \mathsf{in}\ N \mid \mathsf{force}\, V \mid \mathsf{box}_T\, V \mid \mathsf{apply}(V, W).$$

Lemma 2.5.4. *Suppose M and N are proto-redexes. If $E[M] \equiv E'[N]$, then $E \equiv E'$.*

Proof. By induction on the form of E:

- Case $E \equiv [\cdot]$. In this case $E[M] \equiv M \equiv E'[N]$. Suppose $E' \not\equiv [\cdot]$. We proceed by cases on M:
  - $M \equiv VW$. In this case the only possibilities are $E' \equiv [\cdot]W$ and $E' \equiv V[\cdot]$. However, both would imply that N is a value, which is impossible since N is a proto-redex.
  - $M \equiv \mathsf{let}\ \langle x, y\rangle = V\ \mathsf{in}\ P$. In this case the only possibility is $E' \equiv \mathsf{let}\ \langle x, y\rangle = [\cdot]\ \mathsf{in}\ P$. However, this would imply that N is a value, which is impossible since N is a proto-redex.
  - $M \equiv \mathsf{force}(V)$. In this case the only possibility is $E' \equiv \mathsf{force}\,[\cdot]$. However, this would imply that N is a value, which is impossible since N is a proto-redex.
  - $M \equiv \mathsf{box}_T(V)$. In this case the only possibility is $E' \equiv \mathsf{box}_T\,[\cdot]$. However, this would imply that N is a value, which is impossible since N is a proto-redex.
  - $M \equiv \mathsf{apply}(V, W)$. In this case the only possibilities are $E' \equiv \mathsf{apply}([\cdot], W)$ and $E' \equiv \mathsf{apply}(V, [\cdot])$. However, both would imply that N is a value, which is impossible since N is a proto-redex.

  Since all possibilities lead to a contradiction, we conclude that $E' \equiv [\cdot] \equiv E$.
- $E \equiv FP$. In this case $E[M] \equiv (F[M])P \equiv E'[N]$ and the two possibilities are $E' \equiv E''P$ and $E' \equiv (F[M])E''$. In the former case we have $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv FP \equiv E$. The latter case would require that $F[M]$ be a value, which is impossible since M is a proto-redex.

- $E \equiv VF$. In this case $E[M] \equiv V(F[M]) \equiv E'[N]$ and the two possibilities are $E' \equiv E''(F[M])$ and $E' \equiv VE''$. The former would require that $E''[N]$ be a value, which is impossible since N is a proto-redex. In the latter case we have $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv VF \equiv E$.

- $E \equiv \langle F, P\rangle$. In this case $E[M] \equiv \langle F[M], P\rangle \equiv E'[N]$ and the two possibilities are $E' \equiv \langle E'', P\rangle$ and $E' \equiv \langle F[M], E''\rangle$. In the former case we have $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv \langle F, P\rangle \equiv E$. The latter case would require that $F[M]$ be a value, which is impossible since M is a proto-redex.

- $E \equiv \langle V, F\rangle$. In this case $E[M] \equiv \langle V, F[M]\rangle \equiv E'[N]$ and the two possibilities are $E' \equiv \langle E'', F[M]\rangle$ and $E' \equiv \langle V, E''\rangle$. The former would require that $E''[N]$ be a value, which is impossible since N is a proto-redex. In the latter case we have $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv \langle V, F\rangle \equiv E$.

- $E \equiv \mathsf{let}\,\langle x, y\rangle = F\,\mathsf{in}\,P$. In this case $E[M] \equiv \mathsf{let}\,\langle x, y\rangle = F[M]\,\mathsf{in}\,P \equiv E'[N]$ and the only possibility is $E' \equiv \mathsf{let}\,\langle x, y\rangle = E''\,\mathsf{in}\,P$. This implies $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv \mathsf{let}\,\langle x, y\rangle = F\,\mathsf{in}\,P \equiv E$.

- $E \equiv \mathsf{force}\,F$. In this case $E[M] \equiv \mathsf{force}\,F[M] \equiv E'[N]$ and the only possibility is $E' \equiv \mathsf{force}\,E''$. This implies $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv \mathsf{force}\,F \equiv E$.

- $E \equiv \mathsf{box}_T\,F$. In this case $E[M] \equiv \mathsf{box}_T\,F[M] \equiv E'[N]$ and the only possibility is $E' \equiv \mathsf{box}_T\,E''$. This implies $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv \mathsf{box}_T\,F \equiv E$.

- $E \equiv \mathsf{apply}(F, P)$. In this case $E[M] \equiv \mathsf{apply}(F[M], P) \equiv E'[N]$ and the two possibilities are $E' \equiv \mathsf{apply}(E'', P)$ and $E' \equiv \mathsf{apply}(F[M], E'')$. In the former case we have $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv \mathsf{apply}(F, P) \equiv E$. The latter case would require that $F[M]$ be a value, which is impossible since M is a proto-redex.

- $E \equiv \mathsf{apply}(V, F)$. In this case $E[M] \equiv \mathsf{apply}(V, F[M]) \equiv E'[N]$ and the two possibilities are $E' \equiv \mathsf{apply}(E'', F[M])$ and $E' \equiv \mathsf{apply}(V, E'')$. The former would require that $E''[N]$ be a value, which is impossible since N is a proto-redex. In the latter case we have $E''[N] \equiv F[M]$, so by inductive hypothesis we get $E'' \equiv F$ and conclude $E' \equiv \mathsf{apply}(V, F) \equiv E$.

□

Lemma 2.5.5. *If $E[M] \equiv E[N]$, then $M \equiv N$*

Proof. The claim follows naturally from the definition of evaluation context. The proof is trivial by induction on E. □

Proposition 2.5.6 (Context Exclusivity)**.** *Suppose M and N are proto-redexes. If $M \not\equiv N$, then $E[M] \not\equiv F[N]$, for any E, F.*

Proof. Suppose M and N are proto-redexes. If $E[M] \equiv F[N]$ for some E, F, then by Lemma 2.5.4 we get $E \equiv F$. Because $E[M] \equiv F[N] \equiv E[N]$, by Lemma 2.5.5 we also get $M \equiv N$, so we know that $E[M] \equiv F[N]$ entails $M \equiv N$. From this we conclude that if $M \not\equiv N$, then $E[M] \not\equiv F[N]$ for any E, F. □

Corollary 2.5.6.1. *Suppose M and N are redexes. If $M \not\equiv N$, then $E[M] \not\equiv F[N]$, for any E, F.*

Proof. The claim follows naturally from the fact that every redex is also a proto-redex. This is obvious, as every redex can be obtained from a production of the proto-redex grammar by instantiating the generic values occurring in the latter with the adequate explicit forms. □

We prove one last result for evaluation contexts: that if we "inject" an evaluation context into a second evaluation context, the result is still an evaluation context.

Proposition 2.5.7 (Context Propagation)**.** *Suppose we have a term M of the form $E[N]$ for some E. If N is of the form $E'[L]$ for some E', then M is of the form $E''[L]$ for some E''.*

Proof. By induction on the form of E:

- Case $E \equiv [\cdot]$. In this case $M \equiv N$ and the claim is trivially true.
- Case $E \equiv FP$. In this case $M \equiv (F[N])P$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv F'P$.
- Case $E \equiv VF$. In this case $M \equiv V(F[N])$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv VF'$.
- Case $E \equiv \langle F, P\rangle$. In this case $M \equiv \langle F[N], P\rangle$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv \langle F', P\rangle$.
- Case $E \equiv \langle V, F\rangle$. In this case $M \equiv \langle V, F[N]\rangle$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv \langle V, F'\rangle$.
- Case $E \equiv \mathsf{let}\,\langle x, y\rangle = F\,\mathsf{in}\,P$. In this case $M \equiv \mathsf{let}\,\langle x, y\rangle = F[N]\,\mathsf{in}\,P$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv \mathsf{let}\,\langle x, y\rangle = F'\,\mathsf{in}\,P$.
- Case $E \equiv \mathsf{force}\,F$. In this case $M \equiv \mathsf{force}\,F[N]$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv \mathsf{force}\,F'$.

- Case $E \equiv \mathsf{box}_T\, F$. In this case $M \equiv \mathsf{box}_T\, F[N]$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv \mathsf{box}_T\, F'$.
- Case $E \equiv \mathsf{apply}(F, P)$. In this case $M \equiv \mathsf{apply}(F[N], P)$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv \mathsf{apply}(F', P)$.
- Case $E \equiv \mathsf{apply}(V, F)$. In this case $M \equiv \mathsf{apply}(V, F[N])$. By inductive hypothesis we know that $F[N]$ is of the form $F'[L]$ for some F', so M is of the form $E''[L]$ for $E'' \equiv \mathsf{apply}(V, F')$.

□

Corollary 2.5.7.1. *Suppose we have a term M of the form $E[N]$ for some E. If M is not of the form $E'[L]$ for any E', then N is not of the form $E''[L]$ for any E''.*

2.5.2 Convergence, Deadlock and Divergence

Now that we have (deterministic) small-step semantics and evaluation contexts, we can start distinguishing between *converging*, *deadlocking* and *diverging* configurations. Informally, a configuration converges if its evaluation terminates successfully returning a value, it goes into deadlock if it gets stuck without returning a value, and it diverges if its evaluation does not terminate at all. The first two definitions are standard inductive definitions.

Definition 2.5.4 (Converging Small-step Configuration)**.** *Let $\downarrow$ be the smallest unary relation over small-step configurations such that:*

1. *For every circuit C and value V, $(C, V)\downarrow$,*
2. *If $(C, M) \to (D, N)$ and $(D, N)\downarrow$, then $(C, M)\downarrow$.*

We say that a configuration (C, M) is converging *when $(C, M)\downarrow$.*

Definition 2.5.5 (Deadlocking Small-step Configuration)**.** *Let $\bot$ be the smallest unary relation over small-step configurations such that:*

1. *If (C, M) is irreducible and M is neither of the form $E[\mathsf{box}_T(\mathsf{lift}\, N)]$, nor a value, then $(C, M)\bot$.*
2. *If $(C, M) \to (D, N)$ and $(D, N)\bot$, then $(C, M)\bot$.*
3. *If (C, M) is of the form $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])$ and $(id_Q, N\vec{\ell})\bot$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$, then $(C, M)\bot$.*

4. *If* (C, M) *is of the form* $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])$ *and* $(id_Q, N\vec{\ell}) \to^* (D, V)$*, where* $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$*, and* V *is not a label tuple, then* $(C, M)\bot$*.*

We say that a configuration (C, M) goes into deadlock *when* $(C, M)\bot$*.*

The last definition, the one for diverging configurations, is co-inductive. The intuitive difference between the two kinds of definition is that whereas an element belongs in an inductive set if there is a good reason for it to do so, an element belongs in a co-inductive set if there is no good reason for it not to. More practically, an inductive definition starts with the empty set and states the properties that an element must satisfy in order to get into the set, whereas a co-inductive definition starts with the universal set and states the properties that an element which is *already* in the set must satisfy in order not to get kicked out.

Definition 2.5.6 (Diverging Small-step Configuration)**.** *Let* $\uparrow$ *be the largest unary relation over small-step configurations such that whenever* $(C, M) \uparrow$ *either one of the following is true:*

1. $(C, M) \to (D, N)$ *and* $(D, N) \uparrow$*,*

2. (C, M) *is a configuration of the form* $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])$ *and* $(id_Q, N\vec{\ell}) \uparrow$*, where* $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$*.*

We say that a configuration (C, M) *is* diverging *when* $(C, M) \uparrow$*.*

What this definition says is that every configuration is diverging unless it is irreducible (i.e. a normal form or a deadlocked form) or reduces to a configuration that does not diverge (directly or as part of a sub-derivation introduced by box). Naturally, we expect convergence, deadlock and divergence to be mutually exclusive. This expectation is formalized by the following proposition.

Proposition 2.5.8. *The relations* $\downarrow, \bot$ *and* $\uparrow$ *are mutually exclusive over small-step configurations. That is, for every small-step configuration* (C, M)*, the following are true:*

1. *If* $(C, M) \downarrow$*, then* $(C, M) \not\perp$*,*

2. *If* $(C, M) \downarrow$*, then* $(C, M) \not\uparrow$*,*

3. *If* $(C, M)\bot$*, then* $(C, M) \not\uparrow$*.*

Proof. We prove each claim separately:

1. We proceed by induction on $(C, M) \downarrow$:

- Case of $M \equiv V$. Since (C, V) is irreducible, but V is a value and (as a consequence) cannot be of the form $E[\mathsf{box}_T(\mathsf{lift}\, N)]$, there is no way for (C, V) to go into deadlock, so we conclude $(C, V) \not\downarrow$.
- Case of $(C, M) \to (D, N)$ and $(D, N) \downarrow$. Since (C, M) is reducible, it must be that $(C, M) \to (D, N)$ ($\to$ is deterministic) and $(D, N)\bot$ in order for (C, M) to go into deadlock. However, by inductive hypothesis we know that $(D, N) \not\downarrow$, so this is impossible and we conclude $(C, M) \not\downarrow$.

2. We proceed by induction on $(C, M) \downarrow$:

- Case of $M \equiv V$. Since (C, V) is irreducible, V is a value and (as a consequence) cannot be of the form $E[\mathsf{box}_T(\mathsf{lift}\, N)]$, there is no way for (C, V) to diverge, so we conclude $(C, V) \not\uparrow$.
- Case of $(C, M) \to (D, N)$ and $(D, N) \downarrow$. Since (C, M) is reducible, it must be that $(C, M) \to (D, N)$ ($\to$ is deterministic) and $(D, N) \uparrow$ in order for (C, M) to diverge. However, by inductive hypothesis we know that $(D, N) \not\uparrow$, so this is impossible and we conclude $(C, M) \not\uparrow$.

3. We proceed by induction on $(C, M)\bot$:

- Case in which (C, M) is irreducible, $M \not\equiv V$ and $M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$. Since (C, M) is irreducible and M is not of the form $E[\mathsf{box}_T(\mathsf{lift}\, N)]$, there is no way for (C, M) to diverge, so we conclude $(C, M) \not\uparrow$.
- Case of $(C, M) \to (D, N)$ and $(D, N)\bot$. Since (C, M) is reducible, it must be that $(C, M) \to (D, N)$ and $(D, N) \uparrow$ in order for (C, M) to diverge. However, by inductive hypothesis we know that $(D, N) \not\uparrow$, so this is impossible and we conclude $(C, M) \not\uparrow$.
- Case of $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $(id_Q, N\vec{\ell})\bot$. Since (C, M) is irreducible, it must be that $(id_Q, N\vec{\ell}) \uparrow$ in order for (C, M) to diverge. However, by inductive hypothesis we know that $(id_Q, N\vec{\ell}) \not\uparrow$, so this is impossible and we conclude $(C, M) \not\uparrow$.
- Case of $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$, $(id_Q, N\vec{\ell}) \to^* (D, V)$ and V is not a label tuple, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. Since (C, M) is irreducible, it must be that $(id_Q, N\vec{\ell}) \uparrow$ in order for (C, M) to diverge. However, because $(id_Q, N\vec{\ell}) \to^* (D, V)$ (that is, $(id_Q, N\vec{\ell}) \downarrow$), we know by claim 2 that $(id_Q, N\vec{\ell}) \not\uparrow$, so this is impossible and we conclude $(C, M) \not\uparrow$.

□

Just as naturally, we expect the same relations to saturate the space of small-step configurations. That is, the three relations are defined in such a way that every configuration either converges or goes into deadlock or diverges.

Proposition 2.5.9. *Every small-step configuration (C, M) either converges, goes into deadlock or diverges, that is:*

$$(C, M)\downarrow \vee (C, M)\bot \vee (C, M)\uparrow .$$

Proof. Let clen be a function that, given a small-step configuration, returns the number of reduction steps that can be taken starting from that configuration, either at the top-level or in any of the sub-reductions introduced by a boxing operation. The clen function is defined as the least fixed point of the following equation on functions from small-step configurations to $\mathbb{N}^\infty$:

$$\mathsf{clen}_h(C, M) = \begin{cases} 0 & \text{if } (C, M) \text{ is irreducible,} \\ \mathsf{clen}(D, N) + 1 & \text{if } (C, M) \to (D, N). \end{cases}$$

$$\mathsf{clen}_v(C, M) = \begin{cases} 0 & \text{if } M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, N)], \\ \mathsf{clen}(id_Q, N\vec{\ell}) + 1 & \text{if } M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]. \end{cases}$$

$$\mathsf{clen}(C, M) = \mathsf{clen}_h(C, M) + \mathsf{clen}_v(C, M).$$

Where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. If $\mathsf{clen}(C, M) = \infty$, that means that either $(C, M) \to (D, N)$ and $\mathsf{clen}(D, N) = \infty$ or $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $\mathsf{clen}(id_Q, N\vec{\ell}) = \infty$. Because $\uparrow$ is defined as the largest relation such that $(C, M)\uparrow$ implies either $(C, M) \to (D, N)$ and $(D, N)\uparrow$ or $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $(id_Q, N\vec{\ell})\uparrow$, we conclude that $(C, M)\uparrow$. On the other hand, if $\mathsf{clen}(C, M) \in \mathbb{N}$, we proceed by induction on $\mathsf{clen}(C, M)$:

- Case $\mathsf{clen}(C, M) = 0$. In this case (C, M) is irreducible and $M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$. If M is a value, then we trivially conclude $(C, M)\downarrow$. Otherwise, if M is not a value, we trivially conclude $(C, M)\bot$.
- Case $\mathsf{clen}(C, M) = n + 1$. We distinguish two cases:
  - If $\mathsf{clen}_h(C, M) = 0$, then $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $\mathsf{clen}(id_Q, N\vec{\ell}) = n$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. By inductive hypothesis we know that either $(id_Q, N\vec{\ell})\downarrow$ or $(id_Q, N\vec{\ell})\bot$ (we exclude $(id_Q, N\vec{\ell})\uparrow$ as it would entail $\mathsf{clen}(C, E[\mathsf{box}_T(\mathsf{lift}\, N)]) = \infty \notin \mathbb{N}$). If $(id_Q, N\vec{\ell})\bot$ then we immediately conclude $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])\bot$. Otherwise, if $(id_Q, N\vec{\ell})\downarrow$ we know that $(id_Q, N\vec{\ell}) \to^* (D, V)$. If V were a label tuple $\vec{\ell'}$ we would have $(C, \mathsf{box}_T(\mathsf{lift}\, N)) \to (C, (\vec{\ell}, D, \vec{\ell'}))$ and, by Theorem 2.5.3, $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)]) \to (C, E[(\vec{\ell}, D, \vec{\ell'})])$, which would contradict the hypothesis that $\mathsf{clen}_h(C, E[\mathsf{box}_T(\mathsf{lift}\, N)]) = 0$. As a result, V is not a label tuple and we conclude $(E[\mathsf{box}_T(\mathsf{lift}\, N)])\bot$.
  - If $\mathsf{clen}_h(C, M) > 0$, then $(C, M) \to (D, N)$ and $\mathsf{clen}(D, N) \leq n$. By inductive hypothesis we know that either $(D, N)\downarrow$ or $(D, N)\bot$ (we exclude $(D, N)\uparrow$ as it

would entail $\mathsf{clen}(C, M) = \infty \notin \mathbb{N}$). If $(D, N) \downarrow$ then we immediately conclude $(C, M) \downarrow$. Otherwise, if $(D, N)\perp$ we immediately conclude $(C, M)\perp$.

□

2.5.3 Equivalence with the Big-Step Semantics

In order for our small-step semantics to be of any use, we must prove that it behaves like the original big-step semantics.

Convergence

In the following results we prove that our small-step semantics is ultimately equivalent to the big step semantics as far as converging computations are concerned. That is, that $(C, M) \Downarrow (D, N)$ implies $(C, M) \to^* (D, N)$ and vice-versa.

Lemma 2.5.10. *If* $(C, M) \Downarrow (D, V)$*, then* $(C, M) \to^* (D, V)$*.*

Proof. We proceed by induction on the derivation of $(C, M) \Downarrow (D, V)$:

- Case $M \equiv \vec{\ell}$ and

$$\frac{}{(C, \vec{\ell}) \Downarrow (C, \vec{\ell})}$$

  In this case the claim is trivially true by the reflexivity of $\to^*$.

- Case $M \equiv \lambda x.N$ and

$$\frac{}{(C, \lambda x.N) \Downarrow (C, \lambda x.N)}$$

  In this case the claim is trivially true by the reflexivity of $\to^*$.

- Case $M \equiv NP$ and

$$\frac{(C, N) \Downarrow (C_1, \lambda x.L) \quad (C_1, P) \Downarrow (C_2, W) \quad (C_2, L[W/x]) \Downarrow (D, V)}{(C, NP) \Downarrow (D, V)}$$

  In this case by inductive hypothesis we know $(C, N) \to^* (C_1, \lambda x.L)$ and $(C_1, P) \to^* (C_2, W)$, so by a finite number of applications of the *ctx-app-left* and *ctx-app-right* rules we get $(C, NP) \to^* (C_1, (\lambda x.L)P)$ and $(C_1, (\lambda x.L)P) \to^* (C_2, (\lambda x.L)W)$, respectively. Next, by the *β-reduction* rule we get $(C_2, (\lambda x.L)W) \to (C_2, L[W/x])$. By inductive hypothesis we also know $(C_2, L[W/x]) \to^* (D, V)$, so by the transitivity of $\to^*$ we conclude $(C, NP) \to^* (D, V)$.

- Case $M \equiv \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P$ and

$$\frac{(C, N) \Downarrow (C_1, \langle W_1, W_2\rangle) \quad (C_1, P[W_1/x][W_2/y]) \Downarrow (D, V)}{(C, \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P) \Downarrow (D, V)}$$

  In this case by inductive hypothesis we know $(C, N) \to^* (C_1, \langle W_1, W_2\rangle)$, so by a finite number of applications of the *ctx-let* rule we get $(C, \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P) \to^* (C_1, \mathsf{let}\,\langle x, y\rangle = \langle W_1, W_2\rangle\,\mathsf{in}\,P)$. Next, by the *let* rule we get $(C_1, \mathsf{let}\,\langle x, y\rangle = \langle W_1, W_2\rangle\,\mathsf{in}\,P) \to (C_1, P[W_1/x][W_2/y])$. By inductive hypothesis we also know $(C_1, P[W_1/x][W_2/y]) \to^* (D, V)$, so by the transitivity of $\to^*$ we conclude $(C, \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P) \to^* (D, V)$.

- Case $M \equiv \langle N, P\rangle$ and

$$\frac{(C, N) \Downarrow (C_1, W_1) \quad (C_1, P) \Downarrow (D, W_2)}{(C, \langle N, P\rangle) \Downarrow (D, \langle W_1, W_2\rangle)}$$

  In this case by inductive hypothesis we know $(C, N) \to^* (C_1, W_1)$ and $(C_1, P) \to^* (D, W_2)$, so by a finite number of applications of the *ctx-tuple-left* and *ctx-tuple-right* rules we get $(C, \langle N, P\rangle) \to^* (C_1, \langle W_1, P\rangle)$ and $(C_1, \langle W_1, P\rangle) \to^* (D, \langle W_1, W_2\rangle)$. By the transitivity of $\to^*$ we conclude $(C, \langle N, P\rangle) \to^* (D, \langle W_1, W_2\rangle)$.

- Case $M \equiv \mathsf{lift}\,N$ and

$$\frac{}{(C, \mathsf{lift}\,N) \Downarrow (C, \mathsf{lift}\,N)}$$

  In this case the claim is trivially true by the reflexivity of $\to^*$.

- Case $M \equiv \mathsf{force}\,N$ and

$$\frac{(C, N) \Downarrow (C_1, \mathsf{lift}\,P) \quad (C_1, P) \Downarrow (D, V)}{(C, \mathsf{force}\,N) \Downarrow (D, V)}$$

  In this case by inductive hypothesis we know $(C, N) \to^* (C_1, \mathsf{lift}\,P)$, so by a finite number of applications of the *ctx-force* rule we get $(C, \mathsf{force}\,N) \to^* (C_1, \mathsf{force}(\mathsf{lift}\,P))$. Next, by the *force* rule we get $(C_1, \mathsf{force}(\mathsf{lift}\,P)) \to (C_1, P)$. By inductive hypothesis we also know $(C_1, P) \to^* (D, V)$, so by the transitivity of $\to^*$ we conclude $(C, \mathsf{force}\,N) \to^* (D, V)$.

- Case $M \equiv \mathsf{box}_T\,N$ and

$$\frac{(C, N) \Downarrow (D, \mathsf{lift}\,P) \quad (Q, \vec{\ell}) = \mathsf{freshlabels}(P, T) \quad (id_Q, P\vec{\ell}) \Downarrow (D', \vec{\ell'})}{(C, \mathsf{box}_T\,N) \Downarrow (D, (\vec{\ell}, D', \vec{\ell'}))}$$

  In this case by inductive hypothesis we know $(C, N) \to^* (D, \mathsf{lift}\,P)$, so by a finite number of applications of the *ctx-box* rule we get $(C, \mathsf{box}_T\,N) \to^* (D, \mathsf{box}_T(\mathsf{lift}\,P))$.

By inductive hypothesis we also know $(id_Q, P\vec{\ell}) \to^* (D', \vec{\ell'})$, so by the *box* rule we get $(D, \mathsf{box}_T(\mathsf{lift}\, P)) \to (D, (\vec{\ell}, D', \vec{\ell'}))$ and conclude $(C, \mathsf{box}_T\, N) \to^* (D, (\vec{\ell}, D', \vec{\ell'}))$ by the transitivity of $\to^*$.

- Case $M \equiv \mathsf{apply}(N, P)$ and

$$\frac{(C, N) \Downarrow (C_1, (\vec{\ell}, D', \vec{\ell'})) \quad (C_1, P) \Downarrow (C_2, \vec{k}) \quad (D, \vec{k'}) = \mathsf{append}(C_2, \vec{k}, \vec{\ell}, D', \vec{\ell'})}{(C, \mathsf{apply}(N, P)) \Downarrow (D, \vec{k'})}$$

In this case by inductive hypothesis we know $(C, N) \to^* (C_1, (\vec{\ell}, D', \vec{\ell'}))$ and $(C_1, P) \to^* (C_2, \vec{k})$, so by a finite number of applications of the *ctx-apply-left* and *ctx-apply-right* rules we get $(C, \mathsf{apply}(N, P)) \to^* (C_1, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), P))$ and $(C_1, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), P)) \to^* (C_2, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k}))$, respectively. Next, by the *apply* rule we get $(C_2, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})) \to (D, \vec{k'})$ and conclude $(C, \mathsf{apply}(N, P)) \to^* (D, \vec{k'})$ by the transitivity of $\to^*$.

- Case $M \equiv (\vec{\ell}, D, \vec{\ell'})$ and

$$\frac{}{(C, (\vec{\ell}, D, \vec{\ell'})) \Downarrow (C, (\vec{\ell}, D, \vec{\ell'}))}$$

In this case the claim is trivially true by the reflexivity of $\to^*$.

□

The converse result is slightly more complicated to prove. This is because whereas a computation $(C, M) \Downarrow (D, V)$ is derived directly from the evaluation of its sub-terms, on which we can easily apply the inductive hypothesis (for example, $(C, \mathsf{force}\, M) \Downarrow (D, V)$ is derived directly from $(C, M) \Downarrow (C_1, \mathsf{lift}\, N)$ and $(C_1, N) \Downarrow (D, V)$), the same is not true for a reduction sequence $(C, M) \to^* (D, V)$, so it is not immediately clear where we can apply an inductive hypothesis. The following lemma tells us that any reduction sequence $(C, M) \to^* (D, V)$ can be broken down into multiple sub-sequences, each ideally corresponding to a premise of the corresponding big-step rule.

Lemma 2.5.11. *The following are all true:*

1. *If* $(C, MN) \to^+ (D, V)$, *then* $(C, MN) \to^* (C_1, (\lambda x.P)N) \to^* (C_2, (\lambda x.P)W) \to (C_2, P[W/x]) \to^* (D, V)$.
2. *If* $(C, \langle M, N\rangle) \to^+ (D, V)$, *then* $V \equiv \langle V_1, V_2\rangle$ *and* $(C, \langle M, N\rangle) \to^* (C_1, \langle V1, N\rangle) \to^* (D, \langle V_1, V_2\rangle)$.
3. *If* $(C, \mathsf{let}\, \langle x, y\rangle = M\, \mathsf{in}\, N) \to^+ (D, V)$, *then* $(C, \mathsf{let}\, \langle x, y\rangle = M\, \mathsf{in}\, N) \to^* (C_1, \mathsf{let}\, \langle x, y\rangle = \langle V, W\rangle\, \mathsf{in}\, N) \to (C_1, N[V/x][W/y]) \to^* (D, V)$.

4. *If* $(C, \mathsf{force}\, M) \to^+ (D, V)$, *then* $(C, \mathsf{force}\, M) \to^* (C_1, \mathsf{force}(\mathsf{lift}\, N)) \to (C_1, N) \to^* (D, V)$.

5. *If* $(C, \mathsf{box}_T(M)) \to^+ (D, V)$, *then* $(C, \mathsf{box}_T(M)) \to^* (D, \mathsf{box}_T(\mathsf{lift}\, N)) \to (D, (\vec{\ell}, D', \vec{\ell'}))$ *and* $(id_Q, N\vec{\ell}) \to^* (D', \vec{\ell'})$, *for* $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$.

6. *If* $(C, \mathsf{apply}(M, N)) \to^+ (D, V)$, *then* $(C, \mathsf{apply}(M, N)) \to^* (C_1, \mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), N)) \to^* (C_2, \mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})) \to (D, \vec{k'})$ *for* $(D, \vec{k'}) = \mathsf{append}(C_2, \vec{k}, \vec{\ell}, D', \vec{\ell'})$.

Proof. We prove each claim separately:

1. We proceed by induction on the length of the derivation $(C, MN) \to^+ (D, V)$:

   - Case $(C, MN) \to (D, V)$. In this case we necessarily have $M \equiv \lambda x.x$ and $N \equiv V$ and $C = D$. We get $(C, MN) = (C, (\lambda x.x)N) = (C, (\lambda x.x)V) \to (C, V)$ by the *app* rule, which trivially proves the claim by the reflexivity of $\to^*$.
   - Case $(C, MN) \to^+ (D, V)$ in two or more steps. We distinguish three cases
     - $M \equiv \lambda x.P$ and $N \equiv W$. In this case we have $(C, (\lambda x.P)W) \to (C, P[W/x])$ by the *β-reduction* rule and $(C, P[W/x]) \to^+ (D, V)$, so we immediately conclude $(C, MN) = (C, (\lambda x.P)N) = (C, (\lambda x.P)W) \to (C, P[W/x]) \to^* (D, V)$.
     - $M \equiv \lambda x.P$ and $N \not\equiv W$. In this case we have $(C, (\lambda x.P)N) \to (C_1, (\lambda x.P)N')$ by the *ctx-app-right* rule and $(C_1, (\lambda x.P)N') \to^+ (D, V)$. By inductive hypothesis we get $(C_1, (\lambda x.P)N') \to^* (C_2, (\lambda x.P)W) \to (C_2, P[W/x]) \to^* (D, V)$ and conclude $(C, MN) = (C, (\lambda x.P)N) \to^* (C_2, (\lambda x.P)W) \to (C_2, P[W/x]) \to^* (D, V)$ by the transitivity of $\to^*$.
     - $M \not\equiv \lambda x.P$ and $N \not\equiv W$. In this case we have $(C, MN) \to (C_1, M'N)$ by the *ctx-app-left* rule and $(C_1, M'N) \to^+ (D, V)$. By inductive hypothesis we get $(C_1, M'N) \to^* (C_2, (\lambda x.P)N) \to^* (C_3, (\lambda x.P)W) \to (C_3, P[W/x]) \to^* (D, V)$ and conclude $(C, MN) \to^* (C_2, (\lambda x.P)N) \to^* (C_3, (\lambda x.P)W) \to (C_3, P[W/x]) \to^* (D, V)$ by the transitivity of $\to^*$.

2. We proceed by induction on the length of the derivation $(C, \langle M, N\rangle) \to^+ (D, V)$:

   - Case $(C, \langle M, N\rangle) \to (D, V)$. In this case we necessarily have $M \equiv V_1$ and $(C, N) \to (D, V_2)$. We get $(C, \langle M, N\rangle) = (C, \langle V_1, N\rangle) \to (D, \langle V_1, V_2\rangle)$ by the *ctx-tuple-right* rule, which trivially proves the claim by the reflexivity of $\to^*$.
   - Case $(C, \langle M, N\rangle) \to^+ (D, V)$ in two or more steps. We distinguish two cases
     - $M \equiv V_1$. In this case we have $(C, \langle V_1, N\rangle) \to (C_1, \langle V_1, N'\rangle)$ by the *ctx-tuple-right* rule and $(C_1, \langle V_1, N'\rangle) \to^+ (D, V)$. By inductive hypothesis we get $(C_1, \langle V_1, N'\rangle) \to^* (D, \langle V_1, V_2\rangle)$ and conclude $(C, \langle M, N\rangle) = (C, \langle V_1, N\rangle) \to^* (D, \langle V_1, V_2\rangle)$ by the transitivity of $\to^*$.

  - $M \not\equiv V_1$. In this case we have $(C, \langle M, N\rangle) \to (C_1, \langle M', N\rangle)$ by the *ctx-tuple-left* rule and $(C_1, \langle M', N\rangle) \to^+ (D, V)$. By inductive hypothesis we get $(C_1, \langle M', N\rangle) \to^* (C_2, \langle V_1, N\rangle) \to^* (D, \langle V_1, V_2\rangle)$ and conclude $(C, \langle M, N\rangle) \to^* (C_2, \langle V_1, N\rangle) \to^* (D, \langle V_1, V_2\rangle)$ by the transitivity of $\to^*$.

3. We proceed by induction on the length of the derivation $(C, \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N) \to^+ (D, V)$:

   - Case $(C, \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N) \to (D, V)$. In this case we necessarily have $M \equiv \langle W_1, W_2\rangle$ and $N \equiv V$ and $C = D$. We get $(C, \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,V) = (C, \mathsf{let}\,\langle x, y\rangle = \langle W_1, W_2\rangle\,\mathsf{in}\,V) \to (C, V[W_1/x][W_2/y]) = (C, V)$ by the *let* rule, which trivially proves the claim by the reflexivity of $\to^*$.
   - Case $(C, \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N) \to^+ (D, V)$ in two or more steps. We distinguish two cases:
     - Case $M \equiv \langle W_1, W_2\rangle$. In this case we have $(C, \mathsf{let}\,\langle x, y\rangle = \langle W_1, W_2\rangle\,\mathsf{in}\,N) \to (C, N[W_1/x][W_2/y])$ by the *let* rule and $(C, N[W_1/x][W_2/y]) \to^+ (D, V)$, so we immediately conclude $(C, \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N) = (C, \mathsf{let}\,\langle x, y\rangle = \langle W_1, W_2\rangle M\,\mathsf{in}\,N) \to (C, N[W_1/x][W_2/y]) \to^* (D, V)$.
     - Case $M \not\equiv \langle W_1, W_2\rangle$. In this case we have $(C, \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N) \to (C_1, \mathsf{let}\,\langle x, y\rangle = M'\,\mathsf{in}\,N)$ by the *ctx-let* rule and $(C_1, \mathsf{let}\,\langle x, y\rangle = M'\,\mathsf{in}\,N) \to^+ (D, V)$. By inductive hypothesis we get $(C_1, \mathsf{let}\,\langle x, y\rangle = M'\,\mathsf{in}\,N) \to^* (C_2, \mathsf{let}\,\langle x, y\rangle = \langle W_1, W_2\rangle\,\mathsf{in}\,N) \to (C_2, N[W_1/x][W_2/y]) \to^* (D, V)$ and conclude $(C, \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N) \to^* (C_2, \mathsf{let}\,\langle x, y\rangle = \langle W_1, W_2\rangle\,\mathsf{in}\,N) \to (C_2, N[W_1/x][W_2/y]) \to^* (D, V)$ by the transitivity of $\to^*$.

4. We proceed by induction on the length of the derivation $(C, \mathsf{force}\,M) \to^+ (D, V)$:

   - Case $(C, \mathsf{force}\,M) \to (D, V)$. In this case we necessarily have $M \equiv \mathsf{lift}\,V$ and $C = D$. We get $(C, \mathsf{force}\,M) = (C, \mathsf{force}(\mathsf{lift}\,V)) \to (C, V)$ by the *force* rule, which trivially proves the claim by the reflexivity of $\to^*$.
   - Case $(C, \mathsf{force}\,M) \to^+ (D, V)$ in two or more steps. We distinguish two cases:
     - Case $M \equiv \mathsf{lift}\,N$. In this case we have $(C, \mathsf{force}(\mathsf{lift}\,N)) \to (C, N)$ by the *force* rule and $(C, N) \to^+ (D, V)$, so we immediately conclude $(C, \mathsf{force}\,M) = (C, \mathsf{force}(\mathsf{lift}\,N)) \to (C, N) \to^* (D, V)$.
     - Case $M \not\equiv \mathsf{lift}\,N$. In this case we have $(C, \mathsf{force}\,M) \to (C_1, \mathsf{force}\,M')$ by the *ctx-force* rule and $(C_1, \mathsf{force}\,M') \to^+ (D, V)$. By inductive hypothesis we get $(C_1, \mathsf{force}\,M') \to^* (C_2, \mathsf{force}(\mathsf{lift}\,N)) \to (C_2, N) \to^* (D, V)$ and conclude $(C, \mathsf{force}\,M) \to^* (C_2, \mathsf{force}(\mathsf{lift}\,N)) \to (C_2, N) \to^* (D, V)$ by the transitivity of $\to^*$.

5. We proceed by induction on the length of the derivation $(C, \mathsf{box}_T\,M) \to^+ (D, V)$:

- Case $(C, \mathsf{box}_T\, M) \to (D, V)$. In this case we necessarily have $M \equiv \mathsf{lift}\, N$ and $(id_Q, N\vec{\ell}) \to^* (D', \vec{\ell'})$ for $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$ and $C = D$. We get $(C, \mathsf{box}_T\, M) = (C, \mathsf{box}_T(\mathsf{lift}\, N)) \to (C, (\vec{\ell}, D', \vec{\ell'}))$ by the *box* rule, which trivially proves the claim by the reflexivity of $\to^*$.
- Case $(C, \mathsf{box}_T\, M) \to^+ (D, V)$ in two or more steps. We distinguish two cases:
  - Case $M \equiv \mathsf{lift}\, N$. This case is impossible since it would entail $(C, \mathsf{box}_T(\mathsf{lift}\, N)) \to (C, (\vec{\ell}, D', \vec{\ell'}))$ in one step.
  - Case $M \not\equiv \mathsf{lift}\, N$. In this case we have $(C, \mathsf{box}_T\, M) \to (C_1, \mathsf{box}_T\, M')$ by the *ctx-box* rule and $(C_1, \mathsf{box}_T\, M') \to^+ (D, V)$. By inductive hypothesis we get $(C_1, \mathsf{box}_T\, M') \to^* (D, \mathsf{box}_T(\mathsf{lift}\, N)) \to (D, (\vec{\ell}, D', \vec{\ell'}))$ and $(id_Q, N\vec{\ell}) \to^* (D', \vec{\ell'})$ for $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. We conclude $(C, \mathsf{box}_T\, M) \to^* (D, \mathsf{box}_T(\mathsf{lift}\, N)) \to (D, (\vec{\ell}, D', \vec{\ell'}))$ by the transitivity of $\to^*$.

6. We proceed by induction on the length of the derivation $(C, \mathsf{apply}(M, N)) \to^+ (D, V)$:

- Case $(C, \mathsf{apply}(M, N)) \to (D, V)$. In this case we necessarily have $M \equiv (\vec{\ell}, D', \vec{\ell'})$ and $N \equiv \vec{k}$. As a consequence, we know that $(C, \mathsf{apply}(M, N)) = (C, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), N)) = (C, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})) \to (D, \vec{k'})$ by the *apply* rule, where $(D, \vec{k'}) = \mathsf{append}(C, \vec{k}, \vec{\ell}, D', \vec{\ell'})$. This trivially proves the claim by the reflexivity of $\to^*$.
- Case $(C, \mathsf{apply}(M, N)) \to^+ (D, V)$ in two or more steps. We distinguish three cases:
  - $M \equiv (\vec{\ell}, D', \vec{\ell'})$ and $N \equiv \vec{k}$. This case is impossible since it would entail $(C, \mathsf{apply}(M, N)) \to (D, V)$ in one step.
  - $M \equiv (\vec{\ell}, D', \vec{\ell'})$ and $N \not\equiv \vec{k}$. In this case we have $(C, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), N)) \to (C_1, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), N'))$ by the *ctx-apply-right* rule, and we also have $(C_1, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), N')) \to^+ (D, V)$. By inductive hypothesis we know that $(C_1, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), N')) \to^* (C_2, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})) \to (D, \vec{k'})$, where $(D, \vec{k'}) = \mathsf{append}(C_2, \vec{k}, \vec{\ell}, D', \vec{\ell'})$, and conclude $(C, \mathsf{apply}(M, N)) = (C, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), N)) \to^* (C_2, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})) \to (D, \vec{k'})$ by the transitivity of $\to^*$.
  - $M \not\equiv (\vec{\ell}, D', \vec{\ell'})$ and $N \not\equiv \vec{k}$. In this case we have $(C, \mathsf{apply}(M, N)) \to (C_1, \mathsf{apply}(M', N))$ by the *ctx-apply-left* rule and $(C_1, M'N) \to^+ (D, V)$. By inductive hypothesis we get $(C_1, \mathsf{apply}(M', N)) \to^* (C_2, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), N)) \to^* (C_3, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})) \to (D, \vec{k'})$, where $(D, \vec{k'}) = \mathsf{append}(C_3, \vec{k}, \vec{\ell}, D', \vec{\ell'})$ and therefore conclude $(C, \mathsf{apply}(M, N)) \to^* (C_2, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), N)) \to^* (C_3, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})) \to (D, \vec{k'})$ by the transitivity of $\to^*$.

□

Lemma 2.5.12. *If* $(C, M) \to^* (D, V)$*, then* $(C, M) \Downarrow (D, V)$.

Proof. We reason by induction on the size of the proof for $(C, M) \to^* (D, V)$, proceeding by cases on M:

- Case $M \equiv x$. This case is impossible, since by the definition of small-step configuration M must contain no free variables.
- Case $M \equiv \vec{\ell}$. In this case $\vec{\ell}$ is a value, so we trivially have $C = D$ and

$$\frac{}{(C, \vec{\ell}) \Downarrow (C, \vec{\ell})}$$

- Case $M \equiv \lambda x.N$. In this case $\lambda x.N$ is a value, so we trivially have $C = D$ and

$$\frac{}{(C, \lambda x.N) \Downarrow (C, \lambda x.N)}$$

- Case $M \equiv NP$. In this case we have $(C, NP) \to^+ (D, V)$ and by Lemma 2.5.11 we know that $(C, NP) \to^* (C_1, (\lambda x.L)P) \to^* (C_2, (\lambda x.L)W) \to (C_2, L[W/x]) \to^* (D, V)$. By repeated inversion on the *ctx-app-left* and *ctx-app-right* rules we get $(C, N) \to^* (C_1, \lambda x.L)$ and $(C1, P) \to^* (C_2, W)$, respectively. By inductive hypothesis we therefore know $(C, N) \Downarrow (C_1, \lambda x.L)$ and $(C1, P) \Downarrow (C_2, W)$ and $(C_2, L[W/x]) \Downarrow (D, V)$, by which we conclude

$$\frac{(C, N) \Downarrow (C_1, \lambda x.L) \quad (C1, P) \Downarrow (C_2, W) \quad (C_2, L[W/x]) \Downarrow (D, V)}{(C, NP) \Downarrow (D, V)}$$

- Case $M \equiv \langle N, P \rangle$. In this case we distinguish two cases. If $\langle N, P \rangle \equiv \langle V_1, V_2 \rangle$, then $C = D$ and we trivially conclude

$$\frac{}{(C, \langle V_1, V_2 \rangle) \Downarrow (C, \langle V_1, V_2 \rangle)}$$

  Otherwise, if $(C, \langle N, P \rangle) \to^+ (D, V)$, then by Lemma 2.5.11 we know $(C, \langle N, P \rangle) \to^* (C_1, \langle V_1, P \rangle) \to^* (D, \langle V_1, V_2 \rangle)$. By repeated inversion on the *ctx-tuple-left* and *ctx-tuple-right* rules we get $(C, N) \to^* (C_1, V_1)$ and $(C_1, P) \to^* (D, V_2)$, respectively. By inductive hypothesis we therefore know $(C, N) \Downarrow (C_1, V_1)$ and $(C_1, P) \Downarrow (D, V_2)$, by which we conclude

$$\frac{(C, N) \Downarrow (C_1, V_1) \quad (C_1, P) \Downarrow (D, V_2)}{(C, \langle N, P \rangle) \Downarrow (D, \langle V_1, V_2 \rangle)}$$

- Case $M \equiv \mathsf{let}\, \langle x, y \rangle = N \,\mathsf{in}\, P$. In this case we have $(C, \mathsf{let}\, \langle x, y \rangle = N \,\mathsf{in}\, P) \to^+ (D, V)$ and by Lemma 2.5.11 we know that $(C, \mathsf{let}\, \langle x, y \rangle = N \,\mathsf{in}\, P) \to^* (C_1, \mathsf{let}\, \langle x, y \rangle =$

$\langle W_1, W_2\rangle \,\mathsf{in}\, P) \to (C_1, P[W_1/x][W_2/y]) \to^* (D, V)$. By repeated inversion on the *ctx-let* we get $(C, N) \to^* (C_1, \langle W_1, W_2\rangle)$. By inductive hypothesis we therefore know $(C, N) \Downarrow (C_1, \langle W_1, W_2\rangle)$ and $(C_1, P[W_1/x][W_2/y]) \Downarrow (D, V)$, by which we conclude

$$\frac{(C, N) \Downarrow (C_1, \mathsf{lift}\, P) \quad (C_1, P[W_1/x][W_2/y]) \Downarrow (D, V)}{(C, \mathsf{let}\, \langle x, y\rangle = N \,\mathsf{in}\, P) \Downarrow (D, V)}$$

- Case $M \equiv \mathsf{lift}\, N$. In this case $\mathsf{lift}\, N$ is a value, so we trivially have $C = D$ and

$$\frac{}{(C, \mathsf{lift}\, N) \Downarrow (C, \mathsf{lift}\, N)}$$

- Case $M \equiv \mathsf{force}\, N$. In this case we have $(C, \mathsf{force}\, N) \to^+ (D, V)$ and by Lemma 2.5.11 we know that $(C, \mathsf{force}\, N) \to^* (C_1, \mathsf{force}(\mathsf{lift}\, P)) \to (C_1, P) \to^* (D, V)$ and by repeated inversion on the *ctx-force* rule we get $(C, N) \to^* (C_1, \mathsf{lift}\, P)$. By inductive hypothesis we therefore know $(C, N) \Downarrow (C_1, \mathsf{lift}\, P)$ and $(C_1, P) \Downarrow (D, V)$, by which we conclude

$$\frac{(C, N) \Downarrow (C_1, \mathsf{lift}\, P) \quad (C_1, P) \Downarrow (D, V)}{(C, \mathsf{force}(\mathsf{lift}\, P)) \Downarrow (D, V)}$$

- Case $M \equiv \mathsf{box}_T\, N$. In this case we have $(C, \mathsf{box}_T\, N) \to^+ (D, V)$ and by Lemma 2.5.11 we know that $(C, \mathsf{box}_T\, N) \to^* (D, \mathsf{box}_T(\mathsf{lift}\, P)) \to (D, (\vec{\ell}, D', \vec{\ell'}))$ and $(id_Q, P\vec{\ell}) \to^* (D', \vec{\ell'})$, for $(Q, \vec{\ell}) = \mathsf{freshlabels}(P, T)$ and by repeated inversion on the *ctx-box* rule we get $(C, N) \to^* (D, \mathsf{lift}\, P)$. By inductive hypothesis we therefore know $(C, N) \Downarrow (D, \mathsf{lift}\, P)$ and $(id_Q, P\vec{\ell}) \Downarrow (D', \vec{\ell'})$, by which we conclude

$$\frac{(C, N) \Downarrow (D, \mathsf{lift}\, P) \quad (Q, \vec{\ell}) = \mathsf{freshlabels}(P, T) \quad (id_Q, P\vec{\ell}) \Downarrow (D', \vec{\ell'})}{(C, \mathsf{box}_T\, M) \Downarrow (D, (\vec{\ell}, D', \vec{\ell'}))}$$

- Case $M \equiv \mathsf{apply}(N, P)$. In this case we have $(C, \mathsf{apply}(N, P)) \to^+ (D, V)$ and by Lemma 2.5.11 we know that $(C, \mathsf{apply}(N, P)) \to^* (C_1, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), P)) \to^* (C_2, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})) \to (D, \vec{k'})$. By repeated inversion on the *ctx-apply-left* and *ctx-apply-right* rules we get $(C, N) \to^* (C_1, (\vec{\ell}, D', \vec{\ell'}))$ and $(C1, P) \to^* (C_2, \vec{k})$, respectively. By inductive hypothesis we therefore know $(C, N) \Downarrow (C_1, (\vec{\ell}, D', \vec{\ell'}))$ and $(C1, P) \Downarrow (C_2, \vec{k})$, by which we conclude

$$\frac{(C, N) \Downarrow (C_1, (\vec{\ell}, D', \vec{\ell'})) \quad (C1, P) \Downarrow (C_2, \vec{k}) \quad (D, \vec{k'}) = \mathsf{append}(C_2, \vec{k}, \vec{\ell}, D', \vec{\ell'})}{(C, \mathsf{apply}(N, P)) \Downarrow (D, \vec{k'})}$$

- Case $M \equiv (\vec{\ell}, D', \vec{\ell'})$. In this case $(\vec{\ell}, D', \vec{\ell'})$ is a value, so we trivially have $C = D$ and

$$\frac{}{(C, (\vec{\ell}, D', \vec{\ell'})) \Downarrow (C, (\vec{\ell}, D', \vec{\ell'}))}$$

□

Finally, now that we know that the small-step semantics can simulate the big-step semantics and vice-versa, we can summarize the previous results in the following theorem.

Theorem 2.5.13. *Suppose* (C, M) *and* (D, V) *are small-step configurations. We have that* $(C, M) \Downarrow (D, V)$ *if and only if* $(D, M) \to^* (D, V)$.

Proof. The claim follows immediately from lemmata 2.5.10 and 2.5.12. □

Deadlock and Divergence

Although some form of equivalence could be expected between the error relation of the big-step semantics ($\Downarrow$ Error) and the deadlocking relation of the small-step semantics ($\bot$), this is not the case. This is mainly due to the fact that the big-step semantics interrupts the evaluation of a term as soon as an error is encountered, whereas by the definition of deadlocking configuration we reduce a term as much as possible before declaring that it is indeed stuck. If we had, for example, a configuration (C, MN) where the evaluation of M leads to something that is not a function and the evaluation of N diverges, in the big-step semantics the configuration would evaluate to an error as soon as M is evaluated, whereas in the small-step semantics it would diverge. This means that the line between deadlocking and diverging configurations is not the same in the big-step and small-step semantics. This discrepancy, however, is not relevant, as by Theorem 2.5.13 we know that there are no cases in which the same configuration evaluates correctly in one semantics and raises an error (or diverges) in the other.

2.5.4 Safety Results

The operational semantics is not the only aspect under which circuits and terms are related. We mentioned earlier that all the occurrences of labels within a term are free, and in Section 2.3 we saw that labels are given a type by label contexts. In order for a configuration (C, M) to be considered *well-typed*, we must be able to derive a type judgement for M using exclusively labels coming from the outputs of C.

Definition 2.5.7 (Well-typedness)**.** *Given label contexts* Q *and* Q'*, a type* A *and a configuration* (C, M)*, we say that the latter is* well-typed *with input labels* Q*, output labels* Q' *and type* A*, and we write*

$$Q \vdash (C, M) : A; Q',$$

when there exists Q'' *disjoint from* Q' *such that*

$$C : Q \to Q' \cup Q'', \quad \emptyset; Q'' \vdash M : A.$$

In the following pages we prove two key safety properties of the small-step semantics: *subject reduction* and *progress*.

Subject Reduction

The subject reduction result tells us that reducing a configuration in the small-step semantics does not alter its type.

Lemma 2.5.14 (Bridging)**.** *Suppose we have circuits $C : Q_1 \to Q_1' \cup Q_1'', D : Q_2 \to Q_2' \cup Q_2''$, terms M, N and types A, B. If $Q_1 \vdash (C, M) : A; Q_1' \implies Q_2 \vdash (D, N) : B; Q_2'$, then*

$$\emptyset; Q_1'' \vdash M : A \implies \emptyset; Q_2'' \vdash N : B.$$

Proof. The proof follows immediately from the definition of well-typedness. □

Theorem 2.5.15 (Subject Reduction)**.** *If $Q \vdash (C, M) : A; Q'$ and $(C, M) \to (C', M')$, then $Q \vdash (C', M') : A; Q'$*

Proof. By the definition of well-typedness we know that there exists Q'' disjoint from Q' such that $C \to Q' \cup Q''$ and $\emptyset; Q'' \vdash M : A$. We proceed by induction on the derivation of $(C, M) \to (C', M')$:

- Case of *β-reduction*. Suppose $M \equiv (\lambda x.N)V$. We know $Q \vdash (C, (\lambda x.N)V) : A; Q'$ and we must prove $Q \vdash (C, N[V/x]) : A; Q'$. We know $\emptyset; Q'' \vdash (\lambda x.N)V : A$, so by Lemma 2.3.1 we get

  $$\emptyset; Q_1'' \vdash (\lambda x.N) : B \multimap A, \quad \emptyset; Q_2'' \vdash V : B,$$

  for some B and Q_1'', Q_2'' such that $Q'' = Q_1'', Q_2''$. By the same lemma, we get

  $$x : B; Q_1'' \vdash N : A,$$

  By Theorem 2.3.6 we get $\emptyset; Q_1'', Q_2'' \vdash N[V/x] : A$ and we conclude

  $$Q \vdash (C, N[V/x]) : A; Q'.$$

- Case of *let*. Suppose $M \equiv \mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N$. We know $Q \vdash (C, \mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N) : A; Q'$ and we must prove $Q \vdash (C, N[V/x][W/y]) : A; Q'$. We know $\emptyset; Q'' \vdash \mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N : A$, so by applying Lemma 2.3.1 twice consecutively we get

  $$\emptyset; Q_1'' \vdash V : B, \quad \emptyset; Q_2'' \vdash W : C, \quad x : B, y : C; Q_3'' \vdash V : B,$$

  for some B, C and Q_1'', Q_2'', Q_3'' such that $Q_1'', Q_2'', Q_3'' = Q''$. By Theorem 2.3.6 we get first $y : C; Q_3'', Q_1'' \vdash N[V/x] : A$ and then $\emptyset; Q_3'', Q_1'', Q_2'' \vdash N[V/x][W/y] : A$ and we conclude

  $$Q \vdash (C, N[V/x][W/y]) : A; Q'.$$

- Case of *force*. Suppose $M \equiv \mathsf{force}(\mathsf{lift}\, N)$. We know $Q \vdash (C, \mathsf{force}(\mathsf{lift}\, N)) : A; Q'$ and we must prove $Q \vdash (C, N) : A; Q'$. We know $\emptyset; Q'' \vdash \mathsf{force}(\mathsf{lift}\, N) : A$, so by applying Lemma 2.3.1 twice consecutively we get that $Q'' = \emptyset$ and

$$\emptyset; \emptyset \vdash \mathsf{lift}\, N : !A, \quad \emptyset; \emptyset \vdash N : A,$$

  from which we conclude

$$Q \vdash (C, N) : A; Q'.$$

- Case of *box*. Suppose $M \equiv \mathsf{box}_T(\mathsf{lift}\, N)$. We know $Q \vdash \mathsf{box}_T(\mathsf{lift}\, N) : \mathsf{Circ}(T, U); Q'$ and we must prove $Q \vdash (C, (\vec{\ell}, D, \vec{\ell'})) : \mathsf{Circ}(T, U); Q'$, where $(Q_0, \vec{\ell}) = \mathsf{freshlabels}(N, T)$ and $(id_{Q_0}, N\vec{\ell}) \to \ldots \to (D, \vec{\ell'})$. We know $\emptyset; Q'' \vdash \mathsf{box}_T(\mathsf{lift}\, N) : \mathsf{Circ}(T, U)$ and, by the definition of $\mathsf{freshlabels}$, $\emptyset; Q_0 \vdash \vec{\ell} : T$. By applying Lemma 2.3.1 twice we get that $Q'' = \emptyset$ and

$$\emptyset; \emptyset \vdash \mathsf{lift}\, N : !(T \multimap U), \quad \emptyset; \emptyset \vdash N : T \multimap U,$$

  and we therefore know by the *app* rule that

$$\emptyset; Q_0 \vdash N\vec{\ell} : U,$$

  which entails that the configuration $(id_{Q_0}, N\vec{\ell})$ is well-typed, with $Q_0 \vdash (id_{Q_0}, N\vec{\ell}) : U; \emptyset$. By applying the inductive hypothesis n times, we know that all the configurations in $(id_{Q_0}, N\vec{\ell}) \to \ldots \to (D, \vec{\ell'})$ are well typed, with $Q_0 \vdash (C_i, M_i) : U; \emptyset$ for $i = 1 \ldots n$. This includes $(C_n, M_n) \equiv (D, \vec{\ell'})$, so we get

$$Q_0 \vdash (D, \vec{\ell'}) : U; \emptyset.$$

  This in turn entails that there exists Q_0'' such that $\emptyset; Q_0'' \vdash \vec{\ell'} : U$ and D is a circuit of the form $D : Q_0 \to Q_0''$. Thus $D \in \mathbf{M}_{\mathcal{L}}(Q_0, Q_0'')$ and by the *circ* rule we get $\emptyset; \emptyset \vdash (\vec{\ell}, D, \vec{\ell'}) : \mathsf{Circ}(T, U)$, from which we conclude

$$Q \vdash (C, (\vec{\ell}, D, \vec{\ell'})) : \mathsf{Circ}(T, U), Q'.$$

- Case of *apply*. Suppose $M \equiv \mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})$. We know $Q \vdash (C, \mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})) : U; Q'$ and we must prove $Q \vdash (C', \vec{k'}) : U; Q'$, where $(C', \vec{k'}) = \mathsf{append}(C, \vec{k}, \vec{\ell}, D, \vec{\ell'})$. We know $\emptyset; Q'' \vdash \mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k}) : U$, so by Lemma 2.3.1 we get

$$\emptyset; \emptyset \vdash (\vec{\ell}, D, \vec{\ell'}) : \mathsf{Circ}(T, U), \quad \emptyset; Q'' \vdash \vec{k} : T,$$

  for some T. We also know that append finds $(\vec{k}, D', \vec{k'}) \cong (\vec{\ell}, D, \vec{\ell'})$ to apply to the outputs of C. By Lemma 2.3.1 we get $\emptyset; Q'' \vdash \vec{k} : T$, which we already know, as well as

$$\emptyset; Q''' \vdash \vec{k'} : U, \quad D' \in \mathbf{M}_{\mathcal{L}}(Q'', Q'''),$$

for some Q'''. By appending $D' : Q'' \to Q'''$ to $C : Q \to Q' \cup Q''$ we thus obtain $C' : Q \to Q' \cup Q'''$ and because $\emptyset; Q''' \vdash \vec{k'} : U$ we conclude

$$Q \vdash (C', \vec{k'}) : U; Q'.$$

- Case of *ctx-app-left*. Suppose $M \equiv NP$, where N is not a value. We know $Q \vdash (C, NP) : A; Q'$ and we must prove $Q \vdash (C', N'P) : A; Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash NP : A$, so by Lemma 2.3.1 we get

  $$\emptyset; Q''_1 \vdash N : B \multimap A, \quad \emptyset; Q''_2 \vdash P : B,$$

  for some B and Q''_1, Q''_2 such that $Q''_1, Q''_2 = Q''$. This entails $Q \vdash (C, N) : B \multimap A; Q' \cup Q''_2$ and by inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' \vdash N' : B \multimap A$, with $C' : Q \to Q' \cup Q''_2 \cup Q'''$. Finally, by the *app* rule we get $\emptyset; Q''', Q''_2 \vdash N'P : A$ and conclude

  $$Q \vdash (C', N'P) : A; Q'.$$

- Case of *ctx-app-right*. Suppose $M \equiv VN$, where N is not a value. We know $Q \vdash (C, VN) : A; Q'$ and we must prove $Q \vdash (C', VN') : A; Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash VN : A$, so by Lemma 2.3.1 we get

  $$\emptyset; Q''_1 \vdash V : B \multimap A, \quad \emptyset; Q''_2 \vdash N : B,$$

  for some B and Q''_1, Q''_2 such that $Q''_1, Q''_2 = Q''$. This entails $Q \vdash (C, N) : B; Q' \cup Q''_1$ and by inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' \vdash N' : B$, with $C' : Q \to Q' \cup Q''_1 \cup Q'''$. Finally, by the *app* rule we get $\emptyset; Q''_1, Q''' \vdash VN' : A$ and conclude

  $$Q \vdash (C', VN') : A; Q'.$$

- Case of *ctx-tuple-left*. Suppose $M \equiv \langle N, P \rangle$, where N is not a value. We know $Q \vdash (C, \langle N, P \rangle) : B \otimes C; Q'$ and we must prove $Q \vdash (C', \langle N', P \rangle) : B \otimes C; Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash \langle N, P \rangle : B \otimes C$, so by Lemma 2.3.1 we get

  $$\emptyset; Q''_1 \vdash N : B, \quad \emptyset; Q''_1 \vdash P : C,$$

  for some B, C and Q''_1, Q''_2 such that $Q''_1, Q''_2 = Q''$. This entails $Q \vdash (C, N) : B; Q' \cup Q''_2$ and by inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' \vdash N' : B$ with $C' : Q \to Q' \cup Q''_2 \cup Q'''$. Finally, by the *tuple* rule we get $\emptyset; Q''_1, Q''_2 \vdash \langle N', P \rangle : B \otimes C$ and conclude

  $$Q \vdash (C', \langle N', P \rangle) : B \otimes C; Q'.$$

- Case of *ctx-tuple-right*. Suppose $M \equiv \langle V, N\rangle$, where N is not a value. We know $Q \vdash (C, \langle V, N\rangle) : B \otimes C; Q'$ and we must prove $Q \vdash (C', \langle V, N'\rangle) : B \otimes C; Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash \langle V, N\rangle : B \otimes C$, so by Lemma 2.3.1 we get

$$\emptyset; Q_1'' \vdash V : B, \quad \emptyset; Q_1'' \vdash N : C,$$

for some B, C and Q_1'', Q_2'' such that $Q_1'', Q_2'' = Q''$. This entails $Q \vdash (C, N) : B; Q' \cup Q_1''$ and by inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' \vdash N' : B$ with $C' : Q \to Q' \cup Q_1'' \cup Q'''$. Finally, by the *tuple* rule we get $\emptyset; Q_1'', Q_2'' \vdash \langle V, N'\rangle : B \otimes C$ and conclude

$$Q \vdash (C', \langle V, N'\rangle) : B \otimes C; Q'.$$

- Case of *ctx-let*. Suppose $M \equiv \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P$, where N is not a value. We know $Q \vdash (C, \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P) : A; Q'$ and we must prove $Q \vdash (C', \mathsf{let}\,\langle x, y\rangle = N'\,\mathsf{in}\,P) : A; Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P : A$, so by Lemma 2.3.1 we get

$$\emptyset; Q_1'' \vdash N : B \otimes C, \quad x : B, y : C; Q_2'' \vdash P : A,$$

for some B, C and Q_1'', Q_2'' such that $Q_1'', Q_2'' = Q''$. This entails $Q \vdash (C, N) : B \otimes C; Q' \cup Q_2''$ and by inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' \vdash N' : B \otimes C$ with $C' : Q \to Q' \cup Q_2'' \cup Q'''$. Finally, by the *let* rule we get $\emptyset; Q_1'', Q_2'' \vdash \mathsf{let}\,\langle x, y\rangle = N'\,\mathsf{in}\,P : A$ and conclude

$$Q \vdash (C', \mathsf{let}\,\langle x, y\rangle = N'\,\mathsf{in}\,P) : A; Q'.$$

- Case of *ctx-force*. Suppose $M \equiv \mathsf{force}\,N$, where N is not a value. We know $Q \vdash (C, \mathsf{force}\,N) : A; Q'$ and we must prove $Q \vdash (C', \mathsf{force}\,N') : A; Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash \mathsf{force}\,N : A$, so by Lemma 2.3.1 we get

$$\emptyset; Q'' \vdash N : !A,$$

which entails $Q \vdash (C, N) : !A; Q'$. By inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' : N' : !A$, with $C' : Q \to Q' \cup Q'''$. Finally, by the *force* rule we get $\emptyset; Q''' \vdash \mathsf{force}\,N' : A$ and conclude

$$Q \vdash (C', \mathsf{force}\,N') : A; Q'.$$

- Case of *ctx-box*. Suppose $M \equiv \mathsf{box}\,N$, where N is not a value. We know $Q \vdash (C, \mathsf{box}_T\,N) : \mathsf{Circ}(T, U); Q'$ and we must prove $Q \vdash (C', \mathsf{box}_T\,N') : \mathsf{Circ}(T, U); Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash \mathsf{box}_T\,N : \mathsf{Circ}(T, U)$, so by Lemma 2.3.1 we get

$$\emptyset; Q'' \vdash N : !(T \multimap U),$$

which entails $Q \vdash (C, N) : !(T \multimap U); Q'$. By inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' \vdash N' : !(T \multimap U)$ with $C' : Q \to Q' \cup Q'''$. Finally, by the *box* rule we get $\emptyset; Q''' \vdash \mathsf{box}_T\,N' : \mathsf{Circ}(T, U)$ and conclude

$$Q \vdash (C', \mathsf{box}_T\,N') : \mathsf{Circ}(T, U) : Q'.$$

- Case of *ctx-apply-left*. Suppose $M \equiv \mathsf{apply}(N, P)$, where N is not a value. We know $Q \vdash (C, \mathsf{apply}(N, P)) : U; Q'$ and we must prove $Q \vdash (C', \mathsf{apply}(N', P)) : U; Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash \mathsf{apply}(N, P) : U$, so by Lemma 2.3.1 we get

$$\emptyset; Q''_1 \vdash N : \mathsf{Circ}(T, U), \quad \emptyset; Q''_2 \vdash P : T,$$

  for some T and Q''_1, Q''_2 such that $Q''_1, Q''_2 = Q''$. This entails $Q \vdash (C, N) : \mathsf{Circ}(T, U); Q' \cup Q''_2$ and by inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' \vdash N' : \mathsf{Circ}(T, U)$, with $C' : Q \to Q' \cup Q''_2 \cup Q'''$. Finally, by the *app* rule we get $\emptyset; Q''', Q''_2 \vdash \mathsf{apply}(N', P) : U$ and conclude

$$Q \vdash (C', \mathsf{apply}(N', P)) : U; Q'.$$

- Case of *ctx-apply-right*. Suppose $M \equiv \mathsf{apply}(V, N)$, where N is not a value. We know $Q \vdash (C, \mathsf{apply}(V, N)) : U; Q'$ and we must prove $Q \vdash (C', \mathsf{apply}(V, N')) : U; Q'$, where $(C, N) \to (C', N')$. We know $\emptyset; Q'' \vdash \mathsf{apply}(V, N) : U$, so by Lemma 2.3.1 we get

$$\emptyset; Q''_1 \vdash V : \mathsf{Circ}(T, U), \quad \emptyset; Q''_2 \vdash N : T,$$

  for some T and Q''_1, Q''_2 such that $Q''_1, Q''_2 = Q''$. This entails $Q \vdash (C, N) : T; Q' \cup Q''_1$ and by inductive hypothesis and Lemma 2.5.14 we get $\emptyset; Q''' \vdash N' : T$, with $C' : Q \to Q' \cup Q''_1 \cup Q'''$. Finally, by the *app* rule we get $\emptyset; Q''_1, Q''' \vdash \mathsf{apply}(V, N') : U$ and conclude

$$Q \vdash (C', \mathsf{apply}(V, N')) : U; Q'.$$

□

Progress

The progress result tells us that if a configuration is well-typed, then its evaluation can always be carried on by taking a further step either in the main reduction or in a sub-reduction introduced by a boxing operation. In simpler words, it tells us that well-typed configurations are never *deadlocked*. We already know (from Definition 2.5.5) what it means for a configuration to *go* into deadlock, and now we need to know what it means for a configuration to *be* in a deadlock already. Intuitively, this definition corresponds to cases 1, 3 and 4 of Definition 2.5.5, or more formally:

Definition 2.5.8 (Deadlock)**.** *Let $\mathcal{D}$ be the smallest set of configurations such that:*

1. *If (C, M) is irreducible and M is neither of the form $E[\mathsf{box}_T(\mathsf{lift}\, N)]$, nor a value, then $(C, M) \in \mathcal{D}$.*

2. *If M is of the form $E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $(id_Q, N\vec{\ell}) \to^* (D, N)$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$, and $(D, N) \in \mathcal{D}$, then $(C, M) \in \mathcal{D}$.*

3. If M is of the form $E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $(id_{Q_0}, N\vec{\ell}) \to^* (D, V)$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$, and V is not a label tuple, then $(C, M) \in \mathcal{D}$.

We say that a configuration (C, M) is deadlocked *when $(C, M) \in \mathcal{D}$.*

Intuitively, a configuration is deadlocked when it cannot be further reduced to a value or when it contains a boxing operation and the sub-reduction introduced by it ends up going into deadlock too. Note that this definition does not include the case in which such a sub-reduction diverges. In a sense, the following progress result asserts that well-typed configurations are safe from deadlock, but not from *livelock*.

Lemma 2.5.16. *Suppose (C, V) is a configuration. If V is not a label tuple, then there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash M : T$, for any simple M-type T.*

Proof. The proof is trivial and stems directly from the fact that simple M-types are just products of wire types and that labels are the only value that can be assigned a wire type. □

Theorem 2.5.17 (Progress)**.** *If $Q \vdash (C, M) : A; Q'$ then $(C, M) \notin \mathcal{D}$. That is, (C, M) is not deadlocked.*

Proof. By the definition of well-typedness we know that there exists Q'' disjoint from Q' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash M : A$. We therefore prove the contrapositive: if $(C, M) \in \mathcal{D}$, then there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash M : A$. We proceed by induction on the structure of the proof that $(C, M) \in \mathcal{D}$, distinguishing the case in which $M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, L)]$ from the case in which $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, L)]$.

- Case $M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, L)]$. In this case, M is necessarily not a value and there exist no C', M' such that $(C, M) \to (C', M')$. We must prove that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash M : A$ We proceed by induction on the form of M:
  - Case $M \equiv x$. This case is impossible, since by the definition of small-step configuration M must contain no free variables.
  - Case $M \equiv \ell$. This case is impossible since ℓ is a value.
  - Case $M \equiv \lambda x.N$. This case is impossible since $\lambda x.N$ is a value.
  - Case $M \equiv NP$. We know that (C, NP) is irreducible and NP is not of the form $E[\mathsf{box}_T(\mathsf{lift}\, L)]$ and we must prove that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash NP : A$. This would amount to finding Q''_1, Q''_2 such that $Q''_1, Q''_2 = Q''$ and
    $$\emptyset; Q''_1 \vdash N : B \multimap A, \quad \emptyset; Q''_2 \vdash P : B,$$
    for some B. By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, N) is irreducible and N is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$, so we distinguish two cases:

  - N is a value. In this case $N \equiv \lambda x.O$ for some x and O, by Lemma 2.3.2, and we consider P. By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, P) is irreducible and P is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. Also, P is not a value, since if that were the case then $(C, (\lambda x.O)P)$ would be reducible by the *β-reduction* rule. Therefore, by inductive hypothesis we get that there exists no Q_2'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_2'' \vdash P : B$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash NP : A$.
  - N is not a value. In this case, by inductive hypothesis we get that there exists no Q_1'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_1'' \vdash N : B \multimap A$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash NP : A$.

- Case $M \equiv \langle N, P\rangle$. We know that $(C, \langle N, P\rangle)$ is irreducible and $\langle N, P\rangle$ is neither a value, nor of the form $E[\mathsf{box}_T(\mathsf{lift}\, L)]$ and we must prove that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \langle N, P\rangle : B \otimes C$. This would amount to finding Q_1'', Q_2'' such that $Q_1'', Q_2'' = Q''$ and

$$\emptyset; Q_1'' \vdash N : B, \quad \emptyset; Q_2'' \vdash P : C,$$

  for some B, C. By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, N) is irreducible and N is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$, so we distinguish two cases:

  - N is a value. In this case we consider P. By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, P) is irreducible and P is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. Also, P is not a value, since if that were the case then $\langle N, P\rangle$ would be a value. Therefore, by inductive hypothesis we get that there exists no Q_2'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_2'' \vdash P : C$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \langle N, P\rangle : B \otimes C$.
  - N is not a value. In this case, by inductive hypothesis we get that there exists no Q_1'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_1'' \vdash N : B$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \langle N, P\rangle : B \otimes C$.

- Case $M \equiv \mathsf{let}\, \langle x, y\rangle = N \,\mathsf{in}\, P$. We know that $(C, \mathsf{let}\, \langle x, y\rangle = N \,\mathsf{in}\, P)$ is irreducible and $\mathsf{let}\, \langle x, y\rangle = N \,\mathsf{in}\, P$ is not of the form $E[\mathsf{box}_T(\mathsf{lift}\, L)]$ and we must prove that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{let}\, \langle x, y\rangle = N \,\mathsf{in}\, P : A$. This would amount to finding Q_1'', Q_2'' such that $Q_1'', Q_2'' = Q''$ and

$$\emptyset; Q_1'' \vdash N : B \otimes C, \quad x : B, y : C; Q_2'' \vdash P : A,$$

  for some B, C. By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, N) is irreducible and N is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. Also, N is not a value, since if that were the case then $(C, \mathsf{let}\, \langle x, y\rangle = N \,\mathsf{in}\, P)$ would be reducible by the *let* rule. Therefore, by inductive hypothesis we get that there exists no Q_1'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_1'' \vdash N : B \otimes C$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{let}\, \langle x, y\rangle = N \,\mathsf{in}\, P : A$.

– Case $M \equiv \mathsf{lift}\, N$. This case is impossible since $\mathsf{lift}\, N$ is a value.

– Case $M \equiv \mathsf{force}\, N$. We know that $(C, \mathsf{force}\, N)$ is irreducible and $\mathsf{force}\, N$ is not of the form $E[\mathsf{box}_T(\mathsf{lift}\, L)]$ and we must prove that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{force}\, N : A$. This would amount to finding Q'' such that

$$\emptyset; Q'' \vdash N : !A.$$

By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, N) is irreducible and N is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. Also, N is not a value, since if that were the case then we would have $N \equiv \mathsf{lift}\, L$ for some L, by Lemma 2.3.2, and $(C, \mathsf{force}(\mathsf{lift}\, L)) \to (C, L)$ by the *force* rule. Therefore, by inductive hypothesis we get that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash N : !A$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{force}\, N : A$

– Case $M \equiv \mathsf{box}_T\, N$. We know that $(C, \mathsf{box}_T\, N)$ is irreducible and $\mathsf{box}_T\, N$ is not of the form $E[\mathsf{box}_T(\mathsf{lift}\, L)]$ and we must prove that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{box}_T\, N : \mathsf{Circ}(T, U)$. This would amount to finding Q'' such that

$$\emptyset; Q'' \vdash N : !(T \multimap U).$$

By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, N) is irreducible and N is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. Also, N is not a value, since if that were the case then we would have $N \equiv \mathsf{lift}\, L$ for some L, by Lemma 2.3.2, and $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, L)]$ for $E \equiv [\cdot]$, which would contradict the hypothesis. Therefore, by inductive hypothesis we get that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash N : !(T \multimap U)$ and we conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{box}_T\, N : \mathsf{Circ}(T, U)$.

– Case $M \equiv \mathsf{apply}(N, P)$. We know that $(C, \mathsf{apply}(N, P))$ is irreducible and also that $\mathsf{apply}(N, P)$ is not of the form $E[\mathsf{box}_T(\mathsf{lift}\, L)]$ and we must prove that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{apply}(N, P) : U$. This would amount to finding Q''_1, Q''_2 such that $Q''_1, Q''_2 = Q''$ and

$$\emptyset; Q''_1 \vdash N : \mathsf{Circ}(T, U), \quad \emptyset; Q''_2 \vdash P : T,$$

for some T. By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, N) is irreducible and N is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$, so we distinguish two cases:

∗ N is a value. In this case $N \equiv (\vec{\ell}, D, \vec{\ell'})$ for some $\vec{\ell}, D$ and $\vec{\ell'}$, by Lemma 2.3.2, and we consider P. By corollaries 2.5.3.2 and 2.5.7.1 we know that (C, P) is irreducible and P is not of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. Also, P is not a value, since if that were the case then we would have $P \equiv \vec{k}$ for some $\vec{k}$, by Lemma 2.3.2, and $(C, (\vec{\ell}, D, \vec{\ell'}), \vec{k}) \to (C', \vec{k'})$ for $(C', \vec{k'}) = \mathsf{append}(C, \vec{k}, \vec{\ell}, D, \vec{\ell'})$, by the *apply* rule. Therefore, by inductive hypothesis we get that there exists no Q''_2 such that

$C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_2'' \vdash P : T$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{apply}(N, P) : U$.

  - N is not a value. In this case, by inductive hypothesis we get that there exists no Q_1'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_1'' \vdash N : \mathsf{Circ}(T, U)$ and we conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{apply}(N, P) : U$.

- Case $M \equiv (\vec{\ell}, D, \vec{\ell'})$. This case is impossible since $(\vec{\ell}, D, \vec{\ell'})$ is a value.

- Case $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, L)]$. In this case, we necessarily have $(id_{Q_0}, L\vec{\ell}) \to^* (C_n, M_n)$, for $(Q_0, \vec{\ell}) = \mathsf{freshlabels}(L, T)$, and either $(C_n, M_n) \in \mathcal{D}$ or M_n is a value other than a label tuple. We proceed by induction on the form of E:

  - Case $E \equiv [\cdot]$. In this case $M \equiv \mathsf{box}_T(\mathsf{lift}\, L)$. Suppose there exists Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{box}_T(\mathsf{lift}\, L) : \mathsf{Circ}(T, U)$. By applying Lemma 2.3.1 twice we get that $Q'' = \emptyset$ and

$$\emptyset; \emptyset \vdash \mathsf{lift}\, L : !(T \multimap U), \quad \emptyset; \emptyset \vdash L : T \multimap U.$$

    This entails $\emptyset; Q_0 \vdash L\vec{\ell} : U$, where U is a simple M-type, by the definition of $\mathsf{freshlabels}$ and the *app* rule. We know that $(id_{Q_0}, L\vec{\ell}) \to^* (C_n, M_n)$, that is, $(id_{Q_0}, L\vec{\ell})$ reduces to (C_n, M_n) in zero or more steps, so we distinguish two possibilities:

    - $(id_{Q_0}, L\vec{\ell}) = (C_n, M_n)$. In this case we necessarily have $(id_{Q_0}, L\vec{\ell}) \in \mathcal{D}$, since $L\vec{\ell}$ is not a value, so by the outer inductive hypothesis we know that there exists no Q_0'' such that $id_{Q_0} : Q_0 \to Q_0' \cup Q_0''$ and $\emptyset; Q_0'' \vdash L\vec{\ell} : B$, for any B, which contradicts $\emptyset; Q_0 \vdash L\vec{\ell} : U$.
    - $(id_{Q_0}, L\vec{\ell}) \to^+ (C_n, M_n)$. In this case, we know that $Q_0 \vdash (id_{Q_0}, L\vec{\ell}) : U; \emptyset$ and by a finite number of consecutive applications of Theorem 2.5.15 we get $Q_0 \vdash (C_n, M_n) : U; \emptyset$. This entails $C_n : Q_0 \to Q_0$ and $\emptyset; Q_0 \vdash M_n : U$. At the same time, we know that either $(C_n, M_n) \in \mathcal{D}$, and by the outer inductive hypothesis there exists no Q_0'' such that $C_n : Q_0 \to Q_0' \cup Q_0''$ and $\emptyset; Q_0'' \vdash M_n : B$, for any B, or M_n is a value other than a label tuple, and by Lemma 2.5.16 there exists no Q_0'' such that $C' : Q_0 \to Q_0' \cup Q_0''$ and $\emptyset; Q_0'' \vdash M_n : U$, for any simple M-type U. Both cases contradict $\emptyset; Q_0 \vdash M_n : U$.

    In either case, we reach a contradiction and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{box}_T(\mathsf{lift}\, L) : \mathsf{Circ}(T, U)$.

  - Case $E \equiv E'P$. In this case $M \equiv NP$ and N is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q_1'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_1'' \vdash N : B \multimap A$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash NP : A$.
  - Case $E \equiv VE'$. In this case $M \equiv VP$ and P is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q_2'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$

and $\emptyset; Q_2'' \vdash P : B$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash VP : A$.

- Case $E \equiv \langle E', P \rangle$. In this case $M \equiv \langle N, P \rangle$ and N is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q_1'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_1'' \vdash N : B$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \langle N, P \rangle : B \otimes C$.
- Case $E \equiv \langle V, E' \rangle$. In this case $M \equiv \langle V, P \rangle$ and P is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q_2'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_2'' \vdash P : C$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \langle V, P \rangle : B \otimes C$.
- Case $E \equiv \mathsf{let}\, \langle x, y \rangle = E'\, \mathsf{in}\, P$. In this case $M \equiv \mathsf{let}\, \langle x, y \rangle = N\, \mathsf{in}\, P$ and N is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q_1'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_1'' \vdash N : B \otimes C$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{let}\, \langle x, y \rangle = N\, \mathsf{in}\, P : A$.
- Case $E \equiv \mathsf{force}\, E'$. In this case $M \equiv \mathsf{force}\, N$ and N is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash N : !A$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{force}\, N : A$.
- Case $E \equiv \mathsf{box}_T\, E'$. In this case $M \equiv \mathsf{box}_T\, N$ and N is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash N : !(T \multimap U)$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{box}_T\, N : \mathsf{Circ}(T, U)$.
- Case $E \equiv \mathsf{apply}(E', P)$. In this case $M \equiv \mathsf{apply}(N, P)$ and N is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q_1'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_1'' \vdash N : \mathsf{Circ}(T, U)$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{apply}(N, P) : U$.
- Case $E \equiv \mathsf{apply}(V, E')$. In this case $M \equiv \mathsf{apply}(V, P)$ and P is of the form $E'[\mathsf{box}_T(\mathsf{lift}\, L)]$. By inductive hypothesis we get that there exists no Q_2'' such that $C : Q \to Q' \cup Q_1'' \cup Q_2''$ and $\emptyset; Q_2'' \vdash P : T$ and conclude that there exists no Q'' such that $C : Q \to Q' \cup Q''$ and $\emptyset; Q'' \vdash \mathsf{apply}(V, P) : U$.

□

2.5.5 Limitations of the Current Semantics

In this chapter we have introduced Proto-Quipper-M and we have given small-step semantics for its evaluation. We have shown that this semantics is equivalent to the original big-step semantics by Rios and Selinger and we have given the relevant subject reduction and progress results. The proposed small-step semantics constitutes a first step towards

our goal, which is devising some form of machine semantics for Proto-Quipper-M, but they fall short on one crucial detail: they are not truly small-step. This is due to the fact that the *box* reduction rule de-facto requires that a full evaluation $(id_Q, N\vec{\ell}) \to^* (D, \vec{\ell'})$, of arbitrary length, take place in its premises in order to compute the individual step $(C, \mathsf{box}_T(\mathsf{lift}\, N)) \to (C, (\vec{\ell}, D, \vec{\ell'}))$. Furthermore, unlike the other recursive rules of the semantics (the contextual ones), the *box* rule recurs on the term $N\vec{\ell}$, which is not a sub-term of $\mathsf{box}_T(\mathsf{lift}\, N)$. As a consequence, our small-step semantics have been harder-than-usual to reason about and on multiple occasions in this chapter we have had to consider the *box* case separately when giving definitions or proving results about the semantics (take, for example, Definition 2.5.5). In the next chapter, our first goal will be precisely that of altering the current semantics in order to obtain new, truly small-step semantics, which will serve as a stepping-stone to our final machine semantics.

Chapter 3

Towards a Machine Semantics

In this chapter we first introduce an intermediate semantics that solves the problems with the boxing operator that we encountered in the previous chapter. Then, we proceed to give the actual machine semantics for Proto-Quipper-M which is the objective of this thesis. We also give a number of definitions and results about the individual semantics, which will be useful in the next chapter, when we explore the relationship between the different semantics.

In his PhD thesis [14], Ross gives small-step semantics for Proto-Quipper (specifically, Proto-Quipper-S) and avoids our pitfall with the *box* rule by introducing a term of the form $(\vec{\ell}, D, M)$ and a contextual rule that allows to reduce $(C, (\vec{\ell}, D, M)) \to (C, (\vec{\ell}, D', M'))$ whenever $(D, M) \to (D', M')$. This approach introduces an implicit evaluation stack into the language, with every term of the form $(\vec{\ell}, D, M)$ conceptually representing an individual stack frame. This effectively avoids the problems we encountered with *box*, but it also comes with its own set of complications. For instance, despite the fact that terms of the form $(\vec{\ell}, D, M)$ are "intermediate forms" which are never meant to be written by the users of the language, every result that holds for terms in general must hold for $(\vec{\ell}, D, M)$ too.

It is mainly for this reason that we decide to take a different route. Specifically, instead of implicitly modelling a stack through the structure of the terms inside a configuration, we explicitly add a stack to the configurations themselves. At first, we only do it for the sub-reductions introduced by the *box* rule, in order to get a fully small-step semantics which we call a *stacked semantics*. Later, we extend this approach to all the contextual rules to obtain what we call a *machine semantics*.

3.1 Stacked Semantics

As the name suggests, the stacked semantics operate on *stacks of configurations*. Every time a term of the form $\mathsf{box}_T(\mathsf{lift}\, N)$ is ready to be evaluated, a new configuration $(id_Q, N\vec{\ell})$ is pushed on the stack and marked with the labels that are locally available to its evaluation (in this case, $\vec{\ell}$). When $(id_Q, N\vec{\ell})$ eventually evaluates to $(D, \vec{\ell'})$, the configuration is popped from the stack and $\mathsf{box}_T(\mathsf{lift}\, N)$ is replaced with $(\vec{\ell}, D, \vec{\ell'})$ in the previous stack frame.

Definition 3.1.1 (Stacked Configuration)**.** *A* stacked configuration *is given by the following grammar:*

$$X, Y ::= \epsilon \mid (C, M)^{\vec{\ell}}.X,$$

where C is a circuit, M is a term with no free variables and $\vec{\ell}$ is a label tuple, which can possibly be empty ($\vec{\ell} = \emptyset$).

Definition 3.1.2 (Well-formed Stacked Configuration)**.** *A stacked configuration is said to be* well-formed *when it is of the form $(C, M)^{\vec{\ell}}.X$ for some $C, M, \vec{\ell}$ and X and either one of the following conditions is met:*

1. *$\vec{\ell} = \emptyset$ and $X \equiv \epsilon$,*
2. *$\vec{\ell} \neq \emptyset$ and X is a well-formed stacked configuration.*

In this case a configuration of the form $(C, M)^{\emptyset}.\epsilon$ represents a situation in which no sub-reductions are being evaluated, and all the labels occurring in M are global (i.e. they were not introduced by a boxing operation). From this point onward we will assume that every stacked configuration we work with is well-formed. We can define a reduction relation $\rightharpoonup$ on stacked configurations, with the following rules:

$$\frac{(C, M) \to (D, N) \quad M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, P)]}{(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{\ell}}.X}\, head$$

$$\frac{(Q, \vec{\ell}) = \mathsf{freshlabels}(M, T)}{(C, E[\mathsf{box}_T(\mathsf{lift}\, M)])^{\vec{k}}.X \rightharpoonup (id_Q, M\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, M)])^{\vec{k}}.X}\, step\text{-}in$$

$$\frac{}{(D, \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, M)])^{\vec{k}}.X \rightharpoonup (C, E[(\vec{\ell}, D, \vec{\ell'})])^{\vec{k}}.X}\, step\text{-}out$$

where can clearly see that if a term does not contain a sub-term of the form $\mathsf{box}_T(\mathsf{lift}\, M)$ which is ready to be evaluated, then by definition the stacked semantics behave exactly like the small-step semantics, reducing the head configuration (the active stack frame).

It is when a term of the form $\mathsf{box}_T(\mathsf{lift}\, M)$ is ready to be evaluated that we part ways with the small-step semantics and we start taking advantage of the stack structure of these new configurations. Naturally, the reduction relation $\rightharpoonup$ is deterministic.

Lemma 3.1.1. *Every stacked configuration $(C, M)^{\vec{\ell}}.X$ can be reduced by at most one rule of the stacked semantics.*

Proof. If a configuration $(C, M)^{\vec{\ell}}.X$ can be reduced by the *step-out* rule, it means that M is a value. Therefore, $(C, M)^{\vec{\ell}}.X$ cannot be reduced by either *head* (because (C, V) is irreducible) or *step-in* (because V cannot be of the form $E[\mathsf{box}_T(\mathsf{lift}\, N)]$, for any E, N). At the same time, if $(C, M)^{\vec{\ell}}.X$ can be reduced by the *step-in* rule, it means that $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{\ell}}.X$ cannot be reduced by *head*. This is sufficient to conclude that at most one rule can be applied to $(C, M)^{\vec{\ell}}.X$. □

Proposition 3.1.2 (Determinism of Stacked Semantics)**.** *The reduction relation $\rightharpoonup$ is deterministic. That is, if $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.Y$, then for every stacked configuration $(D', N')^{\vec{k'}}.Y'$ such that $(C, M)^{\vec{\ell}}.X \rightharpoonup (D', N')^{\vec{k'}}.Y'$ we have $D = D', N \equiv N', k = k'$ and $Y = Y'$.*

Proof. We already know by Lemma 3.1.1 that at most one rule can be applied to reduce any given stacked configuration. What is left to do is prove that each rule is deterministic by itself, which is straightforward: the *head* rule is deterministic thanks to Proposition 2.5.2, while the *step-out* rule is deterministic because $\mathsf{freshlabels}$ is a function and *step-out* is trivially deterministic. We therefore conclude that $\rightharpoonup$ is deterministic. □

3.1.1 Initiality and Reachability

When reasoning about stacked configurations we must operate a necessary distinction between "starting" and "intermediate" configurations that we did not have to make with small-step configurations. Whereas in the small-step semantics we could expect a computation to start from any configuration (C, M), in the stacked semantics we are only interested in starting a computation from a configuration of the form $(C, M)^{\emptyset}.\epsilon$, where the stack is empty and all of the labels in M are global. Intuitively, a configuration of this form corresponds to the small-step configuration (C, M).

Definition 3.1.3 (Initial Stacked Configuration)**.** *A stacked configuration is said to be* initial *when it is of the form $(C, M)^{\emptyset}.\epsilon$. The set of initial stacked configurations is denoted by $\mathcal{I}_{st}$.*

We also distinguish between those stacked configurations which can be reached by a computation starting from an initial configuration and those which cannot. For example, a configuration of the form $(C, \lambda x.x)^{\vec{\ell}}.(D, \lambda x.x)^{\emptyset}.\epsilon$, although well-formed, is clearly

impossible to obtain during the evaluation of an initial configuration, since new stack frames are only introduced when a term containing a boxing operator is encountered. As a result, configurations such as this one are ill-natured in their own way. We therefore give the definition of *reachable stacked configuration*.

Definition 3.1.4 (Reachable Stacked Configuration)**.** *A stacked configuration of the form* $(C, M)^{\vec{\ell}}.X$ *is said to be* reachable *when either of the following is true:*

1. $(C, M)^{\vec{\ell}}.X \in \mathcal{I}_{st}$,
2. *There exists a stacked configuration* $(D, N)^{\vec{k}}.Y$ *such that* $(D, N)^{\vec{k}}.Y$ *is reachable and* $(D, N)^{\vec{k}}.Y \rightharpoonup (C, M)^{\vec{\ell}}.X$.

This distinction between reachable and unreachable configurations will be essential when we investigate the relationship between the stacked semantics and the other semantics.

3.1.2 Convergence, Deadlock and Divergence

Like we did with small-step configurations, we define what it means for a stacked configuration to converge, go into deadlock or diverge.

Definition 3.1.5 (Converging Stacked Configuration)**.** *Let* $\downarrow$ *be the smallest unary relation over stacked configurations such that:*

1. *For every circuit* C *and value* V, $(C, V)^{\emptyset}.\epsilon \downarrow$,
2. *If* $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.Y$ *and* $(D, N)^{\vec{k}}.Y \downarrow$, *then* $(C, M)^{\vec{\ell}}.X \downarrow$.

We say that a configuration $(C, M)^{\vec{\ell}}.X$ *is* converging *when* $(C, M)^{\vec{\ell}}.X \downarrow$.

Definition 3.1.6 (Deadlocking Stacked Configuration)**.** *Let* $\perp$ *be the smallest unary relation over stacked configurations such that:*

1. *If there exists no* $(D, N)^{\vec{k}}.Y$ *such that* $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.Y$ *and either* M *is not a value or* $\vec{\ell} \neq \emptyset, X \neq \epsilon$, *then* $(C, M)^{\vec{\ell}}.X\perp$,
2. *If* $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.Y$ *and* $(D, N)^{\vec{k}}.Y\perp$, *then* $(C, M)^{\vec{\ell}}.X\perp$.

We say that a configuration $(C, M)^{\vec{\ell}}.X$ goes into deadlock *when* $(C, M)^{\vec{\ell}}.X\perp$.

Definition 3.1.7 (Diverging Stacked Configuration)**.** *Let* $\uparrow$ *be the largest unary relation over stacked configurations such that whenever* $(C, M)^{\vec{\ell}}.X \uparrow$ *there exists* $(D, N)^{\vec{k}}.Y$ *such that* $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.Y$ *and* $(D, N)^{\vec{k}}.Y \uparrow$. *We say that a configuration* $(C, M)^{\vec{\ell}}.X$ *is* diverging *when* $(C, M)^{\vec{\ell}}.X \uparrow$.

Note that although the intuition behind the concepts of convergence, deadlock and divergence is (obviously) still the same, the respective definitions for stacked configurations are much simpler than their small-step counterparts. This is an effect of the "inlining" of the box sub-reductions that we operated when defining the stacked semantics, which allows us to treat the boxing rules (*step-in* and *step-out*) homogeneously with the rest of the rules. Naturally, the three relations are still mutually exclusive.

Proposition 3.1.3. *The relations $\downarrow, \bot$ and $\uparrow$ are mutually exclusive over stacked configurations. That is, for every stacked configuration $(C, M)^{\vec{\ell}}.X$, the following are true:*

1. *If $(C, M)^{\vec{\ell}}.X \downarrow$, then $(C, M)^{\vec{\ell}}.X \not\perp$,*
2. *If $(C, M)^{\vec{\ell}}.X \downarrow$, then $(C, M)^{\vec{\ell}}.X \not\uparrow$,*
3. *If $(C, M)^{\vec{\ell}}.X \bot$, then $(C, M)^{\vec{\ell}}.X \not\uparrow$.*

Proof. We prove each claim separately:

1. We proceed by induction on $(C, M)^{\vec{\ell}}.X \downarrow$:
   - Case of $M \equiv V, \vec{\ell} = \emptyset$ and $X = \epsilon$. Since $(C, V)^{\emptyset}.\epsilon$ is irreducible, but V is a value, there is no way for $(C, V)^{\emptyset}.\epsilon$ to go into deadlock, so we conclude $(C, V)^{\emptyset}.\epsilon \not\perp$.
   - Case of $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.X'$ and $(D, N)^{\vec{k}}.X' \downarrow$. Since $(C, M)^{\vec{\ell}}.X$ is reducible, it must be that $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.X'$ ($\rightharpoonup$ is deterministic) and $(D, N)^{\vec{k}}.X' \bot$ in order for $(C, M)^{\vec{\ell}}.X$ to go into deadlock. However, by inductive hypothesis we know that $(D, N)^{\vec{k}}.X' \not\perp$, so this is impossible and we conclude $(C, M)^{\vec{\ell}}.X \not\perp$.
2. We proceed by induction on $(C, M)^{\vec{\ell}}.X \downarrow$:
   - Case of $M \equiv V, \vec{\ell} = \emptyset$ and $X = \epsilon$. Since $(C, V)^{\emptyset}.\epsilon$ is irreducible, there is no way for it to diverge, so we conclude $(C, V)^{\emptyset}.\epsilon \not\uparrow$.
   - Case of $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.X'$ and $(D, N)^{\vec{k}}.X' \downarrow$. Since $(D, N)^{\vec{k}}.X' \not\uparrow$ by inductive hypothesis and $(C, M)^{\vec{\ell}}.X$ cannot reduce to any other configuration ($\rightharpoonup$ is deterministic), there is no way for $(C, M)^{\vec{\ell}}.X$ to diverge, so we conclude $(C, M)^{\vec{\ell}}.X \not\uparrow$.
3. We proceed by induction on $(C, M)^{\vec{\ell}}.X \bot$:
   - Case in which $(C, M)^{\vec{\ell}}.X$ is irreducible and either $M \not\equiv V$ or $\vec{\ell} \neq \emptyset$ or $X \neq \epsilon$. Since $(C, M)^{\vec{\ell}}.X$ is irreducible, there is no way for it to diverge, so we conclude $(C, M)^{\vec{\ell}}.X \not\uparrow$.

- Case of $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{k}}.X'$ and $(D, N)^{\vec{k}}.X'\bot$. Since $(D, N)^{\vec{k}}.X' \not\downarrow$ by inductive hypothesis and $(C, M)^{\vec{\ell}}.X$ cannot reduce to any other configuration ($\rightharpoonup$ is deterministic), there is no way for $(C, M)^{\vec{\ell}}.X$ to diverge, so we conclude $(C, M)^{\vec{\ell}}.X \not\downarrow$.

□

We now give two specific result about the convergence (or lack thereof) of stacked configurations. First, we prove that whenever a stacked configuration $(C, M)^{\vec{\ell}}.X$ is reachable and converges, then the rest of the stack X (which is itself a stacked configuration) also converges. Note that, in the following proof, the *length* of a stacked configuration is, naturally, the size of its stack.

Lemma 3.1.4. *If $(C, M)^{\vec{\ell}}.X$ is reachable, $(C, M)^{\vec{\ell}}.X \downarrow$ and $X \neq \epsilon$, then $X \downarrow$.*

Proof. $(C, M)^{\vec{\ell}}.X$ is a stacked configuration of length $n \geq 2$. The base case for reachable stacked configurations has length one. This implies that in order for $(C, M)^{\vec{\ell}}.X$ to be reachable there must be at least one lengthening reduction somewhere between some $(D, N)^{\emptyset}.\epsilon \in \mathcal{I}_{st}$ and $(C, M)^{\vec{\ell}}.X$. Because lengthening reductions are only introduced by *step-in*, we must have $X = (D', E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{k}}.X'$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(T, P)$, and

$$\begin{aligned}(D, N)^{\emptyset}.\epsilon &\rightharpoonup^* (D', E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{k}}.X' \\ &\rightharpoonup (id_Q, P\vec{\ell})^{\vec{\ell}}.(D', E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{k}}.X' \\ &\rightharpoonup^* (C, M)^{\vec{\ell}}.(D', E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{k}}.X'.\end{aligned}$$

This entails $(D', E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{k}}.X' \rightharpoonup^+ (C, M)^{\vec{\ell}}.(D', E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{k}}.X'$ and because we already know $(C, M)^{\vec{\ell}}.(D', E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{k}}.X' \downarrow$ by hypothesis, we conclude $(D', E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{k}}.X' \downarrow$. □

Next, conversely, we show that if a stacked configuration goes into deadlock on its own, then augmenting it with an additional underlying stack cannot solve the deadlock.

Lemma 3.1.5. *Let $+_{\vec{k}}$ be a binary function which represents the concatenation of stacked configurations, defined as follows:*

$$\begin{aligned}(C, M)^{\emptyset}.\epsilon +_{\vec{k}} Y &= (C, M)^{\vec{k}}.Y, \\ (C, M)^{\vec{\ell}}.X +_{\vec{k}} Y &= (C, M)^{\vec{\ell}}.(X +_{\vec{k}} Y).\end{aligned}$$

If $(C, M)^{\vec{\ell}}.X\bot$, then $(C, M)^{\vec{\ell}}.X +_{\vec{k}} Y\bot$ for every $\vec{k}$ and $Y \neq \epsilon$.

Proof. We proceed by induction on $(C, M)^{\vec{\ell}}.X\bot$:

- Case in which $(C, M)^{\vec{\ell}}.X$ is irreducible and either $M \not\equiv V$ or $\vec{\ell} \neq \emptyset, X \neq \epsilon$. If $(C, M)^{\vec{\ell}}.X +_{\vec{k}} Y$ were reducible by either the *head* or *step-in* rule, then so would $(C, M)^{\vec{\ell}}.X$, since both *head* and *step-in* can be applied regardless of the local labels or the rest of the stack. This contradicts the hypothesis, so we know that $(C, M)^{\vec{\ell}}.X +_{\vec{k}} Y$ cannot be reduced by these rules. Furthermore, if $(C, M)^{\vec{\ell}}.X +_{\vec{k}} Y$ were reducible via the *step-out* rule, we would have $M \equiv \vec{\ell'}$ for some $\vec{\ell'}$ and as a consequence $\vec{\ell} \neq \emptyset$ and $X \neq \epsilon$ (otherwise we would contradict the hypothesis that either $M \not\equiv V$ or $\vec{\ell} \neq \emptyset, X \neq \epsilon$). We would therefore have $X = (D, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X'$ for some $D, E, N, \vec{k}$ and X'. This would entail the reducibility of $(C, \vec{\ell'})^{\vec{\ell}}.(D, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X'$, which contradicts the hypothesis, so $(C, M)^{\vec{\ell}}.X +_{\vec{k}} Y$ is ultimately irreducible and because $\vec{\ell} \neq \emptyset, X \neq \epsilon$ and $Y \neq \epsilon$ we trivially conclude $(C, M)^{\vec{\ell}}.X +_{\vec{k}} Y\bot$.

- Case of $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{\ell'}}.X'$ and $(D, N)^{\vec{\ell'}}.X'\bot$. We proceed by cases on the introduction of $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{\ell'}}.X'$:

  - Case of *head*. If $(C, M)^{\vec{\ell}}.X \rightharpoonup (D, N)^{\vec{\ell}}.X$, then because the *head* rule can be applied regardless of the local labels and the remaining stack we also have $(C, M)^{\vec{\ell}}.X +_{\vec{k}} Y \rightharpoonup (D, N)^{\vec{\ell}}.X +_{\vec{k}} Y$ by the same rule. By inductive hypothesis we get that $(D, N)^{\vec{\ell}}.X +_{\vec{k}} Y\bot$ and conclude $(C, M)^{\vec{\ell}}.X +_{\vec{k}} Y\bot$.
  - Case of *step-in*. If $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, P)]$ for some E, P and $(C, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell}}.X \rightharpoonup (id_Q, P\vec{\ell'})^{\vec{\ell'}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell}}.X$ for $(Q, \vec{\ell'}) = \mathsf{freshlabels}(P, T)$, then because the *step-in* rule can be applied regardless of the local labels and the remaining stack we also have $(C, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell}}.X +_{\vec{k}} Y \rightharpoonup (id_Q, P\vec{\ell'})^{\vec{\ell'}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell}}.X +_{\vec{k}} Y$ by the same rule. By inductive hypothesis we get $(id_Q, P\vec{\ell'})^{\vec{\ell'}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell}}.X +_{\vec{k}} Y\bot$ and conclude $(C, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell}}.X +_{\vec{k}} Y\bot$.
  - Case of *step-out*. In this case we necessarily have $\vec{\ell} \neq \emptyset, X \neq \epsilon$. If $M \equiv \vec{k'}$ for some $\vec{k'}$, $X = (D, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell'}}.X'$ for some E, P and $(C, \vec{k'})^{\vec{\ell}}.(D, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell'}}.X' \rightharpoonup (D, E[(\vec{\ell}, C, \vec{k'})])^{\vec{\ell'}}.X'$, then because $+_{\vec{k}}$ can only alter $\vec{\ell'}$ and X' and the *step-out* rule does not depend on them, we also have $(C, \vec{k'})^{\vec{\ell}}.(D, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell'}}.X' +_{\vec{k}} Y \rightharpoonup (D, E[(\vec{\ell}, C, \vec{k'})])^{\vec{\ell'}}.X' +_{\vec{k}} Y$ by the same rule. By inductive hypothesis we get that $(D, E[(\vec{\ell}, C, \vec{k'})])^{\vec{\ell'}}.X' +_{\vec{k}} Y \not\perp$ and conclude $(C, \vec{k'})^{\vec{\ell}}.(D, E[\mathsf{box}_T(\mathsf{lift}\, P)])^{\vec{\ell'}}.X' +_{\vec{k}} Y\bot$.

□

3.1.3 Summary

In this section we introduced the concept of stacked configuration and we based a deterministic stacked semantics on it. This semantics effectively circumvents the problem

that we encountered with *box* in the previous small-step semantics. We then established the subset of "benign" stacked configurations in which we are actually interested, namely well-formed and reachable configurations. Lastly, we gave definitions for the convergence, deadlock and divergence of stacked configurations, which we proved to be mutually exclusive relations, similarly to how we did with small-step configurations in Chapter 2.

3.2 Machine Semantics

In this section we finally introduce a machine operational semantics for Proto-Quipper-M, which is the focus of this thesis. We call this a *machine* semantics because it is heavily inspired by the concept of *abstract machine*, which is something we ought to touch upon.

3.2.1 Abstract Machines

In computer science, an abstract machine is simply a theoretical model of a computer. To this effect, *computer* is to be intended in the most abstract way possible, that is, something that computes. In fact, it is irrelevant whether the theoretical model is actually implementable in hardware, as long as it describes a computation the way a realistic mechanical computer would carry it out (read: algorithmically). Usually, this is done by means of a state transition system. Because an abstract machine allows us to know not only *what* a program written in a given programming language evaluates to, but also *how* exactly it is evaluated, abstract machines are often employed to define the semantics of programming languages.

In the context of lambda-calculi, an abstract machine specification does two main things. The first one is fixing an evaluation strategy. For instance, whereas the basic semantics for the lambda-calculus leave the door open for both call-by-value and call-by-name strategies, any individual abstract machine must commit to *either* a call-by-value *or* a call-by-name strategy. The second thing an abstract machine does is provide a concrete algorithm to carry out some of the operations which are otherwise assumed to be elementary, such as the substitution of values for variables within a term, or the exploration of a term in search of a redex. A number of abstract machines already exist that formalize the evaluation of a lambda-term. Notable examples include the SECD machine [9], which implements a call-by-value semantics and is based on multiple evaluation stacks, the Krivine machine [8], which implements a call-by-name semantics, and the CEK machine [4], which also implements a call-by-value semantics, but using continuations rather than a stack. We describe briefly this last machine, since it is the one that most inspired our machine semantics for Proto-Quipper-M. The CEK machine

takes its name from the shape of its states, which are triples of the form

$$\langle C, E, K \rangle,$$

where C is called *control* and corresponds to the term currently being evaluated, E is the *environment*, that is, an associative array from variable names to values, and K is the *continuation* and represents the next action to perform once C has been fully evaluated. For the sake of this presentation, we use the terms of the basic untyped lambda-calculus, that is:

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

The only values of this language are closures, that is, pairs $\{\lambda x.M, E\}$ of abstractions together with their definition environment. Therefore, the environment can be seen as nothing more than a list of bindings of the form

$$x \mapsto \{\lambda y.M, E\}.$$

When the control is a single variable name, we look that variable up in the environment to obtain the corresponding closure, whose abstraction becomes the new control and whose environment becomes the new environment. This is formalized by the following rule:

$$\frac{\{\lambda x.M, E'\} = \mathsf{lookup}(x, E)}{\langle x, E, K \rangle \to \langle \lambda x.M, E', K \rangle} \mathit{var}$$

where $\mathsf{lookup}(x, E)$ finds the first occurrence of x in E and returns the corresponding closure. When the control is an application MN, we start by evaluating M to an abstraction. While we do so, we must remember that after we are done we must proceed to evaluate N. This is where continuations come into play. A continuation of the form $\mathit{FArg}(N, E, K)$ represents a reminder that after we are done evaluating the current control (whatever it might be), we should start evaluating N in the environment E, and then proceed in a similar fashion with continuation K. The rule for evaluating applications is thus the following:

$$\frac{}{\langle MN, E, K \rangle \to \langle M, E, \mathit{FArg}(N, E, K) \rangle} \mathit{split}$$

Once we are done evaluating M to a term of the form $\lambda x.P$, we can start evaluating N. Here we find ourselves in a symmetric situation: as we evaluate N we must remember that after we are done we must apply $\lambda x.P$ to the result. Since functions are represented as closures, this reminder is represented by a continuation of the form $\mathit{FApp}(\lambda x.P, E, K)$, where E is the environment in which $\lambda x.P$ was defined. We therefore introduce a third rule to our CEK machine:

$$\frac{}{\langle \lambda x.P, E, \mathit{FArg}(N, E', K) \rangle \to \langle N, E', \mathit{FApp}(\lambda x.P, E, K) \rangle} \mathit{shift}$$

Lastly, once N has also been evaluated to an abstraction $\lambda y.L$, we can finally apply $\lambda x.P$ to $\lambda y.L$. Concretely, this means that we start evaluating P under the environment E in which $\lambda x.P$ was defined, with the additional binding of x to $\lambda y.L$. With the following rule, the CEK machine is complete:

$$\frac{}{\langle \lambda y.L, E', \mathit{FApp}(\lambda x.P, E, K)\rangle \to \langle P, (x \mapsto \{\lambda y.L, E'\}) :: E, K\rangle}\mathit{join}$$

where :: denotes the concatenation of environments. To see more clearly how these rules interact with each other in order to reduce a term, consider the evaluation of $(\lambda x.xx)(\lambda y.y)$ to $\lambda y.y$.

$$\begin{aligned}
\langle (\lambda x.xx)(\lambda y.y), [], \mathit{Done}\rangle &\to \langle \lambda x.xx, [], \mathit{FArg}(\lambda y.y, [], \mathit{Done})\rangle && \mathit{split}\\
&\to \langle \lambda y.y, [], \mathit{FApp}(\lambda x.xx, [], \mathit{Done})\rangle && \mathit{shift}\\
&\to \langle xx, [x \mapsto \{\lambda y.y, []\}], \mathit{Done}\rangle && \mathit{join}\\
&\to \langle x, [x \mapsto \{\lambda y.y, []\}], \mathit{FArg}(x, [x \mapsto \{\lambda y.y, []\}], \mathit{Done})\rangle && \mathit{split}\\
&\to \langle \lambda y.y, [], \mathit{FArg}(x, [x \mapsto \{\lambda y.y, []\}], \mathit{Done})\rangle && \mathit{var}\\
&\to \langle x, [x \mapsto \{\lambda y.y, []\}], \mathit{FApp}(\lambda y.y, [], \mathit{Done})\rangle && \mathit{shift}\\
&\to \langle \lambda y.y, [], \mathit{FApp}(\lambda y.y, [], \mathit{Done})\rangle && \mathit{var}\\
&\to \langle y, [y \mapsto \{\lambda y.y, []\}], \mathit{Done}\rangle && \mathit{join}\\
&\to \langle \lambda y.y, [], \mathit{Done}\rangle. && \mathit{var}
\end{aligned}$$

3.2.2 An Abstract Machine for Proto-Quipper-M

As we anticipated earlier, our machine semantics for Proto-Quipper-M is largely influenced by the CEK machine. In particular, we retain the use of continuations as a means to schedule the various phases of the evaluation of a term, although we organize them in a stack rather than one within the other. Note that this change is purely syntactic, since in practice the continuations of the CEK machine already recursively define a stack. Another difference between the CEK machine and our semantics is that for the sake of simplicity we keep relying on the substitution function $M[N/x]$ to evaluate function applications instead of employing environments to the same effect. This choice does not fundamentally detract from the the results that we prove in this thesis, which are expected to hold for any sensible explicit implementation of substitutions. We just leave such an implementation as future work. In conclusion, our *machine configurations* are composed of a circuit, a term which builds the circuit, and a stack.

Definition 3.2.1 (Machine Configuration)**.** *A* machine configuration *is a triple*

$$(C, M, S),$$

where C is a circuit, M is a term with no free variables and S is a stack*, which is defined by the following grammar:*

$$\begin{array}{llll} \text{Stack elements} & H & ::= & \mathit{FArg}(M) \mid \mathit{FApp}(V) \\ & & & \mid \mathit{ALabel}(M) \mid \mathit{ACirc}(V) \\ & & & \mid \mathit{TRight}(M) \mid \mathit{TLeft}(V) \\ & & & \mid \mathit{Box}(Q, \vec{\ell}) \mid \mathit{Sub}(C, M, \vec{\ell}, T) \\ & & & \mid \mathit{Let}(x, y, M) \mid \mathit{Force}, \\ \text{Stacks} & S, R & ::= & \epsilon \mid H.S. \end{array}$$

It is worth noting that although stacked configuration and machine configurations may appear very different at first glance, they are based on the common intuition that a computation is easily modelled by a stack. The only difference between the two is the extent to which we apply this intuition: whereas with stacked configurations we only push a frame onto the stack when evaluating an entirely new configuration as part of a boxing operation (while still reducing everything else "in place"), with machine configurations we push a frame onto the stack every time we encounter a composite term. Let $\Rightarrow$ be a reduction relation for machine configurations. We give the following rules:

$$\frac{}{(C, MN, S) \Rightarrow (C, M, \mathit{FArg}(N).S)}\,\textit{app-split}$$

$$\frac{}{(C, V, \mathit{FArg}(N).S) \Rightarrow (C, N, \mathit{FApp}(V).S)}\,\textit{app-shift}$$

$$\frac{}{(C, V, \mathit{FApp}(\lambda x.M).S) \Rightarrow (C, M[V/x], S)}\,\textit{app-join}$$

$$\frac{}{(C, \mathsf{apply}(M, N), S) \Rightarrow (C, M, \mathit{ALabel}(N).S)}\,\textit{apply-split}$$

$$\frac{}{(C, V, \mathit{ALabel}(N).S) \Rightarrow (C, N, \mathit{ACirc}(V).S)}\,\textit{apply-shift}$$

$$\frac{(C', \vec{k'}) = \mathsf{append}(C, \vec{k}, \vec{\ell}, D, \vec{\ell'})}{(C, \vec{k}, \mathit{ACirc}(\vec{\ell}, D, \vec{\ell'}).S) \Rightarrow (C', \vec{k'}, S)}\,\textit{apply-join}$$

$$\frac{\langle M, N\rangle \text{ is not a value}}{(C, \langle M, N\rangle, S) \Rightarrow (C, M, \mathit{TRight}(N).S)}\,\textit{tuple-split}$$

$$\frac{}{(C, V, \mathit{TRight}(N).S) \Rightarrow (C, N, \mathit{TLeft}(V).S)}\,\textit{tuple-shift}$$

$$\frac{}{(C, W, \mathit{TLeft}(V).S) \Rightarrow (C, \langle V, W\rangle, S)}\,\textit{tuple-join}$$

$$\frac{(Q, \vec{\ell}) = \mathsf{freshlabels}(M, T)}{(C, \mathsf{box}_T\, M, S) \Rightarrow (C, M, \mathit{Box}(Q, \vec{\ell}).S)}\,\textit{box-open}$$

$$\frac{\emptyset; Q \vdash \vec{\ell} : T}{(C, \mathsf{lift}\, M, \mathit{Box}(Q, \vec{\ell}).S) \Rightarrow (\mathit{id}_Q, M\vec{\ell}, \mathit{Sub}(C, M, \vec{\ell}, T).S)}\,\textit{box-sub}$$

$$\frac{}{(D, \vec{\ell'}, \mathit{Sub}(C, M, \vec{\ell}, T).S) \Rightarrow (C, (\vec{\ell}, D, \vec{\ell'}), S)}\,\textit{box-close}$$

$$\frac{}{(C, \mathsf{let}\, \langle x, y\rangle = M\, \mathsf{in}\, N, S) \Rightarrow (C, M, \mathit{Let}(x, y, N).S)}\,\textit{let-split}$$

$$\frac{}{(C, \langle V, W\rangle, \mathit{Let}(x, y, M).S) \Rightarrow (C, M[V/x][W/y], S)}\,\textit{let-join}$$

$$\frac{}{(C, \mathsf{force}\, M, S) \Rightarrow (C, M, \mathit{Force}.S)}\,\textit{force-open}$$

$$\frac{}{(C, \mathsf{lift}\, M, \mathit{Force}.S) \Rightarrow (C, M, S)}\,\textit{force-close}$$

In light of the previous exposition of the rules for the CEK machine, the rules for the machine semantics should be self-explanatory. Generally, every binary term constructor (such as the application or the tuple) has an associated *split* rule, which defines how a term is split into smaller sub-terms and which is evaluated first, a *shift rule*, which defines how and when we switch to evaluating the second term, and a *join* rule, which defines the way the results of the two sub-terms are put together to obtain the result of the whole term. To keep track of what to do next, these rules employ two kinds of continuations, one that keeps track of the right sub-term while the left one is being evaluated (like *FArg*) and one that does the opposite (like *FApp*). The case of unary constructors (such as box_T or force) is similar, although the kind of information that is stored on the stack in this case varies. For example, the *Sub* continuation, which roughly corresponds to a stack frame of the stacked semantics, has to store the entire circuit and term whose evaluation was temporarily interrupted by the boxing operation,

as well as the new local labels introduced by it and their associated type (the latter for bookkeeping reasons which will be clear in the next chapter). On the other hand, the *Force* continuation does not need to store any additional information, since all the force operator does is undo the lifting of a term.

Notice how the definition of the machine semantics effectively allows us to relinquish the notion of evaluation context. Whereas in the small-step and stacked semantics we sometimes had to reason about *where* a reduction occurred within a term (see, as an example, the *step-in* rule of the stacked semantics), in the machine semantics we always reduce a redex whose components are immediately available in the term component and on top of the stack. If a redex is not immediately available, the term being evaluated is broken down into smaller pieces, and this decomposition operation is an integral part of the semantics. As a result, the "descent" into a term in search of a redex is no longer implicit in the derivation of an individual step of the reduction relation, but rather it is explicit in the reduction sequence itself.

Naturally, we expect the machine semantics to be deterministic, like the small-step and the stacked semantics. We prove that this is the case with the following results.

Lemma 3.2.1. *Every machine configuration (C, M, S) can be reduced by at most one rule of the machine semantics.*

Proof. We first partition the rules into two sets: one containing the rules that require M to be a value (*app-shift, app-join, apply-shift, apply-join, tuple-shift, tuple-join, box-sub, box-close, let-join* and *force-close*) and the other containing the rules that require M not to be a value (*app-split, apply-split, tuple-split, box-open, let-split, force-open*). Naturally, the applicability of a rule from the first set to any given configuration excludes the possibility of applying any rule from the second set to the same configuration, and vice-versa. Consider now the first set. Each of the rules contained in this set requires a different stack head in order to be applied, so at most one rule from the first set can be applied to any given configuration. Consider now the second set. Each of the rules contained in this set requires a different shape of M in order to be applied, so at most one rule from the second set can be applied to any given configuration. We conclude that every machine configuration (C, M, S) can be reduced by at most one rule of the machine semantics. □

Proposition 3.2.2 (Determinism of Machine Semantics)**.** *The reduction relation $\Rightarrow$ is deterministic. That is, if $(C, M, S) \Rightarrow (D, N, R)$, then for every stacked configuration (D', N', R') such that $(C, M, S) \Rightarrow (D', N', R')$ we have $D = D', N \equiv N'$ and $R = R'$.*

Proof. We already known by Lemma 3.2.1 that at most one rule can be applied to reduce any given machine configuration. What is left to do is prove that each rule is

deterministic by itself. The proof is trivial, since substitution, append, freshlabels and the typing judgement are all functions. □

3.2.3 Initiality and Reachability

The same reasoning about what kind of configuration we can start a computation from that we made for the stacked semantics can (and must) be made for the machine semantics. The following definitions are not fundamentally different from the corresponding definitions that we gave for stacked configurations.

Definition 3.2.2 (Initial Machine Configuration)**.** *A machine configuration is said to be* initial *when it is of the form* (C, M, ϵ)*. The set of initial machine configurations is denoted by* $\mathcal{I}_{ma}$*.*

Definition 3.2.3 (Reachable Machine Configuration)**.** *A machine configuration of the form* (C, M, S) *is said to be* reachable *when either of the following is true:*

1. $(C, M, S) \in \mathcal{I}_{ma}$*,*
2. *There exists a machine configuration* (D, N, R) *such that* (D, N, R) *is reachable and* $(D, N, R) \Rightarrow (C, M, S)$*.*

3.2.4 Convergence, Deadlock and Divergence

We also give the usual definitions of convergence, deadlock and divergence.

Definition 3.2.4 (Converging Machine Configuration)**.** *Let* $\downarrow$ *be the smallest unary relation over machine configurations such that:*

1. *For every circuit* C *and value* V*,* $(C, V, \epsilon)\downarrow$*,*
2. *If* $(C, M, S) \Rightarrow (D, N, R)$ *and* $(D, N, R)\downarrow$*, then* $(C, M, S)\downarrow$*.*

We say that a configuration (C, M, S) *is* converging *when* $(C, M, S)\downarrow$*.*

Definition 3.2.5 (Deadlocking Machine Configuration)**.** *Let* $\perp$ *be the smallest unary relation over machine configurations such that:*

1. *If there exists no* (D, N, R) *such that* $(C, M, S) \Rightarrow (D, N, R)$ *and* $S \neq \epsilon$*, then* $(C, M, S)\perp$*,*
2. *If* $(C, M, S) \Rightarrow (D, N, R)$ *and* $(D, N, R)\perp$*, then* $(C, M, S)\perp$*.*

We say that a configuration (C, M, S) goes into deadlock *when* $(C, M, S)\perp$*.*

Definition 3.2.6 (Diverging Machine Configuration)**.** *Let $\uparrow$ be the largest unary relation over machine configurations such that whenever $(C, M, S) \uparrow$ there exists (D, N, R) such that $(C, M, S) \Rightarrow (D, N, R)$ and $(D, N, R) \uparrow$. We say that a configuration (C, M, S) is* diverging *when $(C, M, S) \uparrow$.*

Notice how similar these definitions are to the corresponding definitions given in Section 3.1.2 for stacked configurations, and how different they are from the corresponding definitions given in Section 2.5.2 for small-step configurations. This is further proof of what we briefly mentioned previously, that is, that the stacked semantics and the machine semantics are built on the same intuition. Like we did with the previous semantics, we prove that the convergence, deadlock and divergence relations are mutually exclusive on machine configurations.

Proposition 3.2.3. *The relations $\downarrow, \bot$ and $\uparrow$ are mutually exclusive over machine configurations. That is, for every machine configuration (C, M, S), the following are true:*

1. *If $(C, M, S) \downarrow$, then $(C, M, S) \not\perp$,*
2. *If $(C, M, S) \downarrow$, then $(C, M, S) \not\uparrow$,*
3. *If $(C, M, S)\bot$, then $(C, M, S) \not\uparrow$.*

Proof. We prove each claim separately:

1. We proceed by induction on $(C, M, S) \downarrow$:
   - Case of $M \equiv V$ and $S = \epsilon$. Since (C, V, ϵ) is irreducible, but the stack is empty, there is no way for (C, V, ϵ) to go into deadlock, so we conclude $(C, V, \epsilon) \not\perp$.
   - Case of $(C, M, S) \Rightarrow (D, N, S')$ and $(D, N, S') \downarrow$. Since (C, M, S) is reducible, it must be that $(C, M, S) \Rightarrow (D, N, S')$ ($\Rightarrow$ is deterministic) and $(D, N, S')\bot$ in order for (C, M, S) to go into deadlock. However, by inductive hypothesis we know that $(D, N, S') \not\perp$, so this is impossible and we conclude $(C, M, S) \not\perp$.
2. We proceed by induction on $(C, M, S) \downarrow$:
   - Case of $M \equiv V$ and $S = \epsilon$. Since (D, V, ϵ) is irreducible, there is no way for it to diverge, so we conclude $(D, V, \epsilon) \not\uparrow$.
   - Case of $(C, M, S) \Rightarrow (D, N, S')$ and $(D, N, S') \downarrow$. Since $(D, N, S') \not\uparrow$ by inductive hypothesis and (C, M, S) cannot reduce to any other configuration ($\Rightarrow$ is deterministic), there is no way for (C, M, S) to diverge, so we conclude $(C, M, S) \not\uparrow$.
3. We proceed by induction on $(C, M, S)\bot$:
   - Case in which (C, M, S) irreducible and $S \neq \epsilon$. Since (C, M, S) is irreducible, there is no way for it to diverge, so we conclude $(C, M, S) \not\uparrow$.

- Case of $(C, M, S) \Rightarrow (D, N, S')$ and $(D, N, S')\bot$. Since $(D, N, S') \not\downarrow$ by inductive hypothesis and (C, M, S) cannot reduce to any other configuration ($\Rightarrow$ is deterministic), there is no way for (C, M, S) to diverge, so we conclude $(C, M, S) \not\uparrow$.

□

We also prove that convergence, deadlock and divergence are total on machine configurations, like we did in the small-step case. In this case, however, we need the following lemma.

Lemma 3.2.4. *If (C, M, S) is irreducible, then M is a value.*

Proof. We prove the contrapositive. That is, if M is not a value then (C, M, S) is reducible. We proceed by cases on the form of M:

- Case $M \equiv x$. This case is impossible, since by the definition of machine configuration M must contain no free variables.
- Case $M \equiv \vec{\ell}$. In this case M is a value and the claim is vacuously true.
- Case $M \equiv \lambda x.N$. In this case M is a value and the claim is vacuously true.
- Case $M \equiv NP$. In this case (C, NP, S) can be reduced by the *app-split* rule.
- Case $M \equiv \langle N, P\rangle$. If N and P are both values then $\langle N, P\rangle$ is a value too and the claim is vacuously true. Otherwise, $(C, \langle N, P\rangle, S)$ can be reduced by the *tuple-split* rule.
- Case $M \equiv \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P$. In this case $(C, \mathsf{let}\,\langle x, y\rangle = N\,\mathsf{in}\,P, S)$ can be reduced by the *let-split* rule.
- Case $M \equiv \mathsf{lift}\,N$. In this case M is a value and the claim is vacuously true.
- Case $M \equiv \mathsf{force}\,N$. In this case $(C, \mathsf{force}\,N, S)$ can be reduced by the *force-open* rule.
- Case $M \equiv \mathsf{box}_T\,N$. In this case $(C, \mathsf{box}_T\,N, S)$ can be reduced by the *box-open* rule.
- Case $M \equiv \mathsf{apply}(N, P)$. In this case $(C, \mathsf{apply}(N, P), S)$ can be reduced by the *apply-split* rule.
- Case $M \equiv (\vec{\ell}, D, \vec{\ell'})$. In this case M is a value and the claim is vacuously true.

□

Proposition 3.2.5. *Every machine configuration (C, M, S) either converges, goes into deadlock or diverges, that is:*

$$(C, M, S)\downarrow \vee (C, M, S)\bot \vee (C, M, S)\uparrow .$$

Proof. Let clen be a function that, given a machine configuration, returns the number of reduction steps that can be taken starting from that configuration. The clen function is defined as the least fixed point of the following equation on functions from machine configurations to $\mathbb{N}^\infty$:

$$\mathsf{clen}(C, M, S) = \begin{cases} \mathsf{clen}(D, N, R) + 1 & \text{if } (C, M, S) \Rightarrow (D, N, R), \\ 0 & \text{otherwise.} \end{cases}$$

If $\mathsf{clen}(C, M, S) = \infty$, that means that $(C, M, S) \Rightarrow (D, N, R)$ and $\mathsf{clen}(D, N, R) = \infty$. Because $\uparrow$ is defined as the largest relation such that $(C, M, S) \uparrow$ implies $(C, M, S) \Rightarrow (D, N, R)$ and $(D, N, R) \uparrow$, we conclude that $(C, M, S) \uparrow$. On the other hand, if $\mathsf{clen}(C, M, S) \in \mathbb{N}$, we proceed by induction on $\mathsf{clen}(C, M, S)$:

- Case $\mathsf{clen}(C, M, S) = 0$. In this case (C, M, S) is irreducible and by Lemma 3.2.4 we know $M \equiv V$. If $S = \epsilon$, then we trivially conclude $(C, V, \epsilon) \downarrow$. Otherwise, if $S \neq \epsilon$, we trivially conclude $(C, V, S)\bot$.

- Case $\mathsf{clen}(C, M, S) = n + 1$. In this case we know that $(C, M, S) \Rightarrow (D, N, R)$ and $\mathsf{clen}(D, N, R) = n$. By inductive hypothesis, we know that either $(D, N, R) \downarrow$ or $(D, N, R)\bot$ or $(D, N, R) \uparrow$. We exclude the last option, because if $(D, N, R) \uparrow$, then $\mathsf{clen}(D, N, R)$ (and as a consequence $\mathsf{clen}(C, M, S)$) would be undefined, contradicting the hypothesis. If $(D, N, R) \downarrow$, we have $(C, M, S) \Rightarrow (D, N, R)$ and we conclude $(C, M, S) \downarrow$, whereas if $(D, N, R)\bot$ we conclude $(C, M, S)\bot$ by the same reasoning.

□

3.2.5 Summary

In this section we introduced a machine semantics for Proto-Quipper-M inspired by the CEK abstract machine. This semantics takes the stack approach of the stacked semantics even further, effectively pushing a new frame onto the stack every time a sub-term has to be evaluated. As a result, this semantics is devoid of contextual rules. Similarly to what we did with the stacked semantics in Section 3.1, we established the subset of "benign" machine configurations in which we are actually interested, namely reachable configurations. Lastly, we gave definitions for the convergence, deadlock and divergence of machine configurations, which we proved to be mutually exclusive and total (when considered in disjunction) relations.

Chapter 4

Correspondence Results

We are at a point where we have presented three different semantics for the evaluation of Proto-Quipper-M programs and we have analyzed their individual properties. In this chapter, we start looking at the various semantics in relationship with each other, in order to eventually prove that the machine semantics that we arrived to in Chapter 3 is ultimately equivalent to the small-step semantics given in Chapter 2 and, as a consequence, to the original big-step operational semantics given by Rios and Selinger. To this effect, we first focus on the relationship between the stacked semantics and the other two semantics. Then, in light of the respective results, we prove that the computations carried out in the small-step and machine semantics are equivalent by proving that they are simulated by the same computation in the stacked semantics.

4.1 From Small-step to Stacked Semantics

Since we plan to use the stacked semantics as a "middle ground" on which the small-step and machine semantics ought to agree, when in this section we prove results about the relationship between the small-step and stacked semantics we mainly focus on the direction that goes from the former to the latter.

4.1.1 Configurations

In Section 3.1.1 we alluded to some sort of relationship between small-step configurations of the form (C, M) and stacked configurations of the form $(C, M)^{\emptyset}.\epsilon$. To formalize this relationship, we define the following function:

$$\mathsf{fromSmallStep}(C, M) = (C, M)^{\emptyset}.\epsilon.$$

Note that although we focus on one direction of this relationship, the set of small-step configurations and the set $\mathcal{I}_{st}$ of initial stacked configurations are effectively in bijection,

since fromSmallStep is trivially invertible.

4.1.2 Reductions

The most relevant result that we prove in this section is that a single reduction step in the small-step semantics can always be simulated by a reduction sequence in the stacked semantics.

Lemma 4.1.1. *Let $\rightharpoonup^+$ be the transitive closure of $\rightharpoonup$. If $(C, M) \to (C', M')$, then for every $\vec{k}$ and X we have*

$$(C, M)^{\vec{k}}.X \rightharpoonup^+ (C', M')^{\vec{k}}.X.$$

Furthermore, if $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ then we also have $C' \equiv C, M' \equiv E[(\vec{\ell}, D, \vec{\ell'})]$ and

$$(id_Q, N\vec{\ell})^{\vec{\ell}}.X \rightharpoonup^+ (D, \vec{\ell'})^{\vec{\ell}}.X,$$

for all X and for $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$.

Proof. By induction on the derivation of $(C, M) \to (C', M')$. We distinguish two cases. If $M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$, then we immediately conclude$(C, M)^{\vec{k}}.X \rightharpoonup (C', M')^{\vec{k}}.X$, for all $\vec{k}$ and X, by the *head* rule. Otherwise, if $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$, we proceed by cases on E:

- Case $E \equiv [\cdot]$. In this case we have

$$\frac{(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T) \quad (id_Q, N\vec{\ell}) \to \ldots \to (D, \vec{\ell'})}{(C, \mathsf{box}_T(\mathsf{lift}\, N)) \to (C, (\vec{\ell}, D, \vec{\ell}))}\, box$$

  Let $(C_1, M_1) \to (C_2, M_2), (C_2, M_2) \to (C_3, M_3), \ldots, (C_{n-1}, M_{n-1}) \to (C_n, M_n)$ be the sequence of reduction steps denoted by $(id_Q, N\vec{\ell}) \to \ldots \to (D, \vec{\ell'})$, such that $(id_Q, N\vec{\ell}) \equiv (C_1, M_1)$ and $(C_n, M_n) \equiv (D, \vec{\ell'})$. By applying the inductive hypothesis to each of these reduction steps, we get $(id_Q, N\vec{\ell})^{\vec{\ell}}.X \rightharpoonup^+ (C_2, M_2)^{\vec{\ell}}.X, (C_2, M_2)^{\vec{\ell}}.X \rightharpoonup^+ (C_3, M_3)^{\vec{\ell}}.X$, up until $(C_{n-1}, M_{n-1})^{\vec{\ell}}.X \rightharpoonup^+ (D, \vec{\ell'})^{\vec{\ell}}.X$, for all X, and therefore $(id_Q, N\vec{\ell})^{\vec{\ell}}.X \rightharpoonup^+ (D, \vec{\ell'})^{\vec{\ell}}.X$, for all X, by the transitivity of $\rightharpoonup^+$. We therefore consider the following derivation:

$$\frac{(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)}{(C, \mathsf{box}_T(\mathsf{lift}\, N))^{\vec{k}}.X' \rightharpoonup (id_Q, N\vec{\ell})^{\vec{\ell}}.(C, \mathsf{box}_T(\mathsf{lift}\, N))^{\vec{k}}.X'}\, step\text{-}in$$

  and by appropriately setting $X = (C, \mathsf{box}_T(\mathsf{lift}\, N))^{\vec{k}}.X'$ we get $(C, \mathsf{box}_T(\mathsf{lift}\, N))^{\vec{k}}.X' \rightharpoonup^+ (D, \vec{\ell'})^{\vec{\ell}}.(C, \mathsf{box}_T(\mathsf{lift}\, N))^{\vec{k}}.X'$, for all X', thanks to the transitivity of $\rightharpoonup^+$. Next, we consider the following derivation:

$$\frac{}{(D, \vec{\ell'})^{\vec{\ell}}.(C, \mathsf{box}_T(\mathsf{lift}\, N))^{\vec{k}}.X' \rightharpoonup (C, (\vec{\ell}, D, \vec{\ell'}))^{\vec{k}}.X'}\, step\text{-}out$$

by which we conclude $(C, \mathsf{box}_T(\mathsf{lift}\, N))^{\vec{k}}.X' \rightharpoonup^+ (C, (\vec{\ell}, D, \vec{\ell'}))^{\vec{k}}.X'$, for all $\vec{k}$ and X', thanks to the transitivity of $\rightharpoonup^+$.

- Case $E \equiv FP$. In this case we have

$$\frac{(C, F[\mathsf{box}_T(\mathsf{lift}\, N)]) \to (C', M')}{(C, F[\mathsf{box}_T(\mathsf{lift}\, N)]P) \to (C', M'P)}\, \textit{ctx-app-left}$$

  By inductive hypothesis we know that $C \equiv C'$, that $M' \equiv F[(\vec{\ell}, D, \vec{\ell'})]P$ and that $(id_Q, N\vec{\ell})^{\vec{\ell}}.X \rightharpoonup^+ (D, \vec{\ell'})^{\vec{\ell}}.X$, for all X and for $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. By the same reasoning used in the $E \equiv [\cdot]$ case we can derive $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X' \rightharpoonup (id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X'$ by the *step-in* rule. Similarly, we can derive $(D, \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X' \rightharpoonup (C, E[(\vec{\ell}, D, \vec{\ell'})])^{\vec{k}}.X'$ by the *step-out* rule. Therefore, by setting $X = (C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X'$ we conclude $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X' \rightharpoonup^+ (C, E[(\vec{\ell}, D, \vec{\ell'})])^{\vec{k}}.X'$, for all $\vec{k}$ and X', thanks to the transitivity of $\rightharpoonup^+$.

- Case $E \equiv VF$. This case is proven in the same way as the $E \equiv FP$ case, except with *ctx-app-right* instead of *ctx-app-left*.

- Case $E \equiv \langle F, P\rangle$. This case is proven in the same way as the $E \equiv FP$ case, except with *ctx-tuple-left* instead of *ctx-app-left*.

- Case $E \equiv \langle V, F\rangle$. This case is proven in the same way as the $E \equiv FP$ case, except with *ctx-tuple-right* instead of *ctx-app-left*.

- Case $E \equiv \mathsf{let}\, \langle x, y\rangle = F\, \mathsf{in}\, P$. This case is proven in the same way as the $E \equiv FP$ case, except with *ctx-let* instead of *ctx-app-left*.

- Case $E \equiv \mathsf{force}\, F$. This case is proven in the same way as the $E \equiv FP$ case, except with *ctx-force* instead of *ctx-app-left*.

- Case $E \equiv \mathsf{box}_U\, F$. This case is proven in the same way as the $E \equiv FP$ case, except with *ctx-box* instead of *ctx-app-left*.

- Case $E \equiv \mathsf{apply}(F, P)$. This case is proven in the same way as the $E \equiv FP$ case, except with *ctx-apply-left* instead of *ctx-app-left*.

- Case $E \equiv \mathsf{apply}(V, F)$. This case is proven in the same way as the $E \equiv FP$ case, except with *ctx-apply-right* instead of *ctx-app-left*.

□

Corollary 4.1.1.1. *Suppose* (C, M) *and* (C', M') *are two small-step configurations. If* $(C, M) \to (C', M')$*, then* $\mathsf{fromSmallStep}(C, M) \rightharpoonup^+ \mathsf{fromSmallStep}(C', M')$.

Proof. The claim follows immediately from Lemma 4.1.1 by setting $\vec{k} = \emptyset$ and $X = \epsilon$. □

A weaker result holds in the other direction. Specifically, a reduction sequence in the stacked semantics can be simulated by a reduction sequence in the small-step semantics only if the former begins and ends on configurations of the same length and all of the intermediate configurations have length greater or equal to that of the endpoints. In other words, the small-step semantics can only simulate computations that begin and end in the same stack frame. This is to be expected, since in the small-step semantics the stack of sub-reductions introduced by the boxing operator exists only in the derivation tree of the reduction sequence, and not in the reduction sequence itself, so a computation $(C, M) \to^* (D, N)$ where (C, M) and (D, N) belong to different sub-reductions is effectively meaningless.

Lemma 4.1.2. *Let $(C, M)^{\vec{k}}.X$ and $(D, N)^{\vec{k}}.X$ be two stacked configurations of length m. If $(C, M)^{\vec{k}}.X \rightharpoonup^+ (D, N)^{\vec{k}}.X$ and all the intermediate configurations in this reduction have length m or greater, then $(C, M) \to^+ (D, N)$.*

Proof. By induction on the length of the reduction $(C, M)^{\vec{k}}.X \rightharpoonup^+ (D, N)^{\vec{k}}.X$:

- Case of 1. In this case we have $(C, M)^{\vec{k}}.X \rightharpoonup (D, N)^{\vec{k}}.X$. The only rule which is consistent with the hypothesis is *head*, so we know that $(C, M) \to (D, N)$ and the claim is trivially true.
- Case of $n + 1$. In this case we have $(C, M)^{\vec{k}}.X \rightharpoonup (C', M')^{\vec{\ell}}.X' \rightharpoonup^+ (D, N)^{\vec{k}}.X$ and we proceed by cases on the introduction of $(C, M)^{\vec{k}}.X \rightharpoonup (C', M')^{\vec{\ell}}.X'$:
  - Case of *head*. In this case we have $C' \equiv C, \vec{\ell} = \vec{k}$ and $X' = X$ and we know that $(C, M) \to (C', M')$.By inductive hypothesis we also know that $(C', M') \to^+ (D, N)$, so by the transitivity of $\to^+$ we conclude $(C, M) \to^+ (D, N)$.
  - Case of *step-in*. In this case we know that $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and we have $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X \rightharpoonup (id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. We know that $(id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$ is a configuration of length $m+1$, so in order for it to eventually reduce to $(D, N)^{\vec{k}}.X$ there must be a shortening reduction somewhere between $(id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$ and $(D, N)^{\vec{k}}.X$. Because the only shortening reductions are introduced by the *step-out* rule we must have

$$\begin{aligned}(id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X &\rightharpoonup^+ (D', \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X \\ &\rightharpoonup (C, E[(\vec{\ell}, D', \vec{\ell'})])^{\vec{k}}.X \\ &\rightharpoonup^* (D, N)^{\vec{k}}.X,\end{aligned}$$

where every intermediate configuration in $(id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X \multimap^+ (D', \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$ has length greater or equal than $m+1$. By applying the inductive hypothesis on this reduction, which has length less than n, we get $(id_Q, N\vec{\ell}) \to^+ (D', \vec{\ell'})$ and therefore $(C, \mathsf{box}_T(\mathsf{lift}\, N)) \to (C, (\vec{\ell}, D', \vec{\ell'}))$ by the *box* rule. By applying Theorem 2.5.3 we also get $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)]) \to (C, E[(\vec{\ell}, D', \vec{\ell'})])$. At this point, if $(C, E[(\vec{\ell}, D', \vec{\ell'})])^{\vec{k}}.X = (D, N)^{\vec{k}}.X$ then trivially $(C, E[(\vec{\ell}, D', \vec{\ell'})]) = (D, N)$ and the claim is proven. Otherwise, we have a reduction sequence $(C, E[(\vec{\ell}, D', \vec{\ell'})])^{\vec{k}}.X \multimap^+ (D, N)^{\vec{k}}.X$, of length less than n, so by inductive hypothesis we get $(C, E[(\vec{\ell}, D', \vec{\ell'})]) \to^+ (D, N)$ and conclude $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)]) \multimap^+ (D, N)$.

– Case of *step-out*. This case is impossible since it immediately violates the hypothesis that every intermediate configuration is of length m or greater.

□

Corollary 4.1.2.1. *Suppose* (C, M) *and* (D, N) *are two small-step configurations. If* $\mathsf{fromSmallStep}(C, M) \multimap^+ \mathsf{fromSmallStep}(D, N)$*, then* $(C, M) \to^+ (D, N)$*.*

Proof. The claim follows immediately from Lemma 4.1.2 by setting $\vec{k} = \emptyset$ and $X = \epsilon$. □

These two results will play a relevant role in the following sections, and they can be summarized graphically in the following diagram:

$$\begin{array}{ccc} (C, M) & \xrightarrow{\quad\quad\quad\quad\quad\quad\quad\quad} & (D, N) \\ \updownarrow & & \updownarrow \\ (C, M)^{\vec{\ell}}.X & \xrightarrow{\quad\quad\quad\quad\quad\quad\quad\quad} & (D, N)^{\vec{\ell}}.X \end{array}$$

4.1.3 Convergence, Deadlock and Divergence

For the sake of the equivalence between the small-step and machine semantics, it is not essential that the $\mathsf{fromSmallStep}$ function preserve all three of the convergence, deadlock and divergence relations between the small-step and the stacked semantics. In particular, we will see that it is sufficient to prove that $\mathsf{fromSmallStep}$ preserves convergence and that whenever (C, M) goes into deadlock, then $\mathsf{fromSmallStep}(C, M)$ also goes into deadlock. If we consider that to converge essentially means to evaluate to a term of a certain form, we can see that the first result is a trivial consequence of the two previous lemmata, as can be seen by this specific instance of the diagram that we just presented:

$$\begin{array}{ccc} (C, M) & \xrightarrow{\quad\quad\quad\quad\quad\quad\quad\quad} & (D, N) \\ \shortparallel & & \shortparallel \\ \mathsf{fromSmallStep} & & \mathsf{fromSmallStep} \\ \shortparallel & & \shortparallel \\ (C, M)^{\emptyset}.\epsilon & \xrightarrow{\quad\quad\quad\quad\quad\quad\quad\quad} & (D, N)^{\emptyset}.\epsilon \end{array}$$

More formally, we give the following result.

Proposition 4.1.3. $(C, M)\downarrow$ *if and only if* $\mathsf{fromSmallStep}(C, M)\downarrow$.

Proof. It is easy to see that a small-step configuration (C, M) converges if and only if $(C, M) \to^* (D, V)$ for some D, V. It is equally easy to see that a stacked configuration $(C, M)^{\emptyset}.\epsilon$ converges if and only if $(C, M)^{\emptyset}.\epsilon \rightharpoonup^* (D, V)^{\emptyset}.\epsilon$ for some D, V. As a result, the claim follows trivially from corollaries 4.1.1.1 and 4.1.2.1.

□

A similar intuition applies to the deadlocking case. However, because small-step configurations can go into deadlock because of a sub-reduction introduced by *box*, the proof of the second result is less trivial.

Lemma 4.1.4. *Suppose* (C, M) *is a small-step configuration. If* $(C, M)\bot$*, then* $\mathsf{fromSmallStep}(C, M)\bot$.

Proof. We proceed by induction on $(C, M)\bot$:

- Case in which (C, M) is irreducible, $M \not\equiv V$ and $M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$. Consider $\mathsf{fromSmallStep}(C, M) = (C, M)^{\emptyset}.\epsilon$. This configuration cannot be reduced by the *head* rule, since this would imply the reducibility of (C, M), nor by the *step-in* rule, since $M \not\equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$, nor by *step-out*, since $M \not\equiv V$ and the rest of the stack is empty. Therefore $(C, M)^{\emptyset}.\epsilon$ is irreducible and because $M \not\equiv V$ we conclude $(C, M)^{\emptyset}.\epsilon\bot$.

- Case of $(C, M) \to (D, N)$ and $(D, N)\bot$. By Lemma 4.1.1 we know that $\mathsf{fromSmallStep}(C, M) \rightharpoonup^+ \mathsf{fromSmallStep}(D, N)$. By inductive hypothesis we know that $\mathsf{fromSmallStep}(D, N)\bot$ and conclude that $\mathsf{fromSmallStep}(C, M)\bot$.

- Case of $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $(id_Q, N\vec{\ell})\bot$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. In this case we have $\mathsf{fromMachine}(C, E[\mathsf{box}_T(\mathsf{lift}\, N)]) = (C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon \rightharpoonup (id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon$ by the *step-in* rule and inductive hypothesis we know that $\mathsf{fromMachine}(id_Q, N\vec{\ell}) = (id_Q, N\vec{\ell})^{\emptyset}.\epsilon\bot$. From this and Lemma 3.1.5 we get that $(id_Q, N\vec{\ell})^{\vec{k}}.X\bot$ for all $\vec{k}$ and X, including $\vec{k} = \vec{\ell}$ and $X = (C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon$, so we know $(id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon\bot$ and conclude $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon\bot$.

- Case of $M \equiv E[\mathsf{box}_T(\mathsf{lift}\, N)]$ and $(id_Q, N\vec{\ell}) \to^* (D, V)$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$ and V is not a label tuple. In this case we have $\mathsf{fromMachine}(C, E[\mathsf{box}_T(\mathsf{lift}\, N)]) = (C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon \rightharpoonup (id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon$ by the *step-in* rule and we know $(id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon \rightharpoonup^* (D, V)^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon$ by Lemma 4.1.1. Since $(D, V)^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon$ cannot be reduced by the *head* rule (since V is a value), nor by the *step-in* rule (since V cannot be of the form $F[\mathsf{box}_T(\mathsf{lift}\, P)]$ for any F, P), nor by the *step-out* rule (since V is not a label tuple), and the rest of

the stack is not empty, we get that $(D, V)^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon\bot$ and conclude that $(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\emptyset}.\epsilon\bot$.

□

4.1.4 Summary

In this section we established a relationship between small-step configurations and stacked configurations via the fromSmallStep function. We showed that a reduction sequence between small-step configurations can be simulated by a reduction sequence between the corresponding stacked configurations, and vice-versa (to an extent). As a result, we proved that fromMachine preserves convergence between small-step and stacked configurations, and that whenever a small-step configuration goes into deadlock, then so does the corresponding stacked configuration.

4.2 From Machine to Stacked Semantics

We now explore the "other side" of the equivalence between the small-step and machine semantics, that is, the relationship between the machine semantics and the stacked semantics, with emphasis on the direction that goes from the former to the latter.

4.2.1 Configurations

Whereas the relationship between small-step and stacked configurations is trivial and mainly concerns initial configurations, the relationship between machine and stacked configurations is at the same time more pervasive (it holds for all configurations, not just initial ones) and slightly more complicated to define. For this purpose, we formalize this relationship via a fromMachine function which maps machine configurations into stacked configurations:

$$
\begin{aligned}
\mathsf{fromMachine}(C, M, \epsilon) &= (C, M)^{\emptyset}.\epsilon \\
\mathsf{fromMachine}(C, M, \mathit{FArg}(N).S) &= \mathsf{fromMachine}(C, MN, S) \\
\mathsf{fromMachine}(C, M, \mathit{FApp}(V).S) &= \mathsf{fromMachine}(C, VM, S) \\
\mathsf{fromMachine}(C, M, \mathit{ALabel}(N).S) &= \mathsf{fromMachine}(C, \mathsf{apply}(M, N), S) \\
\mathsf{fromMachine}(C, M, \mathit{ACirc}(V).S) &= \mathsf{fromMachine}(C, \mathsf{apply}(V, M), S) \\
\mathsf{fromMachine}(C, M, \mathit{TRight}(N).S) &= \mathsf{fromMachine}(C, \langle M, N\rangle, S) \\
\mathsf{fromMachine}(C, M, \mathit{TLeft}(V).S) &= \mathsf{fromMachine}(C, \langle V, M\rangle, S) \\
\mathsf{fromMachine}(C, M, \mathit{Box}(Q, \vec{\ell}).S) &= \mathsf{fromMachine}(C, \mathsf{box}_T\, M, S) \text{ where } \emptyset; Q \vdash \vec{\ell} : T \\
\mathsf{fromMachine}(C, M, \mathit{Sub}(D, N, \vec{\ell}, T).S) &= (C, M)^{\vec{\ell}}.\, \mathsf{fromMachine}(D, \mathsf{box}_T(\mathsf{lift}\, N), S) \\
\mathsf{fromMachine}(C, M, \mathit{Let}(x, y, N).S) &= \mathsf{fromMachine}(C, \mathsf{let}\, \langle x, y\rangle = M\, \mathsf{in}\, N, S) \\
\mathsf{fromMachine}(C, M, \mathit{Force}.S) &= \mathsf{fromMachine}(C, \mathsf{force}\, M, S).
\end{aligned}
$$

Before we discuss this definition in greater detail, it is worth noting that by restricting the domain of fromMachine to the set $\mathcal{I}_{ma}$ of initial machine configurations, we trivially have a bijection between $\mathcal{I}_{ma}$ and the set $\mathcal{I}_{st}$ of initial stacked configurations, which is analogous to the one we had in Section 4.1.1 between small-step configurations and initial stacked configurations.

Informally, the fromMachine function takes the term that the machine configuration is currently focused on and gradually unwinds the stack to rebuild the term being evaluated in its entirety. The only case in which we actually change the current stack frame in the resulting stacked configuration is, unsurprisingly, when we encounter a continuation of type *Sub*. As a result, the machine stack is encoded in the resulting stacked configuration partly as the configuration stack itself, and partly as the structure of the terms contained within each individual stack frame. This structure is not arbitrary. In fact, we have that it always corresponds to the structure of an evaluation context, as we prove in the following result.

Proposition 4.2.1. *If* $\mathsf{fromMachine}(C, M, S) = (D, N)^{\vec{\ell}}.X$*, then* $C \equiv D$ *and* N *is of the form* $E[M]$ *for some evaluation context* E*.*

Proof. By induction on $|S|$, the size of S, defined in the natural manner. If $|S| = 0$, it means that $S = \epsilon$. In this case we have $\mathsf{fromMachine}(C, M, \epsilon) = (C, M)^{\emptyset}.\epsilon$ and the claim is true for $E \equiv [\cdot]$. If $|S| = n + 1$, suppose $S \equiv H.S'$, where $|S'| = n$. We proceed by cases on H:

- Case $H \equiv \mathit{FArg}(N)$. In this case we have $\mathsf{fromMachine}(C, M, \mathit{FArg}(N).S') = \mathsf{fromMachine}(C, MN, S')$. We know that $MN \equiv E[M]$ for $E \equiv [\cdot]N$. Furthermore,

by inductive hypothesis we get that $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{\ell}}.X$ for some E' and by Proposition 2.5.7 we conclude $\mathsf{fromMachine}(C, M, \mathit{FArg}(N).S') = (C, E''[M])^{\vec{\ell}}.X$ for some E''.

- Case $H \equiv \mathit{FApp}(V)$. In this case we have $\mathsf{fromMachine}(C, M, \mathit{FApp}(V).S') = \mathsf{fromMachine}(C, VM, S')$. We know that $VM \equiv E[M]$ for $E \equiv V[\cdot]$. Furthermore, by inductive hypothesis we get that $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{\ell}}.X$ for some E' and by Proposition 2.5.7 we eventually conclude that $\mathsf{fromMachine}(C, M, \mathit{FApp}(V).S') = (C, E''[M])^{\vec{\ell}}.X$ for some E''.

- Case $H \equiv \mathit{ALabel}(N)$. In this case we have $\mathsf{fromMachine}(C, M, \mathit{ALabel}(N).S') = \mathsf{fromMachine}(C, \mathsf{apply}(M, N), S')$. We know that $\mathsf{apply}(M, N) \equiv E[M]$ for $E \equiv \mathsf{apply}([\cdot], N)$. Furthermore, by inductive hypothesis we get $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{\ell}}.X$ for some E' and by Proposition 2.5.7 we eventually conclude that $\mathsf{fromMachine}(C, M, \mathit{ALabel}(N).S') = (C, E''[M])^{\vec{\ell}}.X$ for some E''.

- Case $H \equiv \mathit{ACirc}(V)$. In this case we have $\mathsf{fromMachine}(C, M, \mathit{ACirc}(V).S') = \mathsf{fromMachine}(C, \mathsf{apply}(V, M), S')$. We know that $\mathsf{apply}(V, M) \equiv E[M]$ for $E \equiv \mathsf{apply}(V, [\cdot])$. Furthermore, by inductive hypothesis we get $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{k}}.X$ for some E' and by Proposition 2.5.7 we can conclude that $\mathsf{fromMachine}(C, M, \mathit{ACirc}(V).S') = (C, E''[M])^{\vec{k}}.X$ for some E''.

- Case $H \equiv \mathit{TRight}(N)$. In this case we have $\mathsf{fromMachine}(C, M, \mathit{TRight}(N).S') = \mathsf{fromMachine}(C, \langle M, N\rangle, S')$. We know that $\langle M, N\rangle \equiv E[M]$ for $E \equiv \langle [\cdot], N\rangle$. Furthermore, by inductive hypothesis we get that $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{\ell}}.X$ for some E' and by Proposition 2.5.7 we eventually conclude that $\mathsf{fromMachine}(C, M, \mathit{TRight}(N).S') = (C, E''[M])^{\vec{\ell}}.X$ for some E''.

- Case $H \equiv \mathit{TLeft}(V)$. In this case we have $\mathsf{fromMachine}(C, M, \mathit{TLeft}(V).S') = \mathsf{fromMachine}(C, \langle V, M\rangle, S')$. We know that $\langle V, M\rangle \equiv E[M]$ for $E \equiv \langle V, [\cdot]\rangle$. Furthermore, by inductive hypothesis we get that $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{\ell}}.X$ for some E' and by Proposition 2.5.7 we eventually conclude that $\mathsf{fromMachine}(C, M, \mathit{TLeft}(V).S') = (C, E''[M])^{\vec{\ell}}.X$ for some E''.

- Case $H \equiv \mathit{Box}(Q, \vec{\ell})$. In this case we have $\mathsf{fromMachine}(C, M, \mathit{Box}(Q, \vec{\ell}).S') = \mathsf{fromMachine}(C, \mathsf{box}_T\, M, S')$, where $\emptyset; Q \vdash \vec{\ell} : T$. We know that $\mathsf{box}_T\, M \equiv E[M]$ for $E \equiv \mathsf{box}_T\, [\cdot]$. Furthermore, by inductive hypothesis we get $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{\ell}}.X$ for some E' and by Proposition 2.5.7 we eventually conclude that $\mathsf{fromMachine}(C, M, \mathit{Box}(Q, \vec{\ell'}).S') = (C, E''[M])^{\vec{k}}.X$ for some E''.

- Case $H \equiv \mathit{Sub}(D, N, \vec{\ell}, T)$. In this case $\mathsf{fromMachine}(C, M, \mathit{Sub}(D, N, \vec{\ell}, T).S') = (C, M)^{\vec{\ell}}.\, \mathsf{fromMachine}(D, \mathsf{box}_T(\mathsf{lift}\, N), S')$ and the claim is trivially true for $E \equiv [\cdot]$.

- Case $H \equiv Let(x, y, N)$. In this case we have $\mathsf{fromMachine}(C, M, Let(x, y, N).S') = \mathsf{fromMachine}(C, \mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N, S')$. We know that $\mathsf{let}\,\langle x, y\rangle = M\,\mathsf{in}\,N \equiv E[M]$ for $E \equiv \mathsf{let}\,\langle x, y\rangle = [\cdot]\,\mathsf{in}\,N$. Furthermore, by inductive hypothesis we get that $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{\ell}}.X$ for some E' and by Proposition 2.5.7 we conclude $\mathsf{fromMachine}(C, M, Let(x, y, N).S') = (C, E''[M])^{\vec{\ell}}.X$ for some E''.

- Case $H \equiv Force$. In this case we know that $\mathsf{fromMachine}(C, M, Force.S') = \mathsf{fromMachine}(C, \mathsf{force}\,M, S')$. We also know that $\mathsf{force}\,M \equiv E[M]$ for $E \equiv \mathsf{force}\,[\cdot]$. Furthermore, by inductive hypothesis we get that $\mathsf{fromMachine}(C, E[M], S') = (C, E'[E[M]])^{\vec{\ell}}.X$ for some E' and by Proposition 2.5.7 we eventually conclude that $\mathsf{fromMachine}(C, M, Force.S') = (C, E''[M])^{\vec{k}}.X$ for some E''.

□

The following result goes even further, as it proves that the evaluation context E that we introduced in the previous lemma, as well as the locally available labels $\vec{\ell}$ and the rest of the stack X all depend *exclusively* on the machine stack S.

Proposition 4.2.2. *If two machine configurations (C, M, S) and (D, N, S) share the same stack S, then $\mathsf{fromMachine}(C, M, S) = (C, E[M])^{\vec{\ell}}.X$ and $\mathsf{fromMachine}(D, N, S) = (D, E[N])^{\vec{\ell}}.X$ for the same E, $\vec{\ell}$ and X.*

Proof. The existence of E is guaranteed by Proposition 4.2.1. The identity of $E, \vec{\ell}$ and X can be proven trivially by induction on the length of S. □

4.2.2 Reductions

Because the machine semantics is, intuitively, more fine-grained than the stacked semantics, it comes as no surprise that distinct machine configurations are mapped by $\mathsf{fromMachine}$ to the same stacked configuration. As an example, take the configurations $(C, V, FArg(W).\epsilon)$ and $(C, W, FApp(V).\epsilon)$, for any values V, W. We have

$$\mathsf{fromMachine}(C, V, FArg(W).\epsilon) = \mathsf{fromMachine}(C, VW, \epsilon) = (C, VW)^{\emptyset}.\epsilon,$$
$$\mathsf{fromMachine}(C, W, FApp(V).\epsilon) = \mathsf{fromMachine}(C, VW, \epsilon) = (C, VW)^{\emptyset}.\epsilon.$$

Intuitively, this is due to the fact that the two configurations represents two different phases in the evaluation of the same application VW, which in contrast can be evaluated in a single step in the stacked semantics. In fact, most of the rules of the machine semantics only serve to decompose or move around terms (e.g, the *split* or *shift* rules) and because they do not actually evaluate anything, they have no appreciable effect on the corresponding stacked configuration. This will be a crucial aspect to consider in the coming results, so we ought to formalize it. Let $\Rightarrow_b$ be the proper subset of $\Rightarrow$ that

can be derived by only using rules for $\Rightarrow$ whose conclusion $(C, M, S) \Rightarrow (D, N, R)$ is such that $\mathsf{fromMachine}(C, M, S) = \mathsf{fromMachine}(D, N, R)$. These rules are specifically *app-split, app-shift, apply-split, apply-shift, let-split, tuple-split, tuple-shift, tuple-join, box-open, force-open.* Also, let $\Rightarrow_r$ be the subset of $\Rightarrow$ that can be derived by only using the remaining rules, that is, *app-join, apply-join, box-sub, box-close, let-join, force-close*, such that

$$(C, M, S) \Rightarrow (D, N, R) \iff (C, M, S) \Rightarrow_b (D, N, R) \vee (C, M, S) \Rightarrow_r (D, N, R).$$

The most essential result that we must prove is that there is a limit to the number of times we can reduce a machine configuration (C, M, S) without causing any change in the corresponding stacked configuration $\mathsf{fromMachine}(C, M, S)$. In other words, we must prove that $\Rightarrow_b$ is strongly normalizing.

Lemma 4.2.3. *The reduction relation $\Rightarrow_b$ is strongly normalizing.*

Proof. Let $\#_t$ be a function on terms defined as such:

$$\begin{aligned} \#_t(V) &= 0, \\ \#_t(\mathsf{box}_T\, M) = \#_t(\mathsf{force}\, M) = \#_t(\mathsf{let}\, \langle x, y\rangle = M\, \mathsf{in}\, N) &= \#_t(M) + 1, \\ \#_t(MN) = \#_t(\mathsf{apply}(M, N)) &= \#_t(M) + \#_t(N) + 2, \\ \#_t(\langle M, N\rangle) &= \#_t(M) + \#_t(N) + 3. \end{aligned}$$

Now let $\#_s$ be a function on machine stacks defined as such:

$$\begin{aligned} \#_s(\epsilon) = \#_s(\mathit{Sub}(D, N, \vec{\ell}, T).S') &= 0, \\ \#_s(\mathit{Box}(Q, \vec{\ell}).S') = \#_s(\mathit{Force}.S') = \#_s(\mathit{Let}(x, y, M).S') &= \#_s(S'), \\ \#_s(\mathit{FArg}(M).S') = \#_s(\mathit{ALabel}(M).S') &= \#_t(M) + 1 + \#_s(S'), \\ \#_s(\mathit{TRight}(M).S') &= \#_t(M) + 2 + \#_s(S'), \\ \#_s(\mathit{FApp}(V).S') = \#_s(\mathit{ACirc}(V).S') &= \#_s(S'), \\ \#_s(\mathit{TLeft}(V).S') &= 1 + \#_s(S'). \end{aligned}$$

Finally, let $\#(C, M, S) = \#_t(M) + \#_s(S)$. It is trivial to see that both $\#_t$ and $\#_s$, and as a consequence $\#$, are non-negative. Because of this, it is sufficient to show that whenever $(C, M, S) \Rightarrow_b (D, N, S')$ then $\#(D, N, S') < \#(C, M, S)$ to prove that $\Rightarrow_b$ is strongly normalizing. We proceed by cases on the introduction of $(C, M, S) \Rightarrow_b (D, N, S')$:

- Case of *app-split*. In this case $(C, MN, S) \Rightarrow_b (C, M, \mathit{FArg}(N).S)$. We have $\#(C, MN, S) = \#_t(M) + \#_t(N) + 2 + \#_s(S)$ and $\#(C, M, \mathit{FArg}(N).S) = \#_t(M) + \#_t(N) + 1 + \#_s(S)$ and the claim is proven.

- Case of *app-shift*. In this case $(C, V, \mathit{FArg}(N).S) \Rightarrow_b (C, N, \mathit{FApp}(V).S)$. We have $\#(C, V, \mathit{FArg}(N).S) = \#_t(N) + 1 + \#_s(S)$ and $\#(C, N, \mathit{FApp}(V).S) = \#_t(N) + \#_s(S)$ and the claim is proven.

- Case of *apply-split*. In this case $(C, \mathsf{apply}(M, N), S) \Rightarrow_b (C, M, \mathit{ALabel}(N).S)$. We have $\#(C, \mathsf{apply}(M, N), S) = \#_t(M) + \#_t(N) + 2 + \#_s(S)$ and $\#(C, M, \mathit{ALabel}(N).S) = \#_t(M) + \#_t(N) + 1 + \#_s(S)$ and the claim is proven.

- Case of *apply-shift*. In this case $(C, V, \mathit{ALabel}(N).S) \Rightarrow_b (C, N, \mathit{ACirc}(V).S)$. We have $\#(C, V, \mathit{ALabel}(N).S) = \#_t(N) + 1 + \#_s(S)$ and $\#(C, N, \mathit{ACirc}(V).S) = \#_t(N) + \#_s(S)$ and the claim is proven.

- Case of *let-split*. In this case $(C, \mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N, S) \Rightarrow_b (C, M, \mathit{Let}(x, y, N).S)$. We have $\#(C, \mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N, S) = \#_t(M) + 1 + \#_s(S)$ and $\#(C, M, \mathit{Let}(x, y, N).S) = \#_t(M) + \#_s(S)$ and the claim is proven.

- Case of *tuple-split*. In this case $(C, \langle M, N\rangle, S) \Rightarrow_b (C, M, \mathit{TRight}(N).S)$. We have $\#(C, \langle M, N\rangle, S) = \#_t(M) + \#_t(N) + 3 + \#_s(S)$ and $\#(C, M, \mathit{TRight}(N).S) = \#_t(M) + \#_t(N) + 2 + \#_s(S)$ and the claim is proven.

- Case of *tuple-shift*. In this case $(C, V, \mathit{TRight}(N).S) \Rightarrow_b (C, N, \mathit{TLeft}(V).S)$. We have $\#(C, V, \mathit{TRight}(N).S) = \#_t(N) + 2 + \#_s(S)$ and $\#(C, N, \mathit{TLeft}(V).S) = \#_t(N) + 1 + \#_s(S)$ and the claim is proven.

- Case of *tuple-join*. In this case $(C, W, \mathit{TLeft}(V).S) \Rightarrow_b (C, \langle V, W\rangle, S)$. We have $\#(C, W, \mathit{TLeft}(V).S) = \#_s(S) + 1$ and $\#(C, \langle V, W\rangle, S) = \#_s(S)$ and the claim is proven.

- Case of *box-open*. In this case $(C, \mathsf{box}_T\, M, S) \Rightarrow_b (C, M, \mathit{Box}(Q, \vec{\ell}).S)$. We have $\#(C, \mathsf{box}_T\, M, S) = \#_t(M) + 1 + \#_s(S)$ and $\#(C, M, \mathit{Box}(Q, \vec{\ell}).S) = \#_t(M) + \#_s(S)$ and the claim is proven.

- Case of *force-open*. In this case $(C, \mathsf{force}\, M, S) \Rightarrow_b (C, M, \mathit{Force}.S)$. We have $\#(C, \mathsf{force}\, M, S) = \#_t(M) + 1 + \#_s(S)$ and $\#(C, M, \mathit{Force}.S) = \#_t(M) + \#_s(S)$ and the claim is proven.

□

Now we can prove the most significant result of this section, which is similar to the one we gave in Lemma 4.1.1. Namely, we prove that a single reduction step in the machine semantics can always be simulated by zero or one steps in the stacked semantics.

Lemma 4.2.4. *Suppose* (C, M, S) *and* (C', M', S') *are two machine configurations such that* $(C, M, S) \Rightarrow (C', M', S')$. *The following hold:*

1. *If* $(C, M, S) \Rightarrow_b (C', M', S')$, *then* $\mathsf{fromMachine}(C, M, S) = \mathsf{fromMachine}(C', M', S')$,

2. *If* $(C, M, S) \Rightarrow_r (C', M', S')$, *then* $\mathsf{fromMachine}(C, M, S) \rightharpoonup \mathsf{fromMachine}(C', M', S')$.

Proof. By cases on the introduction of $(C, M, S) \Rightarrow (C', M', S')$:

- Case of *app-split*. In this case we have $(C, NP, S) \Rightarrow_b (C, N, \mathit{FArg}(P).S)$ and we immediately conclude $\mathsf{fromMachine}(C, N, \mathit{FArg}(P).S) = \mathsf{fromMachine}(C, NP, S)$ by the definition of $\mathsf{fromMachine}$.

- Case of *app-shift*. In this case we have $(C, V, \mathit{FArg}(P).S) \Rightarrow (C, P, \mathit{FApp}(V).S)$ and we immediately conclude

$$\begin{aligned}\mathsf{fromMachine}(C, V, \mathit{FArg}(P).S) &= \mathsf{fromMachine}(C, VP, S)\\ &= \mathsf{fromMachine}(C, P, \mathit{FApp}(V).S).\end{aligned}$$

- Case of *app-join*. In this case we have $(C, V, \mathit{FApp}(\lambda x.N).S) \Rightarrow_r (C, N[V/x], S)$. By propositions 4.2.1 and 4.2.2 we know that

$$\begin{aligned}\mathsf{fromMachine}(C, V, \mathit{FApp}(\lambda x.N).S) &= \mathsf{fromMachine}(C, (\lambda x.N)V, S)\\ &= (C, E[(\lambda x.N)V])^{\vec{\ell}}.X,\end{aligned}$$

$$\mathsf{fromMachine}(C, N[V/x], S) = (C, E[N[V/x]])^{\vec{\ell}}.X.$$

  Because $(C, (\lambda x.N)V) \to (C, N[V/x])$ by the *app* rule, we get $(C, E[(\lambda x.N)V]) \to (C, E[N[V/x]])$ by Theorem 2.5.3. Furthermore, by Corollary 2.5.6.1 we know that $E[(\lambda x.N)V] \not\equiv E'[\mathsf{box}_T(\mathsf{lift}\ P)]$ for any E', P and therefore by the *head* rule we conclude

$$(C, E[(\lambda x.N)V])^{\vec{\ell}}.X \rightharpoonup (C, E[N[V/x]])^{\vec{\ell}}.X.$$

- Case of *apply-split*. In this case we have $(C, \mathsf{apply}(N, P), S) \Rightarrow_b (C, N, \mathit{ALabel}(P).S)$ and we conclude $\mathsf{fromMachine}(C, N, \mathit{ALabel}(P).S) = \mathsf{fromMachine}(C, \mathsf{apply}(N, P), S)$ by the definition of $\mathsf{fromMachine}$.

- Case of *apply-shift*. In this case we have $(C, V, \mathit{ALabel}(P).S) \Rightarrow_b (C, P, \mathit{ACirc}(V).S)$ and we immediately conclude

$$\begin{aligned}\mathsf{fromMachine}(C, V, \mathit{ALabel}(P).S) &= \mathsf{fromMachine}(C, \mathsf{apply}(V, P), S)\\ &= \mathsf{fromMachine}(C, P, \mathit{ACirc}(V).S).\end{aligned}$$

- Case of *apply-join*. In this case we have $(C, \vec{k}, ACirc((\vec{\ell}, D, \vec{\ell'})).S) \Rightarrow_r (C', \vec{k'}, S)$, where $(C', \vec{k'}) = \mathsf{append}(C, \vec{k}, \vec{\ell}, D, \vec{\ell'})$. By propositions 4.2.1 and 4.2.2 we know that

$$\begin{aligned}\mathsf{fromMachine}(C, \vec{k}, ACirc((\vec{\ell}, D, \vec{\ell'})).S) &= \mathsf{fromMachine}(C, \mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k}), S) \\ &= (C, E[\mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})])^{\vec{\ell''}}.X,\end{aligned}$$

$$\mathsf{fromMachine}(C, \vec{k'}, S) = (C, E[\vec{k'}])^{\vec{\ell''}}.X.$$

  Because $(C, \mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})) \rightarrow (C, \vec{k'})$ by the *apply* rule, we have that $(C, E[\mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})]) \rightarrow (C, E[\vec{k'}])$ by Theorem 2.5.3. Furthermore, by Corollary 2.5.6.1 we know that $E[\mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})] \not\equiv E'[\mathsf{box}_T(\mathsf{lift}\, P)]$ for any E', P and therefore by the *head* rule we conclude

$$(C, E[\mathsf{apply}((\vec{\ell}, D, \vec{\ell'}), \vec{k})])^{\vec{\ell''}}.X \rightharpoonup (C, E[\vec{k'}])^{\vec{\ell''}}.X.$$

- Case of *tuple-split*. In this case we have $(C, \langle N, P\rangle, S) \Rightarrow_b (C, N, TRight(P).S)$ and we immediately conclude $\mathsf{fromMachine}(C, N, TRight(P).S) = \mathsf{fromMachine}(C, \langle N, P\rangle, S)$ by the definition of $\mathsf{fromMachine}$.

- Case of *tuple-shift*. In this case we have $(C, V, TRight(P).S) \Rightarrow_b (C, P, TLeft(V).S)$ and we immediately conclude

$$\begin{aligned}\mathsf{fromMachine}(C, V, TRight(P).S) &= \mathsf{fromMachine}(C, \langle V, P\rangle, S) \\ &= \mathsf{fromMachine}(C, P, TLeft(V).S).\end{aligned}$$

- Case of *tuple-join*. In this case we have $(C, W, TLeft(V).S) \Rightarrow_b (C, \langle V, W\rangle, S)$ and we immediately conclude $\mathsf{fromMachine}(C, W, TLeft(V).S) = \mathsf{fromMachine}(C, \langle V, W\rangle, S)$ by the definition of $\mathsf{fromMachine}$.

- Case of *box-open*. In this case we have $(C, \mathsf{box}_T\, N, S) \Rightarrow_b (C, N, Box(Q, \vec{\ell}).S)$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. Since we know by the definition of $\mathsf{freshlabels}$ that $\emptyset; Q \vdash \vec{\ell} : T$, we immediately conclude $\mathsf{fromMachine}(C, N, Box(Q, \vec{\ell}).S) = \mathsf{fromMachine}(C, \mathsf{box}_T\, N, S)$.

- Case of *box-sub*. In this case $(C, \mathsf{lift}\, N, Box(Q, \vec{\ell}).S) \Rightarrow_r (id_Q, N\vec{\ell}, Sub(C, N, \vec{\ell}, T).S)$, where $\emptyset; Q \vdash \vec{\ell} : T$. By propositions 4.2.1 and 4.2.2 we know that

$$\begin{aligned}\mathsf{fromMachine}(C, \mathsf{lift}\, N, Box(Q, \vec{\ell}).S) &= \mathsf{fromMachine}(C, \mathsf{box}_T(\mathsf{lift}\, N), S) \\ &= (C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{\ell'}}.X,\end{aligned}$$

$$\begin{aligned}\mathsf{fromMachine}(id_Q, N\vec{\ell}, Sub(C, N, \vec{\ell}, T).S) &= (id_Q, N\vec{\ell})^{\vec{\ell}}.\, \mathsf{fromMachine}(C, \mathsf{box}_T(\mathsf{lift}\, N), S) \\ &= (id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{\ell'}}.X,\end{aligned}$$

and by the *step-in* rule we conclude

$$(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{\ell'}}.X \rightharpoonup (id_Q, N\vec{\ell})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{\ell'}}.X.$$

- Case of *box-close*. In this case we have $(D, \vec{\ell'}, Sub(C, N, \vec{\ell}, T).S) \Rightarrow_r (C, (\vec{\ell}, D, \vec{\ell'}), S)$. By propositions 4.2.1 and 4.2.2 we know that

$$\begin{aligned}\mathsf{fromMachine}(D, \vec{\ell'}, Sub(C, N, \vec{\ell}, T).S) &= (D, \vec{\ell'})^{\vec{\ell}}.\, \mathsf{fromMachine}(C, \mathsf{box}_T(\mathsf{lift}\, N), S) \\ &= (D, \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X,\end{aligned}$$

$$\mathsf{fromMachine}(C, (\vec{\ell}, D, \vec{\ell'}), S) = (C, E[(\vec{\ell}, D, \vec{\ell'})])^{\vec{k}}.X,$$

and by the *step-out* rule we conclude

$$(D, \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X \rightharpoonup (C, E[(\vec{\ell}, D, \vec{\ell'})])^{\vec{k}}.X.$$

- Case of *let-split*. In this case we have $(C, \mathsf{let}\, \langle x, y\rangle = N\, \mathsf{in}\, P, S) \Rightarrow_b (C, N, Let(x, y, P).S)$ and we conclude $\mathsf{fromMachine}(C, N, Let(x, y, P).S) = \mathsf{fromMachine}(C, \mathsf{let}\, \langle x, y\rangle = N\, \mathsf{in}\, P, S)$ by the definition of $\mathsf{fromMachine}$.

- Case of *let-join*. In this case we have $(C, \langle V, W\rangle, Let(x, y, N).S) \Rightarrow_r (C, N[V/x][W/y], S)$. By propositions 4.2.1 and 4.2.2 we know that

$$\begin{aligned}\mathsf{fromMachine}(C, \langle V, W\rangle, Let(x, y, N).S) &= \mathsf{fromMachine}(C, \mathsf{let}\, \langle x, y\rangle = \langle V, W\rangle\, \mathsf{in}\, N, S) \\ &= (C, E[\mathsf{let}\, \langle x, y\rangle = \langle V, W\rangle\, \mathsf{in}\, N])^{\vec{\ell}}.X,\end{aligned}$$

$$\mathsf{fromMachine}(C, N[V/x][W/y], S) = (C, E[N[V/x][W/y]])^{\vec{\ell}}.X.$$

Because $(C, \mathsf{let}\, \langle x, y\rangle = \langle V, W\rangle\, \mathsf{in}\, N) \rightarrow (C, N[V/x][W/y])$ by the *let* rule, we get $(C, E[\mathsf{let}\, \langle x, y\rangle = \langle V, W\rangle\, \mathsf{in}\, N]) \rightarrow (C, E[N[V/x][W/y]])$ by Theorem 2.5.3 and by the *head* rule we conclude

$$(C, E[\mathsf{let}\, \langle x, y\rangle = \langle V, W\rangle\, \mathsf{in}\, N])^{\vec{\ell}}.X \rightharpoonup (C, E[N[V/x][W/y]])^{\vec{\ell}}.X.$$

- Case of *force-open*. In this case we have $(C, \mathsf{force}\, N, S) \Rightarrow_b (C, N, Force.S)$ and we immediately conclude $\mathsf{fromMachine}(C, N, Force.S) = \mathsf{fromMachine}(C, \mathsf{force}\, N, S)$ by the definition of $\mathsf{fromMachine}$.

- Case of *force-close*. In this case we have $(C, \mathsf{lift}\, N, \mathit{Force}.S) \Rightarrow_r (C, N, S)$. By propositions 4.2.1 and 4.2.2 we know that

$$\begin{aligned}\mathsf{fromMachine}(C, \mathsf{lift}\, N, \mathit{Force}.S) &= \mathsf{fromMachine}(C, \mathsf{force}(\mathsf{lift}\, N), S)\\ &= (C, E[\mathsf{force}(\mathsf{lift}\, N)])^{\vec{\ell}}.X,\end{aligned}$$

$$\mathsf{fromMachine}(C, N, S) = (C, E[N])^{\vec{\ell}}.X.$$

  Because $(C, \mathsf{force}(\mathsf{lift}\, N)) \to (C, N)$ by the *force* rule, we get $(C, E[\mathsf{force}(\mathsf{lift}\, N)]) \to (C, E[N])$ by Theorem 2.5.3 and by the *head* rule we conclude

$$(C, E[\mathsf{force}(\mathsf{lift}\, N)])^{\vec{\ell}}.X \rightharpoonup (C, E[N])^{\vec{\ell}}.X.$$

□

Like we did in Section 4.1.2, it is useful to represent the result of the lemma that we just proved in a diagrammatic way, as follows:

$$\begin{array}{ccc}(C, E[M])^{\vec{\ell}}.X & \rightharpoonup & (D, F[N])^{\vec{k}}.Y\\ \uparrow{\scriptstyle \mathsf{fromMachine}} & & \uparrow{\scriptstyle \mathsf{fromMachine}}\\ (C, M, S) & \Longrightarrow_r & (D, N, R)\end{array} \qquad \begin{array}{ccc} & (C, E[M])^{\vec{\ell}}.X & \\ {\scriptstyle \mathsf{fromMachine}}\nearrow & & \nwarrow{\scriptstyle \mathsf{fromMachine}}\\ (C, M, S) & \Longrightarrow_b & (D, N, R)\end{array}$$

Or, more generally and synthetically, as a single diagram:

$$\begin{array}{ccc}(C, E[M])^{\vec{\ell}}.X & \rightharpoonup^* & (D, F[N])^{\vec{k}}.Y\\ \uparrow{\scriptstyle \mathsf{fromMachine}} & & \uparrow{\scriptstyle \mathsf{fromMachine}}\\ (C, M, S) & \Longrightarrow^+ & (D, N, R)\end{array}$$

4.2.3 Convergence, Deadlock and Divergence

In order to prove the equivalence between the small-step and machine semantics in the next section, it is sufficient to prove that fromMachine preserves the convergence of reachable configurations and that whenever (C, M, S) goes into deadlock then also $\mathsf{fromMachine}(C, M, S)$ goes into deadlock.

Let us start with deadlock. Informally, a machine configuration goes into deadlock when it evaluates to an irreducible configuration whose stack is not empty. Similarly, a stacked configuration goes into deadlock when it evaluates to an irreducible configuration in which either the stack is not empty, or the term in the head is not a value. Because we know that a computation in the machine semantics can always be simulated by a computation in the stacked semantics, to show that whenever a machine configuration goes into deadlock then the corresponding stacked configuration also goes into deadlock it is sufficient to show that every irreducible machine configuration (C, M, S) such that S is not empty is mapped by $\mathsf{fromMachine}$ to an irreducible stacked configuration $(C, E[M])^{\vec{\ell}}.X$ such that either $E[M]$ is not a value or X is not empty.

Lemma 4.2.5. *Suppose (C, V, S) is a machine configuration. If (C, V, S) is irreducible and $S \neq \epsilon$, then $\mathsf{fromMachine}(C, V, S) = (C, E[V])^{\vec{\ell}}.X$ is irreducible and either $E[V]$ is not a value or $\vec{\ell} \neq \emptyset, X \neq \epsilon$.*

Proof. Suppose $S = H.S'$. We proceed by cases on H:

- Case of $FArg(N)$. This case is impossible, since it would allow (C, V, S) to be reduced by the *app-shift* rule, regardless of V, contradicting the hypothesis.
- Case of $FApp(W)$. In this case $W \not\equiv \lambda x.N$, since otherwise $(C, V, FApp(\lambda x.N).S')$ would be reducible by the *app-join* rule. We have $\mathsf{fromMachine}(C, V, FApp(W).S') = \mathsf{fromMachine}(C, WV, S') = (C, E[WV])^{\vec{k}}.X$. This configuration cannot be reduced by the *head* rule (since by Theorem 2.5.3 the reducibility of $(C, E[WV])$ would imply the reducibility of (C, WV), which is absurd since $W \not\equiv \lambda x.N$), nor by the *step-in* rule (since WV is a proto-redex and therefore by Proposition 2.5.6 $E[WV] \not\equiv F[\mathsf{box}_T(\mathsf{lift}\, P)]$, for any F, P), nor by the *step-out* rule (since $E[WV]$ cannot be a value). Since $(C, E[WV])^{\vec{k}}.X$ is irreducible and $E[WV]$ is not a value, the claim is proven.
- Case of $ALabel(N)$. This case is impossible, since it would allow (C, V, S) to be reduced by the *apply-shift* rule, regardless of V, contradicting the hypothesis.
- Case of $ACirc(W)$. In this case $W \not\equiv (\vec{\ell}, D, \vec{\ell'})$, or else $(C, V, ACirc((\vec{\ell}, D, \vec{\ell'})).S')$ would be reducible by the *apply-join* rule. We have $\mathsf{fromMachine}(C, V, ACirc(W).S') = \mathsf{fromMachine}(C, \mathsf{apply}(W, V), S') = (C, E[\mathsf{apply}(W, V)])^{\vec{k}}.X$. This configuration cannot be reduced by the *head* rule (since by Theorem 2.5.3 the reducibility of $(C, E[\mathsf{apply}(W, V)])$ would imply the reducibility of $(C, \mathsf{apply}(W, V))$, which is absurd since $W \not\equiv (\vec{\ell}, D, \vec{\ell'})$), nor by the *step-in* rule (since $\mathsf{apply}(W, V)$ is a proto-redex and therefore by Proposition 2.5.6 $E[\mathsf{apply}(W, V)] \not\equiv F[\mathsf{box}_T(\mathsf{lift}\, P)]$, for any F, P), nor by the *step-out* rule (since $E[\mathsf{apply}(W, V)]$ cannot be a value). Since $(C, E[\mathsf{apply}(W, V)])^{\vec{k}}.X$ is irreducible and $E[\mathsf{apply}(W, V)]$ is not a value, the claim is proven.

- Case of $\mathit{TRight}(N)$. This case is impossible, since it would allow (C, V, S) to be reduced by the *tuple-shift* rule, regardless of V, contradicting the hypothesis.

- Case of $\mathit{TLeft}(W)$. This case is impossible, since it would allow (C, V, S) to be reduced by the *tuple-join* rule, regardless of V, contradicting the hypothesis.

- Case of $\mathit{Box}(Q, \vec{\ell})$. In this case $V \not\equiv \mathsf{lift}\, N$, since otherwise $(C, \mathsf{lift}\, N, \mathit{Box}(Q, \vec{\ell}).S')$ would be reducible by the *box-sub* rule. We have $\mathsf{fromMachine}(C, V, \mathit{Box}(Q, \vec{\ell}).S') = \mathsf{fromMachine}(C, \mathsf{box}_T\, V, S') = (C, E[\mathsf{box}_T\, V])^{\vec{k}}.X$, where $\emptyset; Q \vdash \vec{\ell} : T$. This configuration cannot be reduced by the *head* rule (since by Theorem 2.5.3 the reducibility of $(C, E[\mathsf{box}_T\, V])$ would imply the reducibility of $(C, \mathsf{box}_T\, V)$, which is absurd since $V \not\equiv \mathsf{lift}\, N$), nor by the *step-in* rule (since $V \not\equiv \mathsf{lift}\, N$ and $\mathsf{box}_T\, V$ is a proto-redex, so by Proposition 2.5.6 $E[\mathsf{box}_T\, V] \not\equiv F[\mathsf{box}_T(\mathsf{lift}\, P)]$ for any other F, P), nor by the *step-out* rule (since $E[\mathsf{box}_T(\mathsf{lift}\, N)]$ cannot be a value). Since $(C, E[\mathsf{box}_T\, V])^{\vec{k}}.X$ is irreducible and $E[\mathsf{box}_T\, V]$ is not a value, the claim is proven.

- Case of $\mathit{Sub}(D, N, \vec{\ell}, T)$. In this case $V \not\equiv \vec{\ell'}$, or else $(C, \vec{\ell'}, \mathit{Sub}(D, N, \vec{\ell}, T).S')$ would be reducible by the *box-close* rule. We have $\mathsf{fromMachine}(C, V, \mathit{Sub}(D, N, \vec{\ell}, T).S') = (C, V)^{\vec{\ell}}.\, \mathsf{fromMachine}(D, \mathsf{box}_T(\mathsf{lift}\, N), S') = (C, V)^{\vec{\ell}}.(D, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$. This configuration cannot be reduced by the *head* rule (since V is a value), nor by the *step-in* rule (since V cannot be of the form $F[\mathsf{box}_T(\mathsf{lift}\, P)]$, for any F, P), nor by the *step-out* rule (since $V \not\equiv \vec{\ell'}$). Since $(C, V)^{\vec{\ell}}.(D, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$ is irreducible and $(D, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X \neq \epsilon$, the claim is proven.

- Case of $\mathit{Let}(x, y, N)$. In this case we know $V \not\equiv \langle V', V'' \rangle$, since otherwise $(C, \langle V', V'' \rangle, \mathit{Let}(x, y, N).S')$ would be reducible by the *let-join* rule. We have $\mathsf{fromMachine}(C, V, \mathit{Let}(x, y, N).S') = \mathsf{fromMachine}(C, \mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N, S') = (C, E[\mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N])^{\vec{k}}.X$. This configuration cannot be reduced by the *head* rule (since by Theorem 2.5.3 the reducibility of $(C, E[\mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N])$ would imply the reducibility of $(C, \mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N)$, which is absurd since $V \not\equiv \langle V', V'' \rangle$), nor by the *step-in* rule (since $\mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N$ is a proto-redex and therefore by Proposition 2.5.6 $E[\mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N] \not\equiv F[\mathsf{box}_T(\mathsf{lift}\, P)]$, for any F, P), nor by the *step-out* rule (since $E[\mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N]$ cannot be a value). Since $(C, E[\mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N])^{\vec{k}}.X$ is irreducible and $E[\mathsf{let}\, \langle x, y \rangle = V\, \mathsf{in}\, N]$ is not a value, the claim is proven.

- Case of *Force*. In this case we know $V \not\equiv \mathsf{lift}\, N$, since otherwise $(C, \mathsf{lift}\, N, \mathit{Force}.S')$ would be reducible by the *force-close* rule. We have $\mathsf{fromMachine}(C, V, \mathit{Force}.S') = \mathsf{fromMachine}(C, \mathsf{force}\, V, S') = (C, E[\mathsf{force}\, V])^{\vec{k}}.X$. This configuration cannot be reduced by the *head* rule (since by Theorem 2.5.3 the reducibility of $(C, E[\mathsf{force}\, V])$ would imply the reducibility of $(C, \mathsf{force}\, V)$, which is absurd since $V \not\equiv \mathsf{lift}\, N$), nor by the *step-in* rule (since $\mathsf{force}\, V$ is a proto-redex and therefore by Proposition 2.5.6

$E[\mathsf{force}\, N] \not\equiv F[\mathsf{box}_T(\mathsf{lift}\, P)]$, for any F, P), nor by the *step-out* rule (since $E[\mathsf{force}\, V]$ cannot be a value). Since $(C, E[\mathsf{force}\, V])^{\vec{k}}.X$ is irreducible and $E[\mathsf{force}\, V]$ is not a value, the claim is proven.

□

Lemma 4.2.6. *Suppose (C, M, S) is a machine configuration. If $(C, M, S)\bot$, then* $\mathsf{fromMachine}(C, M, S)\bot$.

Proof. It is easy to see that a machine configuration (C, M, S) goes into deadlock if and only if $(C, M, S) \Rightarrow^* (D, V, R)$ for some irreducible (D, V, R) such that $R \neq \epsilon$. It is also easy to see that a stacked configuration $(C, M)^{\vec{\ell}}.X$ goes into deadlock if and only if $(C, M)^{\vec{\ell}}.X \rightharpoonup^* (D, N)^{\vec{k}}.Y$ for some irreducible $(D, N)^{\vec{k}}.Y$ such that either N is not a value or $\vec{k} \neq \emptyset, Y \neq \epsilon$. By a finite number of applications of Lemma 4.2.4 we know that $\mathsf{fromMachine}(C, M, S) \rightharpoonup^* \mathsf{fromMachine}(D, V, R)$ and by Lemma 4.2.5 we know that $\mathsf{fromMachine}(D, V, R) = (D, E[V])^{\vec{k}}.Y$ for some $E, \vec{k}, Y$ such that either $E[V]$ is not a value or $\vec{k} \neq \emptyset, Y \neq \epsilon$. We therefore conclude that $\mathsf{fromMachine}(C, M, S)\bot$. □

Having proven this result first, it is now easier to prove that fromMachine preserves the convergence of reachable configurations. However, in order to proceed, we must first prove that fromMachine preserves reachability.

Lemma 4.2.7. *If (C, M, S) is a reachable machine configuration, then we have that* $\mathsf{fromMachine}(C, M, S)$ *is a reachable stacked configuration.*

Proof. By induction on the reachability of (C, M, S). In the case in which $S = \epsilon$ we have $\mathsf{fromMachine}(C, M, \epsilon) = (C, M)^{\emptyset}.\epsilon \in \mathcal{I}_{st}$, which is reachable. In the case in which there exists (D, N, S') such that (D, N, S') is reachable and $(D, N, S') \Rightarrow (C, M, S)$ we proceed by cases on the introduction of $(D, N, S') \Rightarrow (C, M, S)$:

- Case of *app-split*. In this case we have $(D, NP, S') \Rightarrow (D, N, \mathit{FArg}(P).S')$. By inductive hypothesis we know that $\mathsf{fromMachine}(D, NP, S')$ is reachable, so we immediately conclude that $\mathsf{fromMachine}(D, N, \mathit{FArg}(P).S') = \mathsf{fromMachine}(D, NP, S')$ is reachable.

- Case of *app-shift*. In this case we have $(D, V, \mathit{FArg}(P).S') \Rightarrow (D, P, \mathit{FApp}(V).S')$. By inductive hypothesis we know that $\mathsf{fromMachine}(D, V, \mathit{FArg}(P).S')$ is reachable. Also, by the definition of fromMachine we have

$$\begin{aligned}\mathsf{fromMachine}(D, V, \mathit{FArg}(P).S') &= \mathsf{fromMachine}(D, VP, S') \\ &= \mathsf{fromMachine}(D, P, \mathit{FApp}(V).S'),\end{aligned}$$

  so we conclude that $\mathsf{fromMachine}(D, P, \mathit{FApp}(V).S')$ is reachable.

- Case of *app-join*. In this case we have $(D, V, FArg(\lambda x.N).S') \Rightarrow (D, N[V/x], S')$. By propositions 4.2.1 and 4.2.2 we know that $\mathsf{fromMachine}(D, V, FArg(\lambda x.N).S') = \mathsf{fromMachine}(D, (\lambda x.N)V, S') = (D, E[(\lambda x.N)V])^{\vec{\ell}}.X, \mathsf{fromMachine}(D, N[V/x], S') = (D, E[N[V/x]])^{\vec{\ell}}.X$ for the same $E, \vec{\ell}, X$, and by inductive hypothesis we know that $(D, E[(\lambda x.N)V])^{\vec{\ell}}.X$ is reachable. Because $(D, (\lambda x.N)V) \to (D, N[V/x])$ by the *β-reduction* rule, by Theorem 2.5.3 we get $(D, E[(\lambda x.N)V]) \to (D, E[N[V/x]])$. Also, because $(\lambda x.N)V$ is a redex, we know by Corollary 2.5.6.1 that $E[(\lambda x.N)V] \not\equiv E'[\mathsf{box}_T(\mathsf{lift}\, P)]$ for any E', so we get $(D, E[(\lambda x.N)V])^{\vec{\ell}}.X \rightharpoonup (D, E[N[V/x]])^{\vec{\ell}}.X$ by the *head* rule and conclude that $(D, E[N[V/x]])^{\vec{\ell}}.X$ is reachable.

- Case of *apply-split*. In this case we have $(D, \mathsf{apply}(N, P), S') \Rightarrow (D, N, ALabel(P).S')$. By inductive hypothesis we know that $\mathsf{fromMachine}(D, \mathsf{apply}(N, P), S')$ is reachable, so we conclude that $\mathsf{fromMachine}(D, N, ALabel(P).S') = \mathsf{fromMachine}(D, \mathsf{apply}(N, P), S')$ is reachable.

- Case of *apply-shift*. In this case we have $(D, V, ALabel(P).S') \Rightarrow (D, P, ACirc(V).S')$. By inductive hypothesis we know that $\mathsf{fromMachine}(D, V, ALabel(P).S')$ is reachable. Also, by the definition of $\mathsf{fromMachine}$ we have

$$\begin{aligned}\mathsf{fromMachine}(D, V, ALabel(P).S') &= \mathsf{fromMachine}(D, \mathsf{apply}(N, P), S') \\ &= \mathsf{fromMachine}(D, P, ACirc(V).S'),\end{aligned}$$

  so we conclude that $\mathsf{fromMachine}(D, P, ACirc(V).S')$ is reachable.

- Case of *apply-join*. Case of *app-join*. In this case we have $(D, \vec{k}, ACirc((\vec{\ell}, D', \vec{\ell'})).S') \Rightarrow (C, \vec{k'}, S')$, where $(C, \vec{k'}) = \mathsf{append}(D, \vec{k}, \vec{\ell}, D', \vec{\ell'})$. By propositions 4.2.1 and 4.2.2 we know $\mathsf{fromMachine}(D, \vec{k}, ACirc((\vec{\ell}, D', \vec{\ell'})).S') = \mathsf{fromMachine}(D, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k}), S') = (D, E[\mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})])^{\vec{\ell''}}.X$ and $\mathsf{fromMachine}(C, \vec{k'}, S') = (C, E[\vec{k'}])^{\vec{\ell''}}.X$, for the same $E, \vec{\ell''}, X$, and by inductive hypothesis we know $(D, E[\mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})])^{\vec{\ell''}}.X$ is reachable. Because $(D, \mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})) \to (C, \vec{k'})$ by the *apply* rule, by Theorem 2.5.3 we get $(D, E[\mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})]) \to (C, E[\vec{k'}])$. Also, because $\mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})$ is a redex, we know by Corollary 2.5.6.1 that $E[\mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})] \not\equiv E'[\mathsf{box}_T(\mathsf{lift}\, P)]$ for any E', so we get $(D, E[\mathsf{apply}((\vec{\ell}, D', \vec{\ell'}), \vec{k})])^{\vec{\ell''}}.X \rightharpoonup (C, E[\vec{k'}])^{\vec{\ell''}}.X$ by the *head* rule and conclude that $(C, E[\vec{k'}])^{\vec{\ell''}}.X$ is reachable.

- Case of *tuple-split*. In this case we have $(D, \langle N, P\rangle, S') \Rightarrow (D, N, TRight(P).S')$. By inductive hypothesis we know that $\mathsf{fromMachine}(D, \langle N, P\rangle, S')$ is reachable, so we immediately conclude that $\mathsf{fromMachine}(D, N, TRight(P).S') = \mathsf{fromMachine}(D, \langle N, P\rangle, S')$ is reachable.

- Case of *tuple-shift*. In this case we have $(D, V, \mathit{TRight}(P).S') \Rightarrow (D, P, \mathit{TLeft}(V).S')$. By inductive hypothesis we know that $\mathsf{fromMachine}(D, V, \mathit{TRight}(P).S')$ is reachable. Also, by the definition of $\mathsf{fromMachine}$ we have

$$\begin{aligned}\mathsf{fromMachine}(D, V, \mathit{TRight}(P).S') &= \mathsf{fromMachine}(D, \langle V, P\rangle, S') \\ &= \mathsf{fromMachine}(D, P, \mathit{TLeft}(V).S'),\end{aligned}$$

  so we conclude that $\mathsf{fromMachine}(D, P, \mathit{TLeft}(V).S')$ is reachable.

- Case of *tuple-join*. In this case we have $(D, W, \mathit{TLeft}(V).S') \Rightarrow (D, \langle V, W\rangle, S')$. By inductive hypothesis we immediately know that $\mathsf{fromMachine}(D, W, \mathit{TLeft}(V).S') = \mathsf{fromMachine}(D, \langle V, W\rangle, S')$ is reachable and the claim is trivially true.

- Case of *box-open*. In this case we have $(D, \mathsf{box}_T\, N, S') \Rightarrow (D, N, \mathit{Box}(Q, \vec{\ell}).S')$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$. By inductive hypothesis we know that the configuration $\mathsf{fromMachine}(D, \mathsf{box}_T\, N, S')$ is reachable, so we immediately conclude that $\mathsf{fromMachine}(D, N, \mathit{Box}(Q, \vec{\ell}).S') = \mathsf{fromMachine}(D, \mathsf{box}_T\, N, S')$ is reachable.

- Case of *box-sub*. In this case $(D, \mathsf{lift}\, N, \mathit{Box}(Q, \vec{\ell}).S') \Rightarrow (id_Q, N\vec{\ell}, \mathit{Sub}(D, N, \vec{\ell}, T).S')$. By inductive hypothesis we know that $\mathsf{fromMachine}(D, \mathsf{lift}\, N, \mathit{Box}(Q, \vec{\ell}).S') = \mathsf{fromMachine}(D, \mathsf{box}_T(\mathsf{lift}\, N), S')$ is reachable. We also know that

$$\begin{aligned}&\mathsf{fromMachine}(id_Q, N\vec{\ell}, \mathit{Sub}(D, N, \vec{\ell}, T).S') \\ &= (id_Q, N\vec{\ell})^{\vec{\ell}}.\, \mathsf{fromMachine}(D, \mathsf{box}_T(\mathsf{lift}\, N), S').\end{aligned}$$

  By Proposition 4.2.1 we get $\mathsf{fromMachine}(D, \mathsf{box}_T(\mathsf{lift}\, N), S') = (D, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$ for some $E, \vec{k}$ and X. Finally, by the *step-in* rule we have $(D, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{\ell}}.X \rightharpoonup (id_Q, N\vec{\ell})^{\vec{\ell}}.(D, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{\ell}}.X$, where $(Q, \vec{\ell}) = \mathsf{freshlabels}(N, T)$, and conclude that the latter is reachable.

- Case of *box-close*. In this case we have $(D, \vec{\ell'}, \mathit{Sub}(C, N, \vec{\ell}, T).S') \Rightarrow (C, (\vec{\ell}, D, \vec{\ell'}), S')$. By propositions 4.2.1 and 4.2.2 we know that $\mathsf{fromMachine}(D, \vec{\ell'}, \mathit{Sub}(C, N, \vec{\ell}, T).S') = (D, \vec{\ell'})^{\vec{\ell}}.\, \mathsf{fromMachine}(C, \mathsf{box}_T(\mathsf{lift}\, N), S') = (D, \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$ and $\mathsf{fromMachine}(C, (\vec{\ell}, D, \vec{\ell'}), S') = (C, E[(\vec{\ell}, D, \vec{\ell'})])^{\vec{k}}.X$, for the same $E, \vec{k}, X$, and by inductive hypothesis we know that $(D, \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X$ is reachable. Because we have $(D, \vec{\ell'})^{\vec{\ell}}.(C, E[\mathsf{box}_T(\mathsf{lift}\, N)])^{\vec{k}}.X \rightharpoonup (C, E[(\vec{\ell}, D, \vec{\ell'})])^{\vec{k}}.X$ by the *step-out* rule, we conclude that $(C, E[(\vec{\ell}, D, \vec{\ell'})])^{\vec{k}}.X$ is reachable.

- Case of *let-split*. In this case we have $(D, \mathsf{let}\, \langle x, y\rangle = N\, \mathsf{in}\, P, S') \Rightarrow (D, N, \mathit{Let}(x, y, P).S')$. By inductive hypothesis we know $\mathsf{fromMachine}(D, \mathsf{let}\, \langle x, y\rangle = N\, \mathsf{in}\, P, S')$ is reachable, so we conclude that $\mathsf{fromMachine}(D, N, \mathit{Let}(x, y, P).S') = \mathsf{fromMachine}(D, \mathsf{let}\, \langle x, y\rangle = N\, \mathsf{in}\, P, S')$ is reachable.

- Case of *let-join*. In this case $(D, \langle V, W\rangle, \mathit{Let}(x, y, N).S') \Rightarrow (D, N[V/x][W/y], S')$. By propositions 4.2.1 and 4.2.2 we know that $\mathsf{fromMachine}(D, \langle V, W\rangle, \mathit{Let}(x, y, N).S') = \mathsf{fromMachine}(D, \mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N, S') = (D, E[\mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N])^{\vec{\ell}}.X$ and $\mathsf{fromMachine}(D, N[V/x][W/y], S') = (D, E[N[V/x][W/y]])^{\vec{\ell}}.X$, for the same $E, \vec{\ell}, X$, and by inductive hypothesis we know that $(D, E[\mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N])^{\vec{\ell}}.X$ is reachable. Because $(D, \mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N) \to (D, N[V/x][W/y])$ by the *let* rule, by Theorem 2.5.3 we get $(D, E[\mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N]) \to (D, E[N[V/x][W/y]])$. Also, because $\mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N$ is a redex, we know by Corollary 2.5.6.1 that $E[\mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N] \not\equiv E'[\mathsf{box}_T(\mathsf{lift}\,P)]$ for any E', so we get $(D, E[\mathsf{let}\,\langle x, y\rangle = \langle V, W\rangle\,\mathsf{in}\,N])^{\vec{\ell}}.X \rightharpoonup (D, E[N[V/x][W/y]])^{\vec{\ell}}.X$ by the *head* rule and conclude that $(D, E[N[V/x][W/y]])^{\vec{\ell}}.X$ is reachable.

- Case of *force-open*. In this case we have $(D, \mathsf{force}\,N, S') \Rightarrow (D, N, \mathit{Force}.S')$. By inductive hypothesis we know that $\mathsf{fromMachine}(D, \mathsf{force}\,N, S')$ is reachable, so we immediately conclude that $\mathsf{fromMachine}(D, N, \mathit{Force}.S') = \mathsf{fromMachine}(D, \mathsf{force}\,N, S')$ is reachable.

- Case of *force-close*. In this case we have $(D, \mathsf{lift}\,N, \mathit{Force}.S') \Rightarrow (D, N, S')$. By propositions 4.2.1 and 4.2.2 we know that $\mathsf{fromMachine}(D, \mathsf{lift}\,N, \mathit{Force}.S') = \mathsf{fromMachine}(D, \mathsf{force}(\mathsf{lift}\,N), S') = (D, E[\mathsf{force}(\mathsf{lift}\,N)])^{\vec{\ell}}.X, \mathsf{fromMachine}(D, N, S') = (D, E[N])^{\vec{\ell}}.X$ for the same $E, \vec{\ell}, X$, and by inductive hypothesis we know that $(D, E[\mathsf{force}(\mathsf{lift}\,N)])^{\vec{\ell}}.X$ is reachable. Because $(D, \mathsf{force}(\mathsf{lift}\,N)) \to (D, N)$ by the *force* rule, by Theorem 2.5.3 we get $(D, E[\mathsf{force}(\mathsf{lift}\,N)]) \to (D, E[N])$. Also, because $\mathsf{force}(\mathsf{lift}\,N)$ is a redex, we know by Corollary 2.5.6.1 that $E[\mathsf{force}(\mathsf{lift}\,N)] \not\equiv E'[\mathsf{box}_T(\mathsf{lift}\,P)]$ for any E', so we get $(D, E[\mathsf{force}(\mathsf{lift}\,N)])^{\vec{\ell}}.X \rightharpoonup (D, E[N])^{\vec{\ell}}.X$ by the *head* rule and conclude that $(D, E[N])^{\vec{\ell}}.X$ is reachable.

□

We can now prove that fromMachine preserves convergence between reachable machine configurations and reachable stacked configurations.

Proposition 4.2.8. *Suppose* (C, M, S) *is a reachable machine configuration.* $(C, M, S)\downarrow$ *if and only if* $\mathsf{fromMachine}(C, M, S)\downarrow$.

Proof. We start by proving that if $(C, M, S)\downarrow$, then $\mathsf{fromMachine}(C, M, S)\downarrow$. We proceed by induction on $(C, M, S)\downarrow$:

- Case $M \equiv V$ and $S = \epsilon$. In this case $\mathsf{fromMachine}(C, V, \epsilon) = (C, V)^{\emptyset}.\epsilon$ and the claim is trivially true.

- Case $(C, M, S) \Rightarrow (D, N, S')$ and $(D, N, S') \downarrow$. In this case we know by inductive hypothesis that $\mathsf{fromMachine}(D, N, S') \downarrow$. By Lemma 4.2.4 we get that $\mathsf{fromMachine}(C, M, S) \rightharpoonup^* \mathsf{fromMachine}(D, N, S')$ and by the definition of converging stacked configuration we conclude $\mathsf{fromMachine}(C, M, S) \downarrow$.

We now need to prove that if $\mathsf{fromMachine}(C, M, S) \downarrow$, then $(C, M, S) \downarrow$. We proceed by induction on $\mathsf{fromMachine}(C, M, S) \downarrow$:

- Case $\mathsf{fromMachine}(C, M, S) = (C, V)^{\emptyset}.\epsilon$. We know that $\mathsf{fromMachine}(C, M, S) = (C, E[M])^{\vec{\ell}}.X$ by Proposition 4.2.1. For $E[M] \equiv V$ to be true we must have $E \equiv [\cdot]$ and $M \equiv V$. We must also have $\vec{\ell} = \emptyset$ and $X = \epsilon$. The only way to have $E \equiv [\cdot], \vec{\ell} = \emptyset$ and $X = \epsilon$ is to have $S = \epsilon$. If $S = H.S'$ were non-empty, we would either have $\vec{\ell} \neq \emptyset$ and $X \neq \epsilon$ (in case of an H of type Sub) or $E \neq [\cdot]$ (for any other H), which would contradict the hypothesis. Therefore we have $M \equiv V$ and $S = \epsilon$ and we conclude $(C, V, \epsilon) \downarrow$.

- Case $\mathsf{fromMachine}(C, M, S) \rightharpoonup (D, N)^{\vec{k}}.X'$ and $(D, N)^{\vec{k}}.X' \downarrow$. Let (C', M', S') be the normal form of (C, M, S) with respect to $\Rightarrow_b$. We distinguish two cases:
  - If (C', M', S') is also normal with respect to $\Rightarrow$, then either $(C, M, S)\bot$ or $(C, M, S) \downarrow$. Because by Lemma 4.2.6 $(C, M, S)\bot$ would imply $\mathsf{fromMachine}(C, M, S)\bot$, contradicting Proposition 3.1.3, we conclude $(C, M, S) \downarrow$
  - If (C', M', S') is not normal with respect to $\Rightarrow$, then we have $(C', M', S') \Rightarrow_r (D', N', S'')$ and by Lemma 4.2.4 we know that $\mathsf{fromMachine}(C', M', S') \rightharpoonup \mathsf{fromMachine}(D', N', S'')$. At the same time, by the definition of $\Rightarrow_b$ we know that $\mathsf{fromMachine}(C', M', S') = \mathsf{fromMachine}(C, M, S)$. Because $\rightharpoonup$ is deterministic, this necessarily entails $\mathsf{fromMachine}(D', N', S'') = (D, N)^{\vec{k}}.X'$ and consequently $\mathsf{fromMachine}(D', N', S'') \downarrow$. By inductive hypothesis we get $(D', N', S'') \downarrow$ and conclude $(C, M, S) \downarrow$ by the definition of converging machine configuration.

□

4.2.4 Summary

In this section we established a relationship between machine configurations and stacked configurations via the fromMachine function, which gradually unwinds the stack of a machine configuration to obtain an equivalent stacked configuration. We showed that for every machine configuration (C, M, S), the stacked configuration $\mathsf{fromMachine}(C, M, S)$ is of the form $(C, E[M])^{\vec{\ell}}.X$ for some $E, \vec{\ell}, X$ that depend exclusively on S. As for the actual operational semantics, we showed that a reduction step in the machine semantics either leaves the corresponding stacked configuration unchanged, or can be simulated by a single reduction step in the stacked semantics. Lastly, we also proved that fromMachine

preserves convergence between machine and stacked configurations and that whenever a machine configuration goes into deadlock, then so does the corresponding stacked configuration.

4.3 Equivalence Between Small-step and Machine Semantics

Finally, in this last section we cover the relationship between the small-step semantics and the machine semantics. In particular, as we anticipated, we prove that they are essentially equivalent. Concretely, this means that converging computations in the small-step semantics translate to converging computations in the machine semantics (which converge to the same values), that computations that go into deadlock in the small-step semantics translate to computations that go into deadlock in the machine semantics, and – last but not least – that diverging computations in the small-step semantics translate to diverging computations in the machine semantics, and vice-versa. In order to prove these results, let us formalize the relationship between small-step and machine configurations through a simple load function, defined as such:

$$\mathsf{load}(C, M) = (C, M, \epsilon).$$

This function is very similar to $\mathsf{fromSmallStep}$. Like $\mathsf{fromSmallStep}$ it is trivially invertible, and as such is establishes a bijection between small-step configurations and the set $\mathcal{I}_{ma}$ of initial machine configurations. In addition, load has the following property:

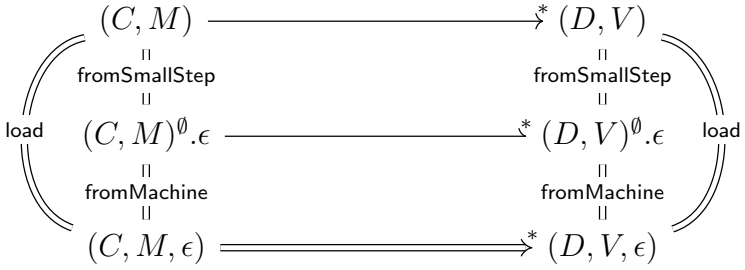
$$\mathsf{fromMachine} \circ \mathsf{load} = \mathsf{fromSmallStep},$$

which will be essential in the coming proofs. The property is actually trivial to prove, as for every small-step configuration (C, M) we have:

$$\mathsf{fromMachine}(\mathsf{load}(C, M)) = \mathsf{fromMachine}(C, M, \epsilon) = (C, M)^{\emptyset}.\epsilon = \mathsf{fromSmallStep}(C, M).$$

4.3.1 Convergence

As we mentioned at the beginning of this chapter, we prove that corresponding small-step and machine configurations evaluate to the same circuit and value in the respective semantics by proving that the two computations are simulated by the same computation in the stacked semantics. To this effect, lemmata 4.1.1 and 4.2.4 are going to play a paramount role. Specifically, we want the diagrams that we introduced in sections 4.1.2 and 4.2.2 to compose as follows:



For simplicity, we prove the two directions of the equivalence separately and then put them together to prove our goal.

Lemma 4.3.1. *Suppose* (C, M) *and* (D, V) *are small-step configurations. If* $(C, M) \to^* (D, V)$*, then* $\mathsf{load}(C, M) \Rightarrow^* \mathsf{load}(D, V)$*.*

Proof. The case in which $(C, M) = (D, V)$ is trivially true, so let us consider the case in which $(C, M) \to^+ (D, V)$. First of all, by Lemma 4.1.1 we get that $(C, M)^\emptyset.\epsilon \rightharpoonup^+ (D, V)^\emptyset.\epsilon$. In parallel, by propositions 4.1.3 and 4.2.8 we have that $\mathsf{load}(C, M) = (C, M, \epsilon)$ converges, since $\mathsf{fromSmallStep}(C, M) = (C, M)^\emptyset.\epsilon = \mathsf{fromMachine}(C, M, \epsilon)$ and (C, M, ϵ) is trivially reachable. This entails that there exists a normal form (D', V', ϵ) such that $(C, M, \epsilon) \Rightarrow^* (D', V', \epsilon)$. By Lemma 4.2.4 this implies that $\mathsf{fromMachine}(C, M, \epsilon) \rightharpoonup^* \mathsf{fromMachine}(D', V', \epsilon)$, or $(C, M)^\emptyset.\epsilon \rightharpoonup^* (D', V')^\emptyset.\epsilon$. We now have $(C, M)^\emptyset.\epsilon \rightharpoonup^+ (D, V)^\emptyset.\epsilon$ and $(C, M)^\emptyset.\epsilon \rightharpoonup^* (D', V')^\emptyset.\epsilon$. Because $\rightharpoonup$ is deterministic and because $(D, V)^\emptyset.\epsilon$ and $(D', V')^\emptyset.\epsilon$ are both normal forms, we get that $(D, V)^\emptyset.\epsilon = (D', V')^\emptyset.\epsilon$, that is, $D = D'$ and $V = V'$, and we conclude $\mathsf{load}(C, M) \Rightarrow^* (D, V, \epsilon)$. □

Lemma 4.3.2. *Suppose* (C, M) *and* (D, V) *are two small-step configurations. If* $\mathsf{load}(C, M) \Rightarrow^* \mathsf{load}(D, V)$*, then* $(C, M) \to^* (D, V)$*.*

Proof. The case in which $\mathsf{load}(C, M) = \mathsf{load}(D, V)$ is trivially true, so let us consider the case in which $\mathsf{load}(C, M) \Rightarrow^+ \mathsf{load}(D, V)$. By Proposition 4.2.8 we know that $\mathsf{fromMachine}(\mathsf{load}(C, M)) = \mathsf{fromSmallStep}(C, M)$ converges, and by Proposition 4.1.3 we know that (C, M) converges too. That is, $(C, M) \to^* (D', V')$. By Lemma 4.3.1 this entails $\mathsf{load}(C, M) \Rightarrow^* \mathsf{load}(D', V')$. Since $\mathsf{load}(D, V)$ and $\mathsf{load}(D', V')$ are both normal forms, $(D, V) \neq (D', V')$ would contradict the determinism of $\Rightarrow$. As a result, we know that $(D, V) = (D', V')$ and conclude $(C, M) \to^* (D, V)$. □

Theorem 4.3.3 (Equivalence in Convergence)**.** *Suppose* (C, M) *and* (D, V) *are small-step configurations. We have that* $(C, M) \to^* (D, V)$ *if and only if* $\mathsf{load}(C, M) \Rightarrow^* \mathsf{load}(D, V)$*.*

Proof. The claim follows immediately from lemmata 4.3.1 and 4.3.2. □

Corollary 4.3.3.1. *Suppose* (C, M) *is a small-step configuration. We have that* $(C, M)\downarrow$ *if and only if* $\mathsf{load}(C, M)\downarrow$*.*

Proof. The claim follows immediately from Theorem 4.3.3 and the definition for converging small-step and machine configurations. Alternatively, it follows from propositions 4.1.3 and 4.2.8. □

4.3.2 Deadlock and Divergence

The equivalence between the small-step and machine semantics is stronger than that between the big-step and the small-step semantics, as we now show that whenever a small-step computation goes into deadlock or diverges, then the corresponding machine computation goes into deadlock or diverges, respectively. For simplicity, we prove both results first in one direction and then in the other.

Lemma 4.3.4. *Suppose* (C, M) *is a small-step configuration. If* $(C, M)\bot$*, then* $\mathsf{load}(C, M)\bot$*.*

Proof. First of all, by Lemma 4.1.4 we know that $\mathsf{fromSmallStep}(C, M)\bot$. Now, suppose $\mathsf{load}(C, M) \not\downarrow$. By Proposition 3.2.5 we know that either $\mathsf{load}(C, M) \downarrow$ or $\mathsf{load}(C, M) \uparrow$. Because $\mathsf{load}(C, M) = (C, M, \epsilon)$ is trivially reachable, by Proposition 4.2.8 we know that if $\mathsf{load}(C, M) \downarrow$ then $\mathsf{fromMachine}(\mathsf{load}(C, M)) = \mathsf{fromSmallStep}(C, M) \downarrow$, which contradicts Proposition 3.1.3, so $\mathsf{load}(C, M) \not\downarrow$. On the other hand, $\mathsf{load}(C, M) \uparrow$ would entail an infinite computation starting from $\mathsf{load}(C, M)$. By lemmata 4.2.4 and 4.2.3 this would entail an infinite computation starting from $\mathsf{fromMachine}(\mathsf{load}(C, M)) = \mathsf{fromSmallStep}(C, M)$ too, which would contradict $\mathsf{fromSmallStep}(C, M)\bot$, since $\mathsf{fromSmallStep}(C, M)\bot$ implies that the same computation is finite ($\rightharpoonup$ is deterministic). Because $\mathsf{load}(C, M) \not\downarrow$ ultimately leads to a contradiction, we conclude $\mathsf{load}(C, M)\bot$. □

Lemma 4.3.5. *Suppose* (C, M) *is a small-step configuration. If* $(C, M) \uparrow$*, then* $\mathsf{load}(C, M) \uparrow$*.*

Proof. First of all, because the computation starting from (C, M) is infinite, by Lemma 4.1.1 we know that the computation starting from $\mathsf{fromSmallStep}(C, M)$ is also infinite ($\rightharpoonup$ is deterministic). Now, suppose $\mathsf{load}(C, M) \not\uparrow$. By Proposition 3.2.5 we know that either $\mathsf{load}(C, M) \downarrow$ or $\mathsf{load}(C, M)\bot$. Because $\mathsf{load}(C, M)$ is trivially reachable, by Proposition 4.2.8 we know that if $\mathsf{load}(C, M) \downarrow$ then $\mathsf{fromMachine}(\mathsf{load}(C, M)) = \mathsf{fromSmallStep}(C, M) \downarrow$. This contradicts the fact that the computation starting from $\mathsf{fromSmallStep}(C, M)$ is infinite, since $\mathsf{fromSmallStep}(C, M) \downarrow$ implies that the same computation is finite, so $\mathsf{load}(C, M) \not\downarrow$. On the other hand, if $\mathsf{load}(C, M)\bot$, by Lemma 4.2.6 we know that $\mathsf{fromMachine}(\mathsf{load}(C, M)) = \mathsf{fromSmallStep}(C, M)\bot$. However, this contradicts the fact that the computation starting from $\mathsf{fromSmallStep}(C, M)$ is infinite, since $\mathsf{fromSmallStep}(C, M)\bot$ implies that the same computation is finite, so $\mathsf{load}(C, M) \not\downarrow$. Because $\mathsf{load}(C, M) \not\uparrow$ ultimately leads to a contradiction, we conclude $\mathsf{load}(C, M) \uparrow$. □

Now that we have these two results, we can use them to prove the other direction in a much more straightforward way.

Lemma 4.3.6. *Suppose* (C, M) *is a small-step configuration. If* $\mathsf{load}(C, M)\bot$*, then* $(C, M)\bot$*.*

Proof. Suppose $(C, M) \not\downarrow$. By Proposition 2.5.9 we know that either $(C, M) \downarrow$ or $(C, M) \uparrow$. However, by Corollary 4.3.3.1 $(C, M) \downarrow$ entails $\mathsf{load}(C, M) \downarrow$, while by Lemma 4.3.5 $(C, M) \uparrow$ entails $\mathsf{load}(C, M) \uparrow$. Because both these conclusions contradict Proposition 3.2.3, we conclude that $(C, M)\bot$. □

Lemma 4.3.7. *Suppose* (C, M) *is a small-step configuration. If* $\mathsf{load}(C, M) \uparrow$*, then* $(C, M) \uparrow$*.*

Proof. Suppose $(C, M) \not\uparrow$. By Proposition 2.5.9 we know that either $(C, M) \downarrow$ or $(C, M)\bot$. However, by Corollary 4.3.3.1 $(C, M) \downarrow$ entails $\mathsf{load}(C, M) \downarrow$, while by Lemma 4.3.4 $(C, M)\bot$ entails $\mathsf{load}(C, M)\bot$. Because both these conclusions contradict Proposition 3.2.3, we conclude that $(C, M) \uparrow$. □

Eventually, the four lemmata can be summarized in the following two theorems, which, together with Theorem 4.3.3, complete the picture of the equivalence between the small-step and machine semantics. Figure 4.1 illustrates the same result.

Theorem 4.3.8 (Equivalence in Deadlock)**.** *Suppose* (C, M) *is a small-step configuration. We have that* $(C, M)\bot$ *if and only if* $\mathsf{load}(C, M)\bot$*.*

Proof. The claim follows immediately from lemmata 4.3.4 and 4.3.6. □

Theorem 4.3.9 (Equivalence in Divergence)**.** *Suppose* (C, M) *is a small-step configuration. We have that* $(C, M) \uparrow$ *if and only if* $\mathsf{load}(C, M) \uparrow$*.*

Proof. The claim follows immediately from lemmata 4.3.5 and 4.3.7. □

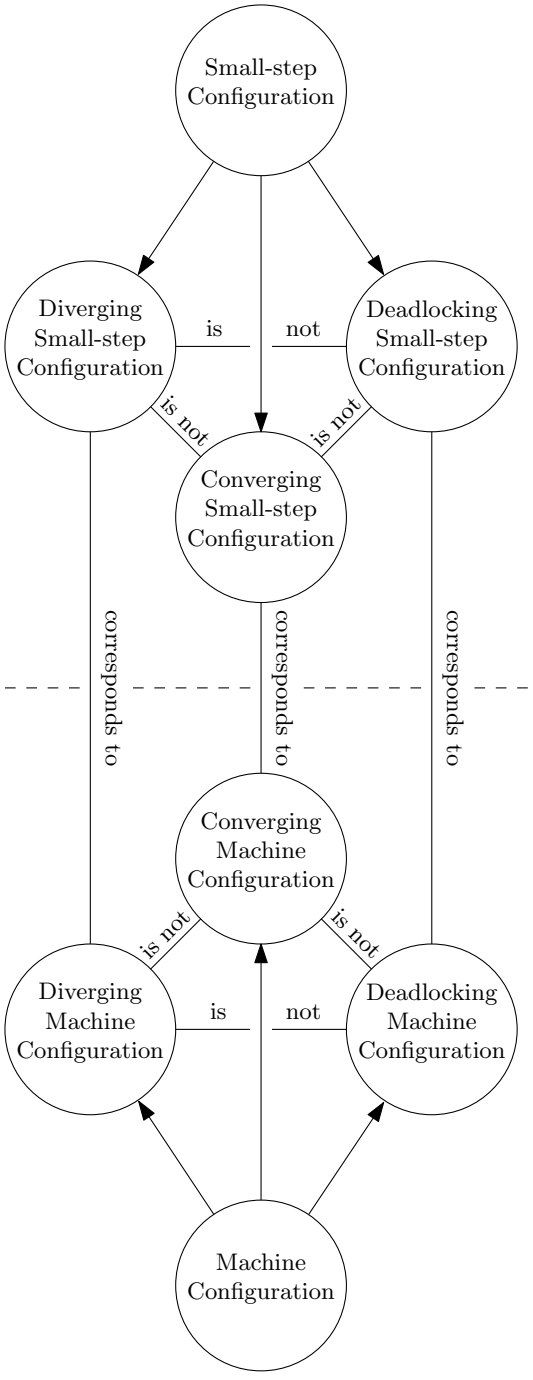


Figure 4.1: The final relationship between the small-step semantics modelled after Proto-Quipper-M's big-step semantics and the machine semantics proposed in this thesis.

Conclusions and Future Work

In the first part of this thesis we reviewed Quipper, a functional language for the description of quantum circuits, and explained its limitations as an embedded language in Haskell. We consequently presented Rios and Selinger's work on Proto-Quipper-M, a member of the Proto-Quipper research language family, which aims at formalizing relevant fragments of Quipper in a type-safe way. We introduced the intuition behind Proto-Quipper-M, its categorical model for quantum circuits and its syntax. We also discussed its linear type system, which allows the enforcement of the no-cloning property of quantum states at compile time, effectively overcoming one of Quipper's greatest weaknesses. Last, but not least, we presented Proto-Quipper-M's big-step operational semantics.

By rewriting the big-step rules of Proto-Quipper-M into small-step rules, we obtained an equivalent semantics which is small-step save for the case of circuit boxing. We showed that this semantics behaves well with respect to Proto-Quipper-M's type system by proving subject reduction and progress results. In the second part of the thesis, we defined a *stacked semantics* for Proto-Quipper-M, which overcomes the aforementioned problems with circuit boxing by organizing all of the sub-reductions introduced by a boxing operation in an explicit stack. We used this semantics as an intermediate step in the definition of a *machine semantics* for Proto-Quipper-M, which takes this approach even further. Inspired by abstract machines such as the CEK machine, this semantics models every phase of the evaluation of a program as a continuation on a stack. Lastly, we concluded this thesis by proving that the proposed machine semantics is equivalent to the initial small-step semantics and – as a consequence – to the original big-step semantics given by Rios and Selinger.

Related Work

It is worth mentioning that the author's curricular internship revolved around the reembedding in Haskell of the original Proto-Quipper-M specification by Rios and Selinger. Although a proposal for the addition of linear types to Haskell [2] has been approved and a portion of it has recently been released as part of GHC 9.0.1, at the time of the internship

this linear extension was still unavailable. In fact, the very goal of the internship was to investigate alternative methods to enforce Quipper's linearity constraints in Haskell in the absence of a full-fledged linear type system. To this effect, we relied on the work of Paykin and Zdancewic [11], which take full advantage of some of the most advanced GHC extensions related to type-level computing to offer a framework for the embedding of domain-specific linear languages in Haskell. We report two main findings. First and foremost, we found that Paykin and Zdancewic's work was not sufficient to enforce Proto-Quipper-M's linearity constraints to their full extent. Specifically, we found it impossible to model the domain and codomain of circuits at compile time. As such, whenever we encountered a term of the form $(\vec{\ell}, D, \vec{\ell'})$, we had to resort to run time checks to enforce that $D \in \mathbf{M}_{\mathcal{L}}(Q, Q')$ for the same Q and Q' used to type $\vec{\ell}$ and $\vec{\ell'}$. Secondly, despite the fact that Paykin and Zdancewic's framework is designed specifically to allow the definition of the semantics of the embedded languages in a big-step manner, we found it exceedingly hard to implement the boxing operator box_T, due to the same reasons that made the *box* rule of the small-step semantics difficult to handle in this thesis. In retrospect, this validates our desire to provide a true small-step semantics for Proto-Quipper-M.

Future Work

The point of arrival of this thesis is a minimal abstract machine which accurately models the operational semantics of the Proto-Quipper-M language, and therefore formalizes a fundamental fragment of the behavior of Quipper itself. From here, we can expect most of the future work to be focused on one of two directions.

The first direction is that of expanding the current machine specification to progressively model a larger and larger portion of Quipper. First and foremost, we have that a considerable number of language features included in the original Proto-Quipper-M specifications by Rios and Selinger have been omitted in this thesis for the sake of feasibility. These are not domain specific features, and include things such as sequencing operators, sum types, pattern matching, naturals, lists, and so on. Although not essential for circuit building, these are the features that usually make a programming language practical and, as a consequence, useful. Therefore, it would be appropriate, although unchallenging (and probably tedious) to extend the current machine specification with these programming constructs and to show that this extension does not compromise the results that we have given. More interestingly, the proposed machine semantics could be used as a starting point to model some of Quipper's most advanced features, which have no counterpart in Proto-Quipper-M. A prime example of such a feature is *dynamic lifting*, which refers to the ability to measure the intermediate state of qubits in the midst of the execution of a circuit and to use the resulting classical information to build the

remaining portion of the circuit on the fly.

The second direction is one that we briefly mentioned in the introduction of this thesis, and it is not completely orthogonal to the first one. The research direction in question is the one that focuses on the static analysis of interesting properties of Quipper programs. In this case, our machine semantics could be used as a reference model to define concepts such as the time needed to construct a circuit, or the number of qubits required by it. The estimation of the latter quantity, in particular, would be extremely valuable in a time where quantum resources are still scarce, and it is not trivial to compute, especially if dynamic lifting is involved.

Lastly, as a side note, the machine itself could be made more concrete than it currently is. As we mentioned in Section 3.2.2, when designing our machine we chose to keep relying on an abstract substitution function for reasons of simplicity. Taking further inspiration from the CEK machine, an explicit substitution algorithm could be implemented by endowing our own machine with an environment component and rules to look up variables inside an environment. One major obstacle in this approach is that our environments would contain linear resources, and thus would have to be handled differently from how they are treated in the CEK machine. Note that this concretization operation is not essential, but it would be particularly beneficial to any research focusing – for example – on the static estimation of the circuit generation time of Quipper programs.

Bibliography

[1] Andrea Asperti and Giuseppe Longo. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist*. M.I.T. Press, Jan. 1991. ISBN: 9780262011259.

[2] Jean-Philippe Bernardy et al. "Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158093. URL: https://doi.org/10.1145/3158093.

[3] Andrea Colledan. *On the Hidden Subgroup Problem as a Pivot in Quantum Complexity Theory*. Bachelor's thesis. URL: http://amslaurea.unibo.it/16112/.

[4] Matthias Felleisen and Daniel P. Friedman. "Control Operators, the SECD-Machine, and the Lambda-Calculus". In: *3rd Working Conference on the Formal Description of Programming Concepts*. Aug. 1986.

[5] Peng Fu et al. *A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper*. 2020. arXiv: 2005.08396 [cs.PL].

[6] Alexander S. Green et al. "An Introduction to Quantum Programming in Quipper". In: *Lecture Notes in Computer Science* (2013), pp. 110–124. ISSN: 1611-3349. DOI: 10.1007/978-3-642-38986-3_10. URL: http://dx.doi.org/10.1007/978-3-642-38986-3_10.

[7] Alexander S. Green et al. "Quipper". In: *ACM SIGPLAN Notices* 48.6 (June 2013), pp. 333–342. ISSN: 1558-1160. DOI: 10.1145/2499370.2462177. URL: http://dx.doi.org/10.1145/2499370.2462177.

[8] Jean-Louis Krivine. "A Call-by-Name Lambda-Calculus Machine". In: *Higher Order Symbol. Comput.* 20.3 (Sept. 2007), pp. 199–207. ISSN: 1388-3690. DOI: 10.1007/s10990-007-9018-9. URL: https://doi.org/10.1007/s10990-007-9018-9.

[9] P. J. Landin. "The Mechanical Evaluation of Expressions". In: *The Computer Journal* 6.4 (Jan. 1964), pp. 308–320. ISSN: 0010-4620. DOI: 10.1093/comjnl/6.4.308. eprint: https://academic.oup.com/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf. URL: https://doi.org/10.1093/comjnl/6.4.308.

[10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011. ISBN: 1107002176.

[11] Jennifer Paykin and Steve Zdancewic. "The Linearity Monad". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 117–132. DOI: 10.1145/3122955.3122965. URL: https://doi.org/10.1145/3122955.3122965.

[12] Emily Riehl. *Category Theory in Context*. Dover Publications, Nov. 2016. ISBN: 9780486809038.

[13] Francisco Rios and Peter Selinger. "A Categorical Model for a Quantum Circuit Description Language (Extended Abstract)". In: *Electronic Proceedings in Theoretical Computer Science* 266 (Feb. 2018), pp. 164–178. ISSN: 2075-2180. DOI: 10.4204/eptcs.266.11. URL: http://dx.doi.org/10.4204/EPTCS.266.11.

[14] Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation*. 2017. arXiv: 1510.02198 [quant-ph].

[15] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum Computing for Computer Scientists*. 1st ed. USA: Cambridge University Press, 2008. ISBN: 0521879965.