Type Systems for Guaranteeing Resource Bounds of Component Software

Hoang Anh Truong

Dissertation for the degree Philosophiae Doctor (PhD)
University of Bergen, Norway

April 2006

Abstract

Since the early days of the development of programming languages, people have been developing various methods to reduce the runtime errors of software programs. These methods range from static analysis, testing to runtime monitoring. The first method is usually preferable if possible since it allows us to detect certain errors of programs before their execution. Type systems are a lightweight method of static analysis and they will be the main interest of the thesis.

Type systems are a syntactic method for proving that certain kinds of programming errors cannot occur. Traditionally, they are used to guarantee that the parameters passed to a function are always of the right data type. Recently, many complex type systems are incorporated into mainstream programming languages and people are actively exploring their other capabilities. We study type systems for preventing errors caused by a lack of resources.

Recently, component software facilitates building large, evolving software systems from a collection of standard reusable components, which can be developed independently. However, component software has a latent problem with resources. Due to the independent development and reuse of components, during the execution of a component program, several instances of the same component may be alive at the same time. For some components, creating a number of coexistent instances more than a certain number is not allowed or exceptions will occur. Examples are components that use special resources such as serial communication ports or database connections. Preventing this class of exceptions by static type systems is the subject study of this thesis.

This thesis develops static type systems for abstract component languages so that well-typed programs will not use more resources than a certain bound. Putting it differently, through the types we know statically the maximum resources that a program needs so that if a system has enough such resources, executing the program on the system will never cause out-of-resource exceptions.

Contents

Acknowledgments **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Background Information 2

1.2.1 Type Systems 2

1.2.2 Component Software 3

1.3 Related Work 4

1.4 Thesis Outline 4

2 Basic Language with Instantiation, Sequencing, Choice, and Scope **7**

2.1 The Basic Language 7

2.1.1 Syntax 7

2.1.2 Operational Semantics 8

2.2 Type System 11

2.3 Properties 14

2.3.1 Type Soundness 14

2.3.2 Typing Properties 16

2.3.3 Soundness Proofs 22

2.4 Type Inference 26

3 Explicit Deallocation **29**

3.1 Language 29

3.1.1 Syntax 29

3.1.2 Operational Semantics 30

3.2 Type System 31

3.3 Properties 33
3.3.1 Type Soundness 33
3.3.2 Typing Properties 34
3.3.3 Soundness Proofs 42
3.4 Type Inference 46

4 Explicit Deallocation and Parallel Composition **47**
4.1 Language 47
4.1.1 Syntax 47
4.1.2 Operational Semantics 48
4.2 Type System 50
4.3 Properties 52
4.3.1 Type Soundness 52
4.3.2 Typing Properties 54
4.3.3 Soundness Proofs 56

5 Reuse Instantiation **65**
5.1 Language 65
5.1.1 Syntax 65
5.1.2 Operational Semantics 66
5.2 Type System 67
5.3 Properties 70
5.3.1 Type Soundness 70
5.3.2 Typing Properties 70
5.3.3 Soundness Proofs 74

6 Reuse Instantiation and Parallel Composition **79**
6.1 Language 79
6.1.1 Syntax 79
6.1.2 Operational Semantics 80
6.2 Type System 83
6.3 Properties 85
6.3.1 Type Soundness 85
6.3.2 Typing Properties 88

6.4 Soundness Proofs 92

7 Conclusions and Future Research **111**

7.1 Conclusions 111

7.2 Future Research 111

Bibliography **113**

List of Tables

2.1 Syntax of the basic language 8

2.2 Transition rules of the basic language, **S** may be empty 9

2.3 Typing rules of the basic language 12

3.1 Syntax of the language with `del` 29

3.2 Transition rules of the language with `del` 30

3.3 Typing rules of the language with `del` 32

4.1 Syntax of the language with `del` and parallel composition 47

4.2 Basic reduction rules of the language with `del` and parallel composition . . 49

4.3 Typing rules of the language with `del` and parallel composition 51

5.1 Syntax of the language with reuse 65

5.2 Transition rules of the language with reuse 66

5.3 Typing rules of the language with reuse 68

6.1 Syntax of the language with `reu` and parallel composition 79

6.2 Reduction rules of the language with `reu` and parallel composition 81

6.3 Structural congruence: basic axioms 81

6.4 Typing rules of the language with `reu` and parallel composition 84

Acknowledgments

I am deeply indebted to my advisor, Professor Marc Bezem, for his invaluable suggestions, guidance, and encouragement during my study at the department. He has introduced the basics of type theory to me, and has guided me a lot during my three years of research. I also especially thank him for suggesting this interesting thesis topic to me. I shall always look up to him for his kindness in general.

I would like to acknowledge the other members of my thesis committee Prof. Dr. Magne Haveraaen, Msc. Jan Heering and Dr. Erik Barendsen.

I am very grateful to other MoSIS project members from whom I have learned a lot through courses and seminars, especially Uwe Wolter, Magne Haveraaen, Michal Walicki. My life and work in Norway have been made much easier thanks to my many colleagues in the department, especially the administrative and technical staff. I greatly appreciate their help and assistance. I would like to express my thanks to Dag Hovland for his discussion about my thesis and his help with the press release in Norwegian, and Wojciech Szajnkenig for the nice time we shared at our office.

My thanks go to my parents for their patience and my friends for their support when I was away from home and for improving the English writing of this thesis. Thank Mrs. Kim-Oanh Le for her care and help during my time in Bergen.

I also wish to thank the Norwegian Research Council for funding this research.

Chapter 1

Introduction

This thesis develops static type systems for abstract component languages to classify a class of their programs that will not cause runtime errors due to a lack of resources. In particular, the type systems ensure that a well-typed program will not use more resources than a certain bound. In other words, through the types we know statically the maximum resources that a program needs during its execution so that if a system has enough such resources, the program will never cause out-of-resource errors at runtime.

1.1 Motivation

Any software program needs certain resources during its execution. These resources range from abundant ones such as memory and file handles, to scarce ones such as communication ports and network bandwidth. Usually, when a program requests a resource, the system (operating system, virtual machine) will try to allocate the resource and if successful, it gives the resource to the program. When the program finishes using the resource, it returns the resource to the system so that other processes can use the resource. If a program requests a resource that is not available—because the system does not have it, or some other processes are occupying the resource—the program may wait for the resource, or raise an out-of-resource exception, or just terminate abnormally. These behaviours are unexpected and are not acceptable for critical systems as they may lead to problems such as loss of data, deadlocks, system overload, or even system crashes.

Component software [37, 38] is a software that has been assembled from standardized, reusable components, which may have been developed by third parties. These third-party components, in turn, may also be composed from other components, and so on. Due to the independent development and reuse of components, several third parties may use the same resource and this is a source of problems.

When component software is executed, instances of its components are created [51]. Because it can happen that several of these components use the same subcomponent, many instances of the subcomponent may be created. If the subcomponent allocates a scarce resource, the system may easily be unable to fulfill all the requests and out-of-resource errors will likely to occur. Moreover, some components are supposed to have at most a certain number of instances at a time due to their own characteristics or application requirements [20, 26]. For instance, a serial ID number generator component is supposed to have a unique instance, or the number of concurrent database connections is usually limited for efficiency. For these components, creating a greater number of instances than the allowed number will cause problems such as resource conflict or inefficiency or even unexpected results. In short, component software has a high risk of causing out-of-resource exceptions and resource conflict problems.

There are several ways to prevent this class of errors, ranging from testing, runtime monitoring to static analysis. Among these methods, static analysis is usually preferable

if possible. Static analysis is a family of techniques for automatically deriving information about the behaviour of computer software from the source text of the software. This is in contrast to dynamic techniques, which analyze directly the run-time behaviour, by testing, benchmarking, profiling and so on. One of the static analysis techniques is static type systems which are the subject of this research.

This work studies static type systems for predicting the class of out-of-resource errors for programs before they are executed, in two ways. First, given a component program, the type systems statically build a *certificate* which states the maximum resources that the program needs. Hence, before launching a program on a system, we can compare the resources of the system with the certificate and once it has enough such resources, we can ensure that executing the program will not cause errors due to a lack of resources. Second, instead of finding the maximum resources that a program needs, we can fix the resource bounds as a requirement and the type systems can verify whether a program respects the resource constraint or not.

This work touches upon two main areas: type systems [4, 9, 27, 31] and component software [35, 38]. The former is, in this work, more prominent than the latter, since the component languages have been abstracted. The next section will briefly introduce the two areas focusing on the aspects relevant for our interests.

1.2 Background Information

1.2.1 Type Systems

Type systems are a lightweight formal method for proving the absence of certain execution errors of a program [9]. The proving process can be performed statically—at compile time, or dynamically—at run time. Proving at compile time, if possible, is more preferable to the other since it allows early detection of errors, even though there is no consensus on this preference. This work focuses on compile time type systems, also known as static type systems.

A type system can be regarded as calculating a kind of static approximation to the run-time behaviour of the terms in a program.

Generally, a programming language is defined by a syntax and a (dynamic) semantics. The syntax allows us to write an infinite number of programs and the semantics defines the behaviour of the programs at runtime. Type systems (sometimes referred to as the static semantics) are like filters which select, at compile time, only programs whose dynamic behaviour satisfies certain properties. Note that type systems are usually *conservative* (cf. [31]) in the sense that there may be programs that satisfy the given properties but the type systems do not accept them. However, the accepted ones always behave correctly with respect to the given characteristics. The latter assertion is often the most important property of a type system and is known as *type soundness* or *type safety* property.

Traditionally, type systems are used to guarantee that the parameters passed to a function are always of the right data type [2]. Recently, type systems have been studied in various areas of computer science such as language design and implementation, software engineering, databases, security and distributed systems. It has shown its capabilities to detect more errors with programming constructs such as race detection [18] and safe dereference [45]. Nowadays, complex type systems are being used in practical programming languages (cf. Generic Java [1, 3, 8]). And more complex type systems are being explored in research, such as effects [29], dependent types [30, 48, 50], module systems [16], security types [21], types for low-level languages [11] and types for XML.

The type systems in this work are not very complex, but they show a capability of type theory in detecting resource bound errors which emerge more commonly in the way large applications today are built—from software components.

1.2.2 Component Software

Learning from the success of the way personal computers and many other machines are built up from a collection of standard components, the idea of component software is that a larger software application can be built from standard, reusable blocks. The idea appeared at the early development of the software industry when McIlroy introduced the idea of software components produced by a software components industry [25]. Recently, the industry has matured and component technologies such as CORBA Component Model [22], Enterprise Javabeans [28] and .NET [41] enable building large software systems from off-the-shelf compiled components. Many aspects of the recent development of this area are discussed by Szyperski [38]. This section only introduces the features of component software that are relevant to this work.

As mentioned in Section 1.1, a component program, which in fact is a larger component, is built from other 'smaller' components. These smaller components in turn may also be composed from other components, and so on. These dependencies usually have an end where there are *primitive components* that do not use any other components. From an abstract point of view, the subcomponents of a larger component are assembled by various composition operators: sequencing, choice, scope, and parallel composition.

Upon execution of the component program, instances of its components are created. The process of creating an instance of a component x does not only mean the allocation of memory space for x's code and data structures, the creation of instances of x's subcomponents (and so on), but possibly also the binding of other system and hardware resources. On one hand, since the resources are usually limited, components are required to have at most a certain number of simultaneously active instances. For example, a component that uses the unique communication port of a machine should have only one instance [26]. On the other hand, because of the specific characteristics of some components, these components are required to have only one or a limited number of instances. For example, the number of concurrent database connections is often limited for efficiency. Therefore, the number of instances of a database connection component should also be limited. Other examples come from the singleton pattern and its extensions (multitons), which have been widely discussed in the literature [19]. These patterns limit the number of objects of a certain class dynamically, at runtime. In conclusion, some software components can have at most a certain number of active instances at a time.

As one of the important properties of component software is that components can be developed independently by different parties, it can happen that a good component x, which supposedly can have at most n instances, is used in several third-party components. When these third-party components are composed in a component program, many instances of x may be created during the execution of the program. Of course, if the number of instances of x is more than the allowed number n, then out-of-resource errors will occur. Preventing this class of errors is the main goal of this work.

We will study several component languages and develop type systems which can prove that for well-typed programs, these out-of-resource errors will not happen. The languages are abstract and have only the most relevant features. These features can be divided into two groups. The first group contains primitives that directly manipulate resources: allocation, deallocation, reuse. The second group is four main composition operators: sequencing, choice, scope, and parallel composition.

In spite of the high abstraction level, these features are present in many programming languages. Sequencing is a basic property of imperative programming languages. Choice models branch operators like 'if..then..else' or 'switch..case' occurring in many programming languages. The scope mechanism allows us to allocate resources to a fresh region and later discards the whole region, claiming back all the resources allocated there. Parallel composition allows many threads to run concurrently, a feature of most modern programming languages.

1.3 Related Work

Work on type systems for checking resource bounds can be divided into two main categories. The first category tries to estimate the resource bound and the second checks the constraints on the resource bound given by programmers. Our type systems cover both approaches, but maintain consistency between the two.

In very recent work [46], Chin *et. al.* presented a type system that can capture memory bounds of object-oriented programs. The type system is based on dependent type [49]. The language supports method calls and object–oriented features, so the language is less abstract than our languages. However, their language does not have the parallel construct and the primitive (`reu` in Chapters 5 and 6) for sharing resources. The combination of the parallel composition and reuse instantiation (sharing resources) in Chapter 6 makes resource bounds computation non-trivial. The earlier work [10] by Chin *et. al.* also provided a framework for inferring abstract size of programs as exact as possible (since they used Presburger formulae for size information). The language is functional and does not have the explicit deallocation primitive nor the reuse primitive of ours. Moreover, our computation of 'sizes' is exact.

Crary and Weirich [12, 13] presented decidable type systems for low level languages which are capable of specifying and certifying that their programs will terminate within a given amount of time, but the type system does not infer any bounds on resource consumption; it can only certify the bounds given by programmers. In contrast, our type systems focus on high level languages and they can infer the sharp upper bound of resources.

Hofmann [23, 24] showed that linear type systems can ensure that programs do not increase the size of their input so that exponential growth of immediate results can be avoided, even with the presence of iterated recursion. His languages are functional while ours are imperative. However, the safety of deallocation that our type systems in Chapters 3 and 4 guarantee is inspired from the linear type systems.

Some works on time analysis of programs such as [7, 15, 34] are not closely related to this work. We do not estimate this class of resources.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 discusses a basic language with only one instantiation primitive: `new`, and three composition operators: sequencing, choice and scope. In Chapter 2, basic concepts of syntax, operational semantics, and type systems are explained in more detail than in the subsequent chapters.

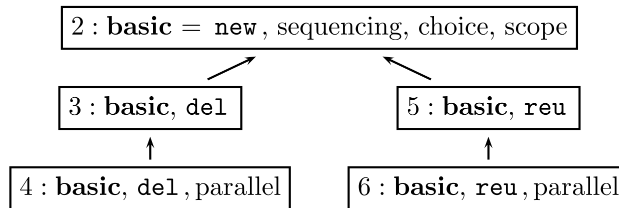


Figure 1.1: Dependencies between the main chapters

The subsequent four chapters have a similar structure to Chapter 2 and they can be grouped into two parts. In the first part, Chapter 3 extends the basic language of Chapter 2 with an explicit deallocation primitive `del`; Then Chapter 4 extends Chapter 3 with a parallel composition. Symmetric to the first part, the second part has two chapters but they consider another instantiation primitive `reu` instead of the deallocation `del`. That is, Chapter 5 extends the basic language of Chapter 2 with a reuse instantiation primitive

`reu`; Then Chapter 6 extends Chapter 5 with the parallel composition. Finally, Chapter 7 discusses some future directions and concludes.

Chapters 3 through 6 are extensions of Chapter 2, so they mainly focus on explaining new features and updating the definitions and properties of Chapter 2. Nevertheless, some notions are repeated to ease independent reading. The following figure highlights the language features of each chapter and shows the dependencies among the main chapters.

The thesis is based on publications of mine [42] and in collaboration with Marc Bezem [5, 6, 43]. Chapter 2 is a generalization of [6]. Chapter 4 is an improvement of [43]. The results of Chapter 5 and Chapter 6 are from [5] and [42], respectively. Chapter 3, a simplified version of Chapter 4, eases the comprehension of Chapter 4.

Chapter 2

Basic Language with Instantiation, Sequencing, Choice, and Scope

As the first step, we study an abstract component language with basic features: instantiation, sequencing, choice and scope. These features will be present in the subsequent chapters, 3 though 6. So in this chapter, we will explain in more detail various concepts and the chosen level of abstraction. We will also consistently use the notations of this chapter throughout the thesis. Therefore, the later chapters will be less detailed on the concepts and notations.

The structure of this chapter is as follows. Section 2.1 introduces the basic language with its syntax and operational semantics. Section 2.2 develops a type system which can tell us the upper bounds of resources that a well-typed program requests. Section 2.3 states and proves the soundness and some other important properties of the type system. Finally, Section 2.4 outlines a decidable type inference algorithm. The next four chapters will extend the language of this chapter with more features but their structure is similar to here.

2.1 The Basic Language

A programming language possesses two fundamental features: syntax and semantics. Syntax refers to the appearance of the well-formed programs of the language, and semantics refers to the meanings of these programs. This section presents the syntax of the basic language by a grammar and its small-step operational semantics. We also give a sample program and use it as a running example in the subsequent sections.

2.1.1 Syntax

Table 2.1 defines the syntax of a core, abstract component language. In the definition, we use extended Backus-Naur Form with the following meta-symbols: '::=' for describing grammars, infix '|' for choice, and overlining for Kleene closure (zero or more iterations) [36].

Let $a, \ldots, z$ range over *component names* (also called *variables*) and $A, \ldots, E$ range over expressions. We collect all the component names in a set $\mathbb{C}$.

The main ingredients of the component language are component declarations and component expressions. We have a primitive `new` for creating an instance of a component, and three primitives for composition: sequential composition, denoted by juxtaposition, choice, denoted by $+$, and scope, denoted by $\{\ldots\}$. Together with the empty expression ϵ

Table 2.1: Syntax of the basic language

$\mathbb{C}$	$=$	$\{a, \ldots, z\}$	Set of component names
Prog	$::=$	$\overline{Decls}; E$	Program
Decls	$::=$	$x \prec E$	Declarations, $x \in \mathbb{C}$
E	$::=$		Expression
		ϵ	Empty
	$\mid$	$\texttt{new}\, x$	Instantiation
	$\mid$	$E\ E$	Sequencing
	$\mid$	$(E + E)$	Choice
	$\mid$	$\{E\}$	Scope

these generate the so-called *component expressions*. A *declaration* $x \prec E$ (read 'x deploys E') states how the component x depends on subcomponents as expressed in the component expression E. We call E the *body* of the declaration of x, or the body of x for short. If x uses no subcomponents, then E is ϵ and we call x a *primitive component*. A *component program* consists of a list of declarations followed by a *main expression*, which will be the startup expression when the program is executed, see Section 2.1.2.

Although the language is abstract, its strength is considerable. Choice allows us to model both conditionals and non-determinism (due to, for example, user input). It can also be used when a component has several compatible versions and the system can choose one of them at runtime. Scope is a mechanism to deallocate instances but it can also be used to model method calls. Sequential composition is associative.

In the following sample program, d and e are primitive components. Component a uses an instance of d in a scope, then it creates an instance of e. Component b starts by creating an instance of e, then it creates either an instance of a or two instances of d.

$$
\begin{aligned}
& d \prec \epsilon \quad e \prec \epsilon \\
& a \prec \{\texttt{new}\, d\}\, \texttt{new}\, e \\
& b \prec \texttt{new}\, e(\texttt{new}\, a + \texttt{new}\, d\, \texttt{new}\, d); \\
& \texttt{new}\, b
\end{aligned}
$$

An abstract machine for executing these programs is described next.

2.1.2 Operational Semantics

Informally, an expression E can be viewed as a sequence of commands of the forms $\texttt{new}\, x$, $(A + B)$ and $\{A\}$ in imperative programming languages and the execution is sequential from left to right. In the operational semantics, E is paired with a multiset of component names, called the *local store* of E. A *multiset* (also called *bag*) is like a set but with multiple occurrences of elements. The first two commands operate locally, that is, within the pair. When executing a command of the form $\texttt{new}\, x$, a new instance of x is created in the local store and the execution continues with the body A of x if A is not ϵ. If A is ϵ, the execution proceeds to the next command after $\texttt{new}\, x$. Executing $(A + B)$ means to choose either A or B to execute with the same local store.

When the current command is a scope expression $\{A\}$, the commands after the expression, say B, are suspended and the commands inside the scope, A, are executed with a new empty store. That is, we create pair $([], A)$ and execute commands in A with a fresh local store. When the execution of the new pair $([], A)$ terminates in the form (M, ϵ), the pair is *discarded* and the execution resumes to the expression B and its local store.

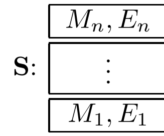


Figure 2.1: Illustration of a stack

The formal operational semantics is defined by a small-step (or one-step) transition system of *configurations* [33]. A configuration is a stack $\mathbf{S}$ of pairs of a local store and an expression, (M, E), where M is a multiset over $\mathbb{C}$, and E is an expression defined in Table 2.1. Figure 2.1 illustrates a configuration. A configuration is *terminal* (also called *final* or *halting*) if it has the form (M, ϵ). We denote stacks by the following syntax:

$$\mathbf{S} ::= (M_1, E_1) \circ \ldots \circ (M_n, E_n)$$

This stack $\mathbf{S}$ has n elements where (M_1, E_1) is the pair at the bottom of the stack, (M_n, E_n) is at the top of the stack, and '$\circ$' is the stack separator. We have designed the system so that under execution our stacks always have at least one element, $n \geq 1$, as we will see in the transition relation below. Before that we recapitulate the notion of multisets.

A *multiset* is like a set but with multiple occurrences of elements. Multisets are denoted by $[\ldots]$, where sets are denoted, as usual, by $\{\ldots\}$. An empty multiset is denoted by $[]$, while an empty set is denoted by $\emptyset$. Let M, N be two multisets, we summarize some operations on multisets. $M(x)$ is the *multiplicity* of element x. If $M(x) = 0$ then x is not an element of M, notation $x \notin M$. Domain of M, notation $\mathsf{dom}(M)$, is the set of elements in M: $\mathsf{dom}(M) = \{x \mid M(x) > 0\}$. The operation $\cup$ is union of two multisets: $(M \cup N)(x) = max(M(x), N(x))$; operation $+$ or $\uplus$ is additive union of two multisets: $(M + N)(x) = M(x) + N(x)$; operation $-$ is subtraction of two multisets: $(M - N)(x) = max(M(x) - N(x), 0)$. We write $M + x$ for $M + [x]$ and when $x \in M$ we write $M - x$ for $M - [x]$. The inclusion relation $\subseteq$: $M \subseteq N$ if $M(x) \leq N(x)$ for all $x \in M$.

Table 2.2: Transition rules of the basic language, $\mathbf{S}$ may be empty

(osNew) $\quad x \prec A \in \mathit{Decls}$
$\mathbf{S} \circ (M, \mathtt{new}\, xE) \longrightarrow \mathbf{S} \circ (M + x, AE)$

(osChoice) $\quad i \in \{1, 2\}$
$\mathbf{S} \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S} \circ (M, A_i E)$

(osPush)
$\mathbf{S} \circ (M, \{A\}E) \longrightarrow \mathbf{S} \circ (M, E) \circ ([], A)$

(osPop)
$\mathbf{S} \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S} \circ (M, E)$

Table 2.2 defines a small-step transition relation. Each transition rule has two lines. The first line contains a rule name followed by a list of conditions and annotations. The second line has the form $\mathbf{S} \longrightarrow \mathbf{S}'$, which states that we can move from state $\mathbf{S}$ to state $\mathbf{S}'$ if all the conditions in the first line hold. As usual, the multi-step transition relation $\longrightarrow^*$ is the reflective and transitive closure of $\longrightarrow$.

By the rules osNew and osChoice we only rewrite the pair at the top of the stack. In the rule osNew, we first create a new instance of component x in the local store. Then if x is a primitive component we continue to execute the remaining expression E; otherwise, we continue to execute A before executing the remaining expression E. By $x \prec A \in \mathit{Decls}$ we denote $x \prec A$ is a declaration in Decls. Here we also assume that there is only one

declaration of x in *Decls*. The type system in the next section will check this condition so that any well-typed program has only one declaration for each component. In the rule osChoice, we simply select one of the two expressions to execute.

The next two rules are for scope and they change the number of elements of the stack. The rule osPush pushes an element on the top of the stack. The rule osPop pops the stack when the stack has at least two elements. Here we have designed the rule so that stacks are never empty. Hence we can avoid the runtime error of popping an empty stack.

The example at the end of Section 2.1.1 is used to illustrate the operational semantics. There are two possible runs of the program. The first possibility is:

$$\begin{array}{rl}
(\mathsf{Startup}) & ([], \texttt{new}\, b) \\
(\mathsf{osNew}) \longrightarrow & ([b], \texttt{new}\, e(\, \texttt{new}\, a + \texttt{new}\, d\, \texttt{new}\, d)) \\
(\mathsf{osNew}) \longrightarrow & ([b,e], (\, \texttt{new}\, a + \texttt{new}\, d\, \texttt{new}\, d)) \\
(\mathsf{osChoice}) \longrightarrow & ([b,e], \texttt{new}\, a) \\
(\mathsf{osNew}) \longrightarrow & ([b,e,a], \{\, \texttt{new}\, d\}\, \texttt{new}\, e) \\
(\mathsf{osPush}) \longrightarrow & ([b,e,a], \texttt{new}\, e) \circ ([], \texttt{new}\, d) \\
(\mathsf{osNew}) \longrightarrow & ([b,e,a], \texttt{new}\, e) \circ ([d], \epsilon) \\
(\mathsf{osPop}) \longrightarrow & ([b,e,a], \texttt{new}\, e) \\
(\mathsf{osNew}) \longrightarrow & ([b,e,a,e], \epsilon) \qquad \text{(terminal)}
\end{array}$$

The other possible run is:

$$\begin{array}{rl}
(\mathsf{Startup}) & ([], \texttt{new}\, b) \\
(\mathsf{osNew}) \longrightarrow & ([b], \texttt{new}\, e(\, \texttt{new}\, a + \texttt{new}\, d\, \texttt{new}\, d)) \\
(\mathsf{osNew}) \longrightarrow & ([b,e], (\, \texttt{new}\, a + \texttt{new}\, d\, \texttt{new}\, d)) \\
(\mathsf{osChoice}) \longrightarrow & ([b,e], \texttt{new}\, d\, \texttt{new}\, d) \\
(\mathsf{osNew}) \longrightarrow & ([b,e,d], \texttt{new}\, d) \\
(\mathsf{osNew}) \longrightarrow & ([b,e,d,d], \epsilon) \qquad \text{(terminal)}
\end{array}$$

Up until now we have fully described the component language, but we have not seen where the resources mentioned in the opening of this chapter and in Chapter 1 are, and how to know the maximum resources that a program needs. In fact, we can abstract the resources in several ways.

Given a program, a natural way to infer its maximum consumption of a certain resource is to annotate the usage of that resource for each component of the program. That is, for a given resource, we have a function that maps every component name to the maximal amount of the resource that the component directly uses. Then we can run the program and infer the total resource consumption of each execution state by taking the sum of resources occupied by all existing instances. For example, in the above illustrative program, suppose that components a and e each uses 1KB of memory, components b and d each uses 2KB. Then at the second terminal state $([b,e,d,d], \epsilon)$, the program occupies 7KB of memory.

Another way to find the maximum of a particular resource that a component program uses is declaring the specific resource as a primitive component. Other components will then instantiate the component in their declarations if they use the resource. The maximum of the resource that the program uses is the maximum number of the instances in all possible states of the program. For instance, if we regard d as a database connection component, then in the above example, component b (also the program) needs a maximum of two database connections (at the second terminal state).

In these methods, we need to examine all possible states of the program to know the maximum resources that the program needs. In general, these methods are not applicable to detect these maxima since testing all possible runs is usually impossible due to a possible exponential number of such runs or circular dependencies of components. The type system in the next section can tell us the maximum resource consumption for a class of programs and it inspires a polynomial algorithm to find such an upper bound.

2.2 Type System

As mentioned in the Introduction, we are interested in static type systems. Static type systems are used to prove the absence of execution (runtime) errors of a program without the need to run the program. By contrast, checking for errors during the execution of a program is a kind of dynamic type system. Each type system usually checks for only a certain class of errors. Before describing our static type system, we should clarify which errors we are aiming at.

One of the common runtime errors during the execution of a program is the *stuck state*. That is a non-terminal state in which none of the transition rules can be applied, usually because the conditions of the rules do not hold. Examining the transition relation in Table 2.2, we can see that an execution is stuck when it tries to instantiate a component x but there is no declaration of x in the declarations of the program. Checking this class of errors is easy, by looking up the list of declarations. Moreover, because at the moment we do not allow recursion and mutual recursion in component declarations, the type system should also be able to rule out cyclic dependencies. Detecting these cyclic dependencies is also not difficult—for example, by using dependency graphs. We are aiming at another goal.

The main goal of the type system is to find the upper bounds of resources that a program may request. Knowing these bounds allows us to prevent runtime errors when the program is to be launched on a system whose resources are below the bounds. In addition, we want our types to be compositional, which means that types can be computed from types of subexpressions. The main benefit of this compositionality is that type information can be attached to a component as a part of its specification. The specification allows hiding the details of internal composition of components but still enables components to be subcomponents for further composition.

Note that since we have abstracted the specific resources in the instances, the upper bounds become the maximum numbers of *simultaneously active instances* during the execution of the program. By the term *simultaneously active instances* we mean all the instances in all the stores of a configuration. We will simplify the term 'number of simultaneously active instances' as 'number of instances' when it causes no ambiguity.

We start by defining the usual notions of a type system: types, typing environments and judgments. First, *types*, ranged over by $U, V, \ldots, Z$, are pairs of two multisets.

Definition 2.2.1 (Types). *Types of component expressions are pairs*

$$X = \langle X^i, X^o \rangle$$

where X^i, X^o are finite multisets over $\mathbb{C}$.

Let us explain informally why multisets, which multisets and why two. The aim is to have an upper bound of the number of simultaneously active instances of any component during the execution of an expression. Multisets are the right data structure to collect and count such instances. We use X^i for this bound. In addition, we want compositionality of typing, that is, we want the types to be computable from types of subexpressions. Since subexpressions may be scoped, it is necessary to have an upper bound of the number of instances that are still active *after* the execution of an expression. We use X^o for this bound. Pairs $\langle X^i, X^o \rangle$ suffice for the purpose of the language of this chapter. Later,

we will need some additional multisets when we enrich the language with more features. However, these two multisets will always be there and they will play the same roles.

Next, a *basis* or *environment* is a list of declarations: $x_1 \prec E_1, \ldots, x_n \prec E_n$. An empty basis is denoted by $\varnothing$. Let Γ, Δ range over bases. The *domain of a basis* $\Gamma = x_1 \prec E_1, \ldots, x_n \prec E_n$, notation $\mathsf{dom}(\Gamma)$, is the set $\{x_1, \ldots, x_n\}$. Formally, the function dom is defined recursively as follows.

$$\mathsf{dom}(\varnothing) = \emptyset$$
$$\mathsf{dom}(\Gamma, x \prec E) = \{x\} \cup \mathsf{dom}(\Gamma)$$

Finally, a *typing judgment* (or just *judgment*) is a tuple of the form:

$$\Gamma \vdash E : X$$

and it asserts that expression E has type X in the environment Γ. A typing judgment can be regarded as *valid* or *invalid*. Valid ones are identified by the following definitions.

Definition 2.2.2 (Valid typing judgments). *Valid typing judgments $\Gamma \vdash A : X$ are derived by applying the typing rules in Table 2.3 in the usual inductive way.*

By the term *usual inductive way* we mean a valid judgment is one that can be obtained as the root of a tree of judgments, where each judgment is obtained from the ones immediately above it by some typing rule in Table 2.3. Such a tree of judgments is called a *typing derivation*.

In Table 2.3, each typing rule has two parts divided by a line. The above part starts with a name of the rule followed by a number of *premise judgment*s and *side condition*s. Below the line is a single *conclusion judgment*, which must hold when all the premises and the side conditions hold. A rule can have no premises (Axiom) and it is usually used for startup.

Table 2.3: Typing rules of the basic language

(Axiom)
$$\frac{}{\varnothing \vdash \epsilon : \langle [], [] \rangle}$$

(WeakenB)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec B \vdash A : X}$$

(New)
$$\frac{\Gamma \vdash A : X \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{new}\, x : \langle X^i + x, X^o + x \rangle}$$

(Seq)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad A, B \neq \epsilon}{\Gamma \vdash AB : \langle X^i \cup (X^o + Y^i), X^o + Y^o \rangle}$$

(Choice)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y}{\Gamma \vdash (A + B) : \langle X^i \cup Y^i, X^o \cup Y^o \rangle}$$

(Scope)
$$\frac{\Gamma \vdash A : X}{\Gamma \vdash \{A\} : \langle X^i, [] \rangle}$$

These typing rules deserve some further explanation. The most critical rule is Seq because sequencing two expressions can lead to an increase in instances of the composed expression. Let us start with the first multiset of the type of AB. During the execution of A, the maximum number of instances of component x is $X^i(x)$. After expression A is executed, there are at most $X^o(x)$ instances of component x. Hence, during the execution of B, the maximum number of instances of x is $X^o(x) + Y^i(x)$. So during the execution of AB the maximum number of instances of x is the maximum of $X^i(x)$ and $X^o(x) + Y^i(x)$. For the second multiset, referring to the operational semantics and the meaning of the second multiset, it is easy to see that the total number of instances of x after the execution of AB is $X^o(x) + Y^o(x)$. Therefore the type of AB is $\langle X^i \cup (X^o + Y^i), X^o + Y^o \rangle$.

Other typing rules are straightforward. The rule Axiom is used for startup. The rule WeakenB allows us to extend the type environment so that the rules Seq and Choice may be applied. In the rule New, we need to add x to both multisets since, after creating an instance x, we go on executing A in the same local store. The side condition $x \notin \mathsf{dom}(\Gamma)$ prevents ambiguity and circularity. We discard all instances surviving expression A when leaving the scope, so the rule Scope empties the second multiset.

Now we can link the program in Section 2.1 with the type system by defining the notion of *well-typed program.* Basically, a program is well-typed if there exists a typing derivation for the main expression of the program with a typing environment formed by a list of the program declarations, maybe in a difference order of the program declarations. Note that, unlike several systems whose order of elements in a basis is unimportant, the order of declarations in our bases is significant which will be emphasized in Lemma 2.3.12 later. In Type Inference Section (2.4), we will show a polynomial algorithm which can automatically decide whether a program is well-typed or not.

Definition 2.2.3 (Well-typed programs). *Program $Prog = Decls; E$ is well-typed if there exist a reordering Γ of declarations in $Decls$ and a type X such that $\Gamma \vdash E : X$.*

We end this section by giving some typing derivations for expressions in the example program of Section 2.1.1. Note that, we omitted some side conditions as they can be checked easily and the rule Axiom is also simplified.

$$\textsc{WeakenB}\dfrac{\textsc{Scope}\dfrac{\textsc{New}\dfrac{\varnothing \vdash \epsilon : \langle [], [] \rangle}{d \prec \epsilon \vdash \mathtt{new}\, d : \langle [d], [d] \rangle}}{d \prec \epsilon \vdash \{\, \mathtt{new}\, d\} : \langle [d], [] \rangle} \quad \textsc{WeakenB}\dfrac{\varnothing \vdash \epsilon : \langle [], [] \rangle}{d \prec \epsilon \vdash \epsilon : \langle [], [] \rangle}}{d \prec \epsilon, e \prec \epsilon \vdash \{\, \mathtt{new}\, d\} : \langle [d], [] \rangle} \tag{2.1}$$

$$\textsc{New}\dfrac{\textsc{Seq}\dfrac{(2.1) \quad \textsc{New}\dfrac{\textsc{WeakenB}\dfrac{\varnothing \vdash \epsilon : \langle [], [] \rangle \quad \varnothing \vdash \epsilon : \langle [], [] \rangle}{d \prec \epsilon \vdash \epsilon : \langle [], [] \rangle}}{d \prec \epsilon, e \prec \epsilon \vdash \mathtt{new}\, e : \langle [e], [e] \rangle}}{d \prec \epsilon, e \prec \epsilon \vdash \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e : \langle [d, e], [e] \rangle}}{d \prec \epsilon, e \prec \epsilon, a \prec \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e \vdash \mathtt{new}\, a : \langle [a, d, e], [a, e] \rangle} \tag{2.2}$$

$$\textsc{WeakenB}\dfrac{\textsc{Seq}\dfrac{d \prec \epsilon \vdash \mathtt{new}\, d : \langle [d], [d] \rangle \quad d \prec \epsilon \vdash \mathtt{new}\, d : \langle [d], [d] \rangle}{d \prec \epsilon \vdash \mathtt{new}\, d\, \mathtt{new}\, d : \langle [d, d], [d, d] \rangle} \quad \dfrac{\cdots}{d \prec \epsilon \vdash \epsilon : \langle [], [] \rangle}}{d \prec \epsilon, e \prec \epsilon \vdash \mathtt{new}\, d\, \mathtt{new}\, d : \langle [d, d], [d, d] \rangle} \tag{2.3}$$

Let $\Gamma_1 = d \prec \epsilon, e \prec \epsilon$. We can weaken the conclusion of (2.3) with $\Gamma_1 \vdash \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e : \langle [d, e], [e] \rangle$ in (2.2) as follows.

$$\textsc{WeakenB}\dfrac{\Gamma_1 \vdash \mathtt{new}\, d\, \mathtt{new}\, d : \langle [d, d], [d, d] \rangle \quad \Gamma_1 \vdash \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e : \langle [d, e], [e] \rangle}{\Gamma_1, a \prec \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e \vdash \mathtt{new}\, d\, \mathtt{new}\, d : \langle [d, d], [d, d] \rangle} \tag{2.4}$$

Let $\Gamma_2 = \Gamma_1, a \prec \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e$. We can apply the rule Choice with (2.2) and (2.4) as premises and get.

$$\textsc{Choice}\dfrac{\Gamma_2 \vdash \mathtt{new}\, a : \langle [a, d, e], [a, e] \rangle \quad \Gamma_2 \vdash \mathtt{new}\, d\, \mathtt{new}\, d : \langle [d, d], [d, d] \rangle}{\Gamma_2 \vdash (\, \mathtt{new}\, a + \mathtt{new}\, d\, \mathtt{new}\, d) : \langle [a, d, d, e], [a, d, d, e] \rangle} \tag{2.5}$$

Similarly, we can derive $\Gamma_2 \vdash \mathtt{new}\, e : \langle [e], [e] \rangle$ and then the type for $\mathtt{new}\, b$ is derived as follows:

$$\textsc{New}\dfrac{\textsc{Seq}\dfrac{\Gamma_2 \vdash \mathtt{new}\, e : \langle [e], [e] \rangle \qquad (2.5)}{\Gamma_2 \vdash \mathtt{new}\, e(\, \mathtt{new}\, a + \mathtt{new}\, d\, \mathtt{new}\, d) : \langle [a, d, d, e, e], [a, d, d, e, e] \rangle}}{\Gamma_2, b \prec \mathtt{new}\, e(\, \mathtt{new}\, a + \mathtt{new}\, d\, \mathtt{new}\, d) \vdash \mathtt{new}\, b : \langle [a, b, d, d, e, e], [a, b, d, d, e, e] \rangle}$$

Now we take a closer look at how to calculate specific resource consumption from the abstraction of resources by component names, as mentioned at the end of Section 2.1.2. Recall that we have showed two ways to infer specific resources. Following the second way, using component names as specific resources, the type system tells us directly the resource bounds of a well-typed program. More specifically, the resource bounds of a well-typed component program are expressed in the first multiset of the type of the program's main expression.

Regarding the first method that uses resource mapping functions, we need to refine the type system by resource effects, in the way of type and effect systems [29, 39]. Basically, starting from the leaves of a derivation tree, we replace component names by the corresponding quantified resource usage, and additive unions and unions on multisets by sums and maxima on the quantities, respectively. If we apply the refinement to a derivation tree for the main expression of a program, then at the root of the tree, we get the maximum resources that the program needs.

For example, assume that components a, e each uses 1KB and b, d each uses 2KB of memory. Then in the derivation tree for $\texttt{new}\, b$ we replace $\texttt{new}\, d : \langle d, d \rangle$ by $\texttt{new}\, d : \langle 2, 2 \rangle$ and $\texttt{new}\, e : \langle e, e \rangle$ by $\texttt{new}\, e : \langle 1, 1 \rangle$. Continuing down the tree, we get $\{\, \texttt{new}\, d\} \,\texttt{new}\, e : \langle 2, 1 \rangle$ and $\texttt{new}\, a : \langle 3, 2 \rangle$, and so on. The first multiset of the type of $\texttt{new}\, a$ does not mean that all of its elements can be simultaneously active, because in this case d is scoped. Therefore, we cannot directly calculate how much memory $\texttt{new}\, a$ needs from its abstract type: $\langle [a, d, e], [a, e] \rangle$. If we apply the resource mapping function to the abstract type and then take the sum, the result is $\langle 4, 2 \rangle$, which is not the correct amount of memory that $\texttt{new}\, a$ needs. So we have to calculate on a derivation tree of $\texttt{new}\, a$.

2.3 Properties

In this section, we study some important properties of the type system. One of the most important properties of a type system is type soundness (or type safety). So we will state this property in the beginning of this section. However, to prove this property we need some technical lemmas, so we will defer the soundness proof until the technical lemmas have been proved.

2.3.1 Type Soundness

As introduced in the beginning of Section 2.2, type errors occur when a program tries to instantiate a component x but there is no declaration of x. We will prove that this will not happen for well-typed programs. We will also prove an additional important property, which guarantees that the maximum number of instances at any state during the execution of a well-typed program will not exceed the bounds stated in the type of the main expression of the program.

The proof of type soundness is based on the standard approach of Wright and Felleisen [47]. We will prove two main lemmas: Preservation and Progress. The former states that well-typedness of configuration is preserved under any one-step transition. The latter guarantees that a well-typed configuration cannot get stuck, that is, move to a non-terminal state, from which it cannot move to another state. These properties together tell us that a well-typed program can never reach a stuck state during its execution. To state these lemmas, we need to define the notion of *well-typed configuration*s. We start with some notations for a stack $\mathbf{S}$.

We denote by $\mathsf{hi}(\mathbf{S})$ the number of elements of the stack $\mathbf{S}$, by $\mathbf{S}(k)$ the element at position k from the bottom of the stack, by $[\mathbf{S}(k)]$ the store M at position k, by $[\mathbf{S}]$ the additive union of all stores in the stack, and by $\mathbf{S}|_k$ the stack from the bottom of $\mathbf{S}$ up to

and including k. For instance, let $\mathbf{S} = (M_1, E_1) \circ \ldots \circ (M_n, E_n)$, then

$$\begin{aligned}
\mathsf{hi}(\mathbf{S}) &= n && \text{(the number of elements of } \mathbf{S}\text{)} \\
\mathbf{S}(k) &= (M_k, E_k) && \text{(the pair at position } k\text{)} \\
\mathbf{S}|_k &= (M_1, E_1) \circ \ldots \circ (M_k, E_k) && \text{(a sub-stack)} \\
[\mathbf{S}(k)] &= M_k && \text{(the store at position } k\text{)} \\
[\mathbf{S}] &= \biguplus_{i=1}^{n} [\mathbf{S}(k)] = \biguplus_{i=1}^{n} M_i && \text{(all active instances of } \mathbf{S}\text{)}
\end{aligned}$$

A configuration is well-typed with respect to a given basic if all the expressions in the configuration are well-typed under the basis.

Definition 2.3.1 (Well-typed configurations). *Configuration $\mathbf{S}$ is well-typed with respect to a basis Γ, notation $\Gamma \models \mathbf{S}$, if for all $1 \leq k \leq \mathsf{hi}(\mathbf{S})$ such that $\mathbf{S}(k) = (M, E)$, there exists X such that*

$$\Gamma \vdash E : X$$

The formal definition of terminal configurations and stuck states can be stated as follows.

Definition 2.3.2 (Terminal configurations). *A configuration $\mathbf{S}$ is terminal if it has the form (M, ϵ).*

Definition 2.3.3 (Stuck states). *A configuration $\mathbf{S}$ is stuck if no transition rule applies and $\mathbf{S}$ is not terminal.*

Having the definition of well-typed configurations, the two main lemmas Preservation and Progress mentioned at the beginning of the section are stated as follows.

Lemma 2.3.4 (Preservation). *If $\Gamma \models \mathbf{S}$ and $\mathbf{S} \longrightarrow \mathbf{S}'$, then $\Gamma \models \mathbf{S}'$.*

Lemma 2.3.5 (Progress). *If $\Gamma \models \mathbf{S}$, then either $\mathbf{S}$ is terminal or there exists a configuration $\mathbf{S}'$ such that $\mathbf{S} \longrightarrow \mathbf{S}'$.*

Next, we formulate an additional invariant, which allows us to infer the resource bounds of a well-typed program. The invariant is about the monotonicity of the maximum number of instances that a well-typed configuration can reach. The maximum is calculated by the function maxins as follows:

$$\mathsf{maxins}(\mathbf{S}) = \bigcup_{k=1}^{\mathsf{hi}(\mathbf{S})} ([\mathbf{S}|_k] + X_k^i)$$

where X_k is the type of the expression at position k. Such X_k exists by Definition 2.3.1. During transition, this maximum number of instances decreases. As will be shown in the proof, the inclusion is related to choice: fewer options means smaller maxima.

Lemma 2.3.6 (Invariant of maxins). *If $\Gamma \models \mathbf{S}$ and $\mathbf{S} \longrightarrow \mathbf{S}'$, then*

$$\mathsf{maxins}(\mathbf{S}) \supseteq \mathsf{maxins}(\mathbf{S}')$$

Now we can state the soundness together with the maximum numbers of instances that a well-typed program always respects.

Theorem 2.3.7 (Soundness). *Let $Prog = Decls; E$ be well-typed, that is, $\Gamma \vdash E : X$ for some reordering Γ of Decls and some type X. Then for any $\mathbf{S}$ such that $([], E) \longrightarrow^* \mathbf{S}$ we have that $\mathbf{S}$ is not stuck and $[\mathbf{S}] \subseteq X^i$.*

As we will see in the proof of this theorem, the bound M is in fact the first multiset of the type of E. Proving the above properties needs some technical lemmas of the type system. These lemmas state properties of the type system alone, independent of the operational semantics.

2.3.2 Typing Properties

This section lists and proves some important properties of the type system. One of the crucial properties is Generation Lemma 2.3.11, because it allows us to build a type inference algorithm, as we will see in Section 2.4. First, we fix some terminology on bases.

Definition 2.3.8 (Bases). *Let* $\Gamma = x_1 \prec A_1, \ldots, x_n \prec A_n$ *be a basis.*

- Γ *is called* legal *if* $\Gamma \vdash A : X$ *for some expression* A *and type* X*.*
- *A declaration* $x \prec A$ is in Γ*, notation* $x \prec A \in \Gamma$*, if* $x \equiv x_i$ *and* $A \equiv A_i$ *for some* i*.*
- Δ *is an* initial segment *of* Γ*, if* $\Delta = x_1 \prec A_1, \ldots, x_j \prec A_j$ *for some* $1 \leq j \leq n$*.*

We define the function $\mathsf{var}(E)$ which returns the set of variables occurring in the expression E as follows:

$$\begin{aligned}
&\mathsf{var}(\epsilon) = \emptyset,\\
&\mathsf{var}(\mathtt{new}\, x) = \{x\}, \quad \mathsf{var}(\{A\}) = \mathsf{var}(A),\\
&\mathsf{var}(AB) = \mathsf{var}((A + B)) = \mathsf{var}(A) \cup \mathsf{var}(B)
\end{aligned}$$

The following lemma collects a number of simple properties of a valid typing judgment. It states that if $\Gamma \vdash A : X$, then the elements of each multiset of X and variables of A are in the domain of Γ. It also shows a relation among multisets of X and any legal basis always has distinct declarations. In the sequel, we use X^* for any of X^i and X^o.

Lemma 2.3.9 (Valid typing judgment). *If* $\Gamma \vdash A : X$*, then*

1. $\mathsf{var}(A) \subseteq \mathsf{dom}(\Gamma)$*,* $\mathsf{dom}(X^*) \subseteq \mathsf{dom}(\Gamma)$*,*
2. *every variable in* $\mathsf{dom}(\Gamma)$ *is declared only once in* Γ*,*
3. $X^i \supseteq X^o$*.*

Proof. By simultaneous induction on typing derivations.

- Base case Axiom:

  $$\text{(Axiom)}\ \frac{}{\varnothing \vdash \epsilon : \langle [], [] \rangle}$$

  Then $\mathsf{var}(\epsilon) = \mathsf{dom}(X^*) = \mathsf{dom}(\varnothing) = \emptyset$ and all the clauses are trivial.

- Case WeakenB:

  $$\text{(WeakenB)}\ \frac{\Gamma' \vdash A : X \quad \Gamma' \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x \prec B \vdash A : X}$$

  Clause 1 holds by the induction hypothesis and $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma', x \prec B) \supset \mathsf{dom}(\Gamma')$. Clause 2 holds by the side condition and the induction hypothesis. Clause 3 holds by the induction hypothesis.

- Case New:

  $$\text{(New)}\ \frac{\Gamma' \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x \prec B \vdash \mathtt{new}\, x : \langle Y^i + x, Y^o + x \rangle}$$

  Clause 1 holds by the induction hypothesis and $x \in \mathsf{var}(\mathtt{new}\, x)$, $x \in X^*$, and $x \in \mathsf{dom}(\Gamma)$. Clause 2 holds by the side condition and the induction hypothesis. Clause 3 follows by the induction hypothesis.

- Case Seq:

$$\frac{\Gamma \vdash B\!:\!Y \quad \Gamma \vdash C\!:\!Z \quad B, C \neq \epsilon}{\Gamma \vdash BC\!:\!\langle Y^i \cup (Y^o + Z^i), Y^o + Z^o \rangle} \ (\textsc{Seq})$$

  Clause 1 holds by the induction hypothesis and $\mathsf{var}(BC) = \mathsf{var}(B) \cup \mathsf{var}(C)$. Clause 2 holds by the induction hypothesis. Clause 3 also follows easily by the induction hypothesis.

- Case Choice:

$$\frac{\Gamma \vdash B\!:\!Y \quad \Gamma \vdash C\!:\!Z}{\Gamma \vdash (B + C)\!:\!\langle Y^i \cup Z^i, Y^o \cup Z^o \rangle} \ (\textsc{Choice})$$

  All clauses follow easily by the induction hypothesis.

- Case Scope:

$$\frac{\Gamma \vdash B\!:\!Y}{\Gamma \vdash \{B\}\!:\!\langle Y^i, [] \rangle} \ (\textsc{Scope})$$

  Clauses 1 and 2 follow easily by the induction hypothesis. Clause 3 is trivial.

□

The following lemma shows the associativity of the sequential composition.

Lemma 2.3.10 (Associativity). *If $\Gamma \vdash A_i : X_i$, for $i \in \{1, 2, 3\}$, then the typing judgments for $(A_1A_2)A_3$ and $A_1(A_2A_3)$ are the same.*

Proof. If $A_i = \epsilon$ for some i, then the proof is trivial. So we assume that $A_i \neq \epsilon$ for all $i \in \{1, 2, 3\}$.

By the rule Seq, we have $\Gamma \vdash A_1A_2\!:\!Y$ with $Y = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o \rangle$:

$$\frac{\Gamma \vdash A_1\!:\!X_1 \quad \Gamma \vdash A_2\!:\!X_2 \quad A_1, A_2 \neq \epsilon}{\Gamma \vdash A_1A_2\!:\!\langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o \rangle} \ (\textsc{Seq})$$

Similarly, we have $\Gamma \vdash A_2A_3\!:\!Z$ with $Z = \langle X_2^i \cup (X_2^o + X_3^i), X_2^o + X_3^o \rangle$. Continue applying the rule Seq, we get $\Gamma \vdash (A_1A_2)A_3 : \langle Y^i \cup (Y^o + X_3^i), Y^o + X_3^o \rangle$ and $\Gamma \vdash A_1(A_2A_3) : \langle X_1^i \cup (X_1^o + Z^i), X_1^o + Z^o \rangle$. Then to prove that the two judgments are the same, we only need to prove that the two types are the same:

$$\begin{aligned} Y^i \cup (Y^o + X_3^i) &= X_1^i \cup (X_1^o + Z^i) \\ Y^o + X_3^o &= X_1^o + Z^o \end{aligned}$$

The first equation is straightforward:

$$\begin{aligned} & Y^i \cup (Y^o + X_3^i) \\ &= X_1^i \cup (X_1^o + X_2^i) \cup (Y^o + X_3^i) && (Y^i = X_1^i \cup (X_1^o + X_2^i)) \\ &= X_1^i \cup (X_1^o + X_2^i) \cup (X_1^o + X_2^o + X_3^i) && (Y^o = X_1^o + X_2^o) \\ &= X_1^i \cup (X_1^o + (X_2^i \cup (X_2^o + X_3^i))) \\ &= X_1^i \cup (X_1^o + Z^i) && (Z^i = X_2^i \cup (X_2^o + X_3^i)) \end{aligned}$$

The second equation is easy: $Y^o + X_3^o = X_1^o + X_2^o + X_3^o = X_1^o + Z^o$.

□

The following lemma is important as it allows us to find a syntax-directed derivation of the type of an expression and hence it allows us to calculate the types of sub-expressions

and is used in type inference (Section 2.4). This lemma is sometimes called the *inversion of the typing relation* [31]. Note that, in the second clause, the sequential decomposition in A and B may not be unique.

Lemma 2.3.11 (Generation).

1. *If* $\Gamma \vdash \texttt{new}\, x : X$*, then there exist* Δ*,* Δ'*,* A*, and* Y *such that* $\Gamma = \Delta, x \prec A, \Delta'$*, and* $\Delta \vdash A : Y$ *with* $X = \langle Y^i + x, Y^o + x \rangle$*.*
2. *If* $\Gamma \vdash AB : Z$ *with* $A, B \neq \epsilon$*, then there exist* X*,* Y *such that* $\Gamma \vdash A : X$*,* $\Gamma \vdash B : Y$ *and* $Z = \langle X^i \cup (X^o + Y^i), X^o + Y^o \rangle$*.*
3. *If* $\Gamma \vdash (A + B) : Z$*, then there exist* X*,* Y *such that* $\Gamma \vdash A : X$ *and* $\Gamma \vdash B : Y$ *and* $Z = \langle X^i \cup Y^i, X^o \cup Y^o \rangle$*.*
4. *If* $\Gamma \vdash \{A\} : Z$*, then there exists* X *such that* $\Gamma \vdash A : X$ *and* $Z = \langle X^i, [] \rangle$*.*

Proof. By induction on typing derivations.

1. $\Gamma \vdash \texttt{new}\, x : X$ can only be derived by the rule New or WeakenB. If it is derived by the rule New, then there is only one possibility:

   $$\text{(New)}\quad \frac{\Delta \vdash B : Y \quad x \notin \mathsf{dom}(\Delta)}{\Delta, x \prec B \vdash \texttt{new}\, x : X}$$

   with $X = \langle Y^i + x, Y^o + x \rangle$ and $\Gamma = \Delta, x \prec B$, so that Δ' is empty.

   If $\Gamma \vdash \texttt{new}\, x : X$ is derived by the rule WeakenB:

   $$\text{(WeakenB)}\quad \frac{\Gamma' \vdash \texttt{new}\, x : X \quad \Gamma' \vdash B : Y \quad y \notin \mathsf{dom}(\Gamma)}{\Gamma', y \prec B \vdash \texttt{new}\, x : X}$$

   then $\Gamma' \vdash \texttt{new}\, x : X$ and by the induction hypothesis applied to $\Gamma' \vdash \texttt{new}\, x : X$, there exist Δ_1, Δ_2 and A such that $\Gamma' = \Delta_1, x \prec A, \Delta_2$ and $\Delta_1 \vdash A : Y$ with $X = \langle Y^i + x, Y^o + x \rangle$. Take $\Delta = \Delta_1$, $\Delta' = \Delta_2, y \prec B$ the clause is proved.

2. $\Gamma \vdash AB : Z$ with $A, B \neq \epsilon$ can only be derived by the rule Seq or WeakenB. If $\Gamma \vdash AB : Z$ is derived by the rule Seq with two component expressions A and B in the premise of the typing rule:

   $$\text{(Seq)}\quad \frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad A, B \neq \epsilon}{\Gamma \vdash AB : \langle X^i \cup (X^o + Y^i), X^o + Y^o \rangle}$$

   We get all the conclusions immediately.

   If $\Gamma \vdash AB : Z$ is derived by the rule Seq with two component expressions $A_1 \neq A$ and $B_1 \neq B$ such that $A_1 B_1 = AB$:

   $$\text{(Seq)}\quad \frac{\Gamma \vdash A_1 : X_1 \quad \Gamma \vdash B_1 : Y_1 \quad A_1, B_1 \neq \epsilon}{\Gamma \vdash A_1 B_1 : \langle X_1^i \cup (X_1^o + Y_1^i), X_1^o + Y_1^o \rangle}$$

   then $Z = \langle X_1^i \cup (X_1^o + Y_1^i), X_1^o + Y_1^o \rangle$. There are two possibilities:

   - $A = A_1 A_2$ for some $A_2 \neq \epsilon$: then $B_1 = A_2 B$ and we have $\Gamma \vdash A_2 B : Y_1$.
     By the induction hypothesis applied to $\Gamma \vdash A_2 B : Y_1$, we get $\Gamma \vdash A_2 : X_2$ and $\Gamma \vdash B : Y$ and
     $$Y_1 = \langle X_2^i \cup (X_2^o + Y^i), X_2^o + Y^o \rangle$$

Now we can apply the rule Seq to $\Gamma \vdash A_1 : X_1$ and $\Gamma \vdash A_2 : X_2$:

$$\text{(Seq)}\ \frac{\Gamma \vdash A_1 : X_1 \quad \Gamma \vdash A_2 : X_2 \quad A_1, A_2 \neq \epsilon}{\Gamma \vdash A_1 A_2 : \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o \rangle}$$

and we get $\Gamma \vdash A : X$ where $X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o \rangle$.
Now we can apply the rule Seq to $\Gamma \vdash A : X$ and $\Gamma \vdash B : Y$ and we get

$$\Gamma \vdash AB : \langle X^i \cup (X^o + Y^i), X^o + Y^o \rangle$$

We still need to show that $Z = \langle X^i \cup (X^o + Y^i), X^o + Y^o \rangle$. This holds by Lemma 2.3.10.

- $B = B_0 B_1$: then $A_1 = AB_0$. Analogous to the previous subcase.

If $\Gamma \vdash AB : Z$ is derived by the rule WeakenB:

$$\text{(WeakenB)}\ \frac{\Gamma' \vdash AB : Z \quad \Gamma' \vdash C : V \quad y \notin \mathsf{dom}(\Gamma')}{\Gamma', y \prec C \vdash AB : Z}$$

with $\Gamma = \Gamma', y \prec C$ then by the induction hypothesis applied to $\Gamma' \vdash AB : Z$, we have $\Gamma' \vdash A : X$, $\Gamma' \vdash B : Y$ and $Z = \langle X^i \cup (X^o + Y^i), X^o + Y^o \rangle$. Now applying the rule WeakenB to $\Gamma' \vdash A : X$ and $\Gamma' \vdash B : Y$ with $\Gamma' \vdash C : V$ we reach the conclusions.

3. $\Gamma \vdash (A + B) : Z$ can only be derived by the rule Choice or WeakenB.

   If it is derived by the rule Choice, then there is only one possibility:

$$\text{(Choice)}\ \frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y}{\Gamma \vdash (A + B) : \langle X^i \cup Y^i, X^o \cup Y^o \rangle}$$

   with $Z = \langle X^i \cup Y^i, X^o \cup Y^o \rangle$. The conclusion follows immediately.

   If $\Gamma \vdash (A + B) : Z$ is derived by the rule WeakenB:

$$\text{(WeakenB)}\ \frac{\Gamma' \vdash (A + B) : Z \quad \Gamma' \vdash E : V \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x \prec E \vdash (A + B) : Z}$$

   then by the induction hypothesis applied to $\Gamma' \vdash (A + B) : Z$, we get $\Gamma' \vdash A : X$ and $\Gamma' \vdash B : Y$. By weakening these two judgments with $\Gamma' \vdash E : V$, we get the conclusions $\Gamma \vdash A : X$ and $\Gamma \vdash B : Y$.

4. $\Gamma \vdash \{A\} : Z$ can only be derived by the rule Scope or WeakenB.

   If it is derived by the rule Scope, then there is only one possibility:

$$\text{(Scope)}\ \frac{\Gamma \vdash A : X}{\Gamma \vdash \{A\} : \langle X^i, [] \rangle}$$

   with $Z = \langle X^i, [] \rangle$. The conclusion follows immediately.

   If $\Gamma \vdash \{A\} : Z$ is derived by the rule WeakenB:

$$\text{(WeakenB)}\ \frac{\Gamma' \vdash \{A\} : X \quad \Gamma' \vdash E : V \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x \prec E \vdash \{A\} : X}$$

then the proof is analogous to the subcase WEAKENB of case 3.

□

The next lemma stresses the significance of the order of declarations in a legal basis. In addition, because of the weakening rule WEAKENB, there can be many legal bases under which a well-typed expression can be derived.

Lemma 2.3.12 (Weakening).

1. *If $\Gamma = \Delta, x \prec E, \Delta'$ is legal, then $\Delta \vdash E : Z$ for some Z.*
2. *If $\Gamma \vdash E : Z$ and Γ is an initial segment of a legal basis Γ', then $\Gamma' \vdash E : Z$.*

Proof. 1. Since Γ is legal, by Definition 2.3.8, there exist B, Y such that $\Gamma \vdash B : Y$. In the typing derivation tree for B, the only way to extend Δ to $\Delta, x \prec E$ is by applying the rule NEW, or WEAKENB.

(NEW)
$$\frac{\Delta \vdash E : Z \quad x \notin \mathsf{dom}(\Delta)}{\Delta, x \prec E \vdash \mathtt{new}\, x : \langle Z^i + x, Z^o + x \rangle}$$

(WEAKENB)
$$\frac{\Delta \vdash A : X \quad \Delta \vdash E : Z \quad x \notin \mathsf{dom}(\Delta)}{\Delta, x \prec E \vdash A : X}$$

Each of the rules has $\Delta \vdash E : Z$ as a premise.

2. Since Γ is an initial segment of Γ', suppose that $\Gamma' = \Gamma, x_1 \prec A_1, \ldots, x_n \prec A_n$. By clause 1 we have for all $k = 1..n$, there exists X_k such that

$$\Gamma, x_1 \prec A_1, \ldots, x_{k-1} \prec A_{k-1} \vdash A_k : X_k$$

By applying the rule WEAKENB for $k = 1$, we get

(WEAKENB)
$$\frac{\Gamma \vdash E : Z \quad \Gamma \vdash A_1 : X_1 \quad x_1 \notin \mathsf{dom}(\Gamma)}{\Gamma, x_1 \prec A_1 \vdash E : Z}$$

Repeat applying the rule WEAKENB for $k = 2..n$, we get the conclusion.

□

The following lemma can be viewed as the inverse of Weakening Lemma 2.3.12. Under certain conditions we can contract a legal basis so that the expression is still well-typed in the smaller basis, an initial segment.

Lemma 2.3.13 (Strengthening). *If $\Gamma, x \prec A \vdash B : Y$ and $x \notin \mathsf{var}(B)$, then $\Gamma \vdash B : Y$ and $x \notin Y^i$.*

Proof. By induction on typing derivations. Let $\Gamma' = \Gamma, x \prec A$.

- Case AXIOM: $B = \epsilon$, does not apply since the basis is not empty.
- Case NEW: $B = \mathtt{new}\, x$, does not apply since $\mathsf{var}(B) = \mathsf{var}(\mathtt{new}\, x) = \{x\}$.
- Case WEAKENB:

(WEAKENB)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec A \vdash B : Y}$$

We have immediately the first conclusion $\Gamma \vdash B : Y$ in the premise. By Lemma 2.3.9 applied to the conclusion, we get $\mathsf{dom}(Y^*) \subseteq \mathsf{dom}(\Gamma)$. Since $x \notin \mathsf{dom}(\Gamma)$ by the side condition, we get $x \notin Y^i$.

- Case Seq:

$$\frac{\Gamma' \vdash B_1 : Y_1 \quad \Gamma' \vdash B_2 : Y_2 \quad B_1, B_2 \neq \epsilon}{\Gamma' \vdash B_1 B_2 : \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o \rangle} \text{(Seq)}$$

  with $Y = \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o \rangle$ and $B = B_1 B_2$.

  Since $x \notin \mathsf{var}(B_1 B_2) = \mathsf{var}(B_1) \cup \mathsf{var}(B_2)$, we have $x \neq \mathsf{var}(B_1)$ and $x \notin \mathsf{var}(B_2)$. By the induction hypothesis, we get $\Gamma \vdash B_1 : Y_1$ and $x \notin Y_1^i$, $\Gamma \vdash B_2 : Y_2$ and $x \notin Y_2^i$. Now we can apply the rule Seq to get the conclusions: $\Gamma \vdash B_1 B_2 : Y$ and $x \notin Y_1^i \cup (Y_1^o + Y_2^i)$.

- Case Choice: then $B = (B_1 + B_2)$, the proof is analogous to the case Seq.

- Case Scope: then $B = \{C\}$, the proof is analogous to the case Seq.

□

When an expression has a type, this type is unique, although there can be many typing derivations for an expression. The uniqueness of type is useful when we want types to be a part of the component specification.

Proposition 2.3.14 (Uniqueness of types). *If* $\Gamma \vdash A : X$ *and* $\Gamma \vdash A : Y$, *then* $X^i = Y^i$ *and* $X^o = Y^o$.

Proof. By induction on typing derivations.

- Base case Axiom: We have $A = \epsilon$ and $\Gamma = \varnothing$ and $X = Y = \langle [], [] \rangle$ so the case holds trivially.

- Case New:

$$\frac{\Gamma' \vdash B : U \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma', x \prec B \vdash \mathtt{new}\, x : X} \text{(New)}$$

  with $X = \langle U^i + x, U^o + x \rangle$ and $\Gamma = \Gamma', x \prec B$.

  Assume Proposition 2.3.14 holds for the premise of this rule and let $\Gamma \vdash \mathtt{new}\, x : Y$. By Generation Lemma 2.3.11, clause 1, there exist Δ_1, Δ_2, C such that $\Gamma = \Delta_1, x \prec C, \Delta_2$ and $\Delta_1 \vdash C : V$ with $Y = \langle V^i + x, V^o + x \rangle$.

  By Lemma 2.3.9, there is only one declaration of x in Γ. This means $\Delta_1 = \Gamma'$, $C = B$ and Δ_2 is empty. So we have $\Gamma' \vdash B : V$. By the induction hypothesis we have $U^* = V^*$ which implies $X^* = Y^*$.

- Case WeakenB:

$$\frac{\Gamma' \vdash A : X \quad \Gamma' \vdash B : Z \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x \prec B \vdash A : X} \text{(WeakenB)}$$

  with $\Gamma = \Gamma', x \prec B$.

  Assume Proposition 2.3.14 holds for the two premises and let $\Gamma \vdash A : Y$. Since $\Gamma' \vdash A : X$ and $x \notin \mathsf{dom}(\Gamma')$, we have $x \notin \mathsf{var}(A)$. By Lemma 2.3.13 applied to $\Gamma', x \prec B \vdash A : Y$, we get $\Gamma' \vdash A : Y$. By the induction hypothesis we have the conclusion $X = Y$.

- Case Seq:

$$\frac{\Gamma \vdash B_1 : Y_1 \quad \Gamma \vdash B_2 : Y_2 \quad B_1, B_2 \neq \epsilon}{\Gamma \vdash B_1 B_2 : \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o \rangle} \text{(Seq)}$$

  with $A = B_1 B_2$ and $X = \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o \rangle$.

By Generation Lemma 2.3.11, clause 1 applied to $\Gamma \vdash B_1B_2 : Y$, we have $\Gamma \vdash B_1 : V_1$, $\Gamma \vdash B_2 : V_2$ and $Y = \langle V_1^i \cup (V_1^o + V_2^i), V_1^o + V_2^o \rangle$. By the induction hypothesis, we have $Y_1 = V_1$ and $Y_2 = V_2$. Hence, $X = Y = \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o \rangle$.

- Case Choice: analogous to the case Seq.
- Case Scope: analogous to the case Seq.

□

2.3.3 Soundness Proofs

In this section, after showing the proofs for Preservation Lemma 2.3.4, Progress Lemma 2.3.5 and Soundness Theorem 2.3.7, we prove the termination property of all well-typed programs.

Proof of Lemma 2.3.4 (Preservation). *If* $\Gamma \models \mathbf{S}$ *and* $\mathbf{S} \longrightarrow \mathbf{S}'$, *then* $\Gamma \models \mathbf{S}'$.

Proof. By the Definition 2.3.1 of well-typed configurations we need to prove that for all pairs (M, E) in $\mathbf{S}'$ there exists X such that $\Gamma \vdash E : X$.

The proof proceeds by case analysis on the relation $\longrightarrow$. In each case, we only need to prove the well-typedness of the new expressions in $\mathbf{S}'$. We assume that $\mathsf{hi}(\mathbf{S}) = n$.

- Case osNew:

$$\frac{(\mathsf{osNew}) \quad x \prec A \in \mathit{Decls}}{\mathbf{S}_1 \circ (M, \mathtt{new}\, x E) \longrightarrow \mathbf{S}_1 \circ (M + x, AE)}$$

Since the two stacks $\mathbf{S}$ and $\mathbf{S}'$ are only different at their tops, we only need to prove that $\Gamma \vdash AE : Z$ for some Z.

Since $\Gamma \models \mathbf{S}$, we have $\Gamma \vdash \mathtt{new}\, x E : X$. By Generation Lemma 2.3.11, clause 2, we have $\Gamma \vdash \mathtt{new}\, x : X_1$ and $\Gamma \vdash E : X_2$ with $X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o \rangle$.

Also by Generation Lemma 2.3.11, clause 1 applied to $\Gamma \vdash \mathtt{new}\, x : X_1$ and Lemma 2.3.12, we get $\Gamma \vdash A : Y$ with $X_1 = \langle Y^i + x, Y^o + x \rangle$. Now sequencing $\Gamma \vdash A : Y$ and $\Gamma \vdash E : X_2$, we get $\Gamma \vdash AE : Z$ with $Z = \langle Y^i \cup (Y^o + X_2^i), Y^o + X_2^o \rangle$.

- Case osChoice:

$$\frac{(\mathsf{osChoice}) \quad i \in \{1, 2\}}{\mathbf{S}_1 \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S}_1 \circ (M, A_i E)}$$

We treat the case $i = 1$, the case $i = 2$ is symmetric. We only need to prove that $\Gamma \vdash A_1 E : Z$ for some Z.

Since $\Gamma \models \mathbf{S}$, we have $\Gamma \vdash (A_1 + A_2)E : X$. By Generation Lemma 2.3.11, clause 2 applied to $\Gamma \vdash (A_1 + A_2)E : X$, we get $\Gamma \vdash (A_1 + A_2) : X_1$ and $\Gamma \vdash E : X_2$ with $X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o \rangle$.

Also by Generation Lemma 2.3.11, clause 3 applied to $\Gamma \vdash (A_1 + A_2) : X_1$, we get $\Gamma \vdash A_1 : Y_1$ and $\Gamma \vdash A_2 : Y_2$ with $X_1 = \langle Y_1^i \cup Y_2^i, Y_1^o \cup Y_2^o \rangle$. Then we can apply the rule Seq to $\Gamma \vdash A_1 : Y_1$ and $\Gamma \vdash E : X_2$ and get $\Gamma \vdash A_1 E : Z$ with $Z = \langle Y_1^i \cup (Y_1^o + X_2^i), Y_1^o + X_2^o \rangle$.

- Case osPush:

$$\frac{(\mathsf{osPush})}{\mathbf{S}_1 \circ (M, \{A\}E) \longrightarrow \mathbf{S}_1 \circ (M, E) \circ ([], A)}$$

We need to prove that $\Gamma \vdash A : Y$ and $\Gamma \vdash E : Z$ for some Y, Z.

Since $\Gamma \models \mathbf{S}$, we have $\Gamma \vdash \{A\}E : X$. By Generation Lemma 2.3.11, clauses 2 and 4, we have $\Gamma \vdash \{A\} : X_1$ with $X_1^o = []$ and $\Gamma \vdash E : Z$ with (for use in proof of

Lemma 2.3.6)

$$\begin{aligned} X &= \langle X_1^i \cup (X_1^o + Z^i), X_1^o + Z^o \rangle \\ &= \langle X_1^i \cup Z^i, Z^o \rangle \end{aligned}$$

Also by Lemma 2.3.11, clause 4 applied to $\Gamma \vdash \{A\} : X_1$, we get $\Gamma \vdash A : Y$ with $Y^i = X_1^i$.

- Case osPop:

$$\begin{array}{l} (\mathsf{osPop}) \\ \mathbf{S}_1 \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S}_1 \circ (M, E) \end{array}$$

The clause holds by the hypothesis.

□

Proof of Lemma 2.3.5 (Progress). *If $\Gamma \models \mathbf{S}$, then either $\mathbf{S}$ is terminal or there exists a configuration $\mathbf{S}'$ such that $\mathbf{S} \longrightarrow \mathbf{S}'$.*

Proof. Among all the transition rules, the only case where the execution may get stuck is that E has the form $\mathtt{new}\, xA$ and $x \notin \mathsf{dom}(\Gamma)$. But this has been guaranteed by Lemma 2.3.9. □

Proof of Lemma 2.3.6 (Invariants of maxins). *If $\Gamma \models \mathbf{S}$ and $\mathbf{S} \longrightarrow \mathbf{S}'$, then $\mathsf{maxins}(\mathbf{S}) \supseteq \mathsf{maxins}(\mathbf{S}')$.*

Proof. The proof proceeds by case analysis on the transition relation $\longrightarrow$. Let $\mathbf{S} = (M_1, C_1) \circ \ldots \circ (M_n, C_n)$ and $\mathbf{S}' = (N_1, E_1) \circ \ldots \circ (N_m, E_m)$. In Lemma 2.3.4 we have proved that all expressions in $\mathbf{S}'$ are well-typed. We assume that C_k has type V_k for $k = 1..n$ and E_k has type Z_k for $k = 1..m$.

To prove the clause, by the definition of maxins, we only need to show that for each element in $\{[\mathbf{S}'|_k] + Z_k^i \mid k = 1..\mathsf{hi}(\mathbf{S}')\}$ there exists an element in $\{[\mathbf{S}|_k] + V_k^i \mid k = 1..\mathsf{hi}(\mathbf{S})\}$ such that the latter multiset includes the former.

- Case osNew:

$$\begin{array}{l} (\mathsf{osNew}) \quad x \prec A \in \mathit{Decls} \\ \mathbf{S}_1 \circ (M, \mathtt{new}\, xE) \longrightarrow \mathbf{S}_1 \circ (M + x, AE) \end{array}$$

We have $m = n$. For $k < n$ we have $[\mathbf{S}|_k] + V_k^i = [\mathbf{S}'|_k] + Z_k^i$ since the two stacks $\mathbf{S}|_k$ and $\mathbf{S}'|_k$ are the same.

For $k = n$ we will prove that $[\mathbf{S}|_n] + V_n^i = [\mathbf{S}'|_n] + Z_n^i$. Using $V_n = X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o \rangle$ and $Z_n = Z = \langle Y^i \cup (Y^o + X_2^i), Y^o + X_2^o \rangle$ in the proof of Lemma 2.3.4, we have:

$$\begin{aligned} [\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && (V_n = X) \\ &= [\mathbf{S}|_n] + X_1^i \cup (X_1^o + X_2^i) && (X^i = X_1^i \cup (X_1^o + X_2^i)) \\ &= [\mathbf{S}|_n] + ((Y^i + x) \cup (Y^o + x + X_2^i)) && (X_1^* = Y^* + x) \\ &= [\mathbf{S}|_n] + x + (Y^i \cup (Y^o + X_2^i)) \\ &= [\mathbf{S}'|_n] + Z^i && (Z^i = Y^i \cup (Y^o + X_2^i)) \\ &= [\mathbf{S}'|_n] + Z_n^i && (Z_n = Z) \end{aligned}$$

- Case osChoice:

$$\begin{array}{l} (\mathsf{osChoice}) \quad i \in \{1, 2\} \\ \mathbf{S}_1 \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S}_1 \circ (M, A_i E) \end{array}$$

We treat the case $i = 1$, the other case is symmetric. We have $m = n$. For $k < n$ we have $[\mathbf{S}|_k] + V_k^i = [\mathbf{S}'|_k] + Z_k^i$ since the two stacks $\mathbf{S}|_k$ and $\mathbf{S}'|_k$ are the same.

For $k = n$ we will prove that $[\mathbf{S}|_n] + V_n^i \supseteq [\mathbf{S}'|_n] + Z_n^i$. Using $V_n = X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o\rangle$ and $Z_n = Z = \langle Y_1^i \cup (Y_1^o + X_2^i), Y_1^o + X_2^o\rangle$ in the proof of Lemma 2.3.4, we have:

$$\begin{aligned}
[\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && (V_n = X)\\
&= [\mathbf{S}|_n] + X_1^i \cup (X_1^o + X_2^i) && (X^i = X_1^i \cup (X_1^o + X_2^i))\\
&\supseteq [\mathbf{S}|_n] + (Y_1^i \cup (Y_1^o + X_2^i)) && (X_1^* = Y_1^* \cup Y_2^*)\\
&= [\mathbf{S}'|_n] + Z^i && (Z^i = Y_1^i \cup (Y_1^o + X_2^i))\\
&= [\mathbf{S}'|_n] + Z_n^i && (Z_n = Z)
\end{aligned}$$

- Case osPush:

$$\begin{array}{l}(\mathsf{osPush})\\ \mathbf{S}_1 \circ (M, \{A\}E) \longrightarrow \mathbf{S}_1 \circ (M, E) \circ ([], A)\end{array}$$

We have $m = n + 1$. For $k < n$ we have $[\mathbf{S}|_k] + V_k^i = [\mathbf{S}'|_k] + Z_k^i$ since the two stacks $\mathbf{S}|_k$ and $\mathbf{S}'|_k$ are the same.

For $k = n$ we will prove that $[\mathbf{S}|_n] + V_n^i \supseteq [\mathbf{S}'|_n] + Z_n^i$. Using $V_n = X = \langle X_1^i \cup Z^i, Z^o\rangle$ and $Z_n = Z$ in the proof of Lemma 2.3.4, we have:

$$\begin{aligned}
[\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && (V_n = X)\\
&= [\mathbf{S}|_n] + (X_1^i \cup Z^i) && (X^i = X_1^i \cup Z^i)\\
&\supseteq [\mathbf{S}|_n] + Z^i && \\
&= [\mathbf{S}'|_n] + Z_n^i && (Z_n = Z)
\end{aligned}$$

For $k = n + 1$ we will prove that $[\mathbf{S}|_n] + V_n^i \supseteq [\mathbf{S}'|_{n+1}] + Z_{n+1}^i$. Using $V_n = X = \langle X_1^i \cup Z^i, Z^o\rangle$ and $Z_{n+1} = Y$ in the proof of Lemma 2.3.4, we have:

$$\begin{aligned}
[\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && (V_n = X)\\
&= [\mathbf{S}|_n] + (X_1^i \cup Z^i) && (X^i = X_1^i \cup Z^i)\\
&\supseteq [\mathbf{S}'|_n] + Y^i && (X_1^i = Y^i)\\
&= [\mathbf{S}'|_{n+1}] + Z_{n+1}^i && ([\mathbf{S}'(n+1)] = [],\ Z_{n+1} = Y)
\end{aligned}$$

- Case osPop:

$$\begin{array}{l}(\mathsf{osPop})\\ \mathbf{S}_1 \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S}_1 \circ (M, E)\end{array}$$

Then $m = n - 1$ and the clause holds easily since the two stacks $\mathbf{S}|_m$ and $\mathbf{S}'|_m$ are the same.

□

Proof of Theorem 2.3.7 (Soundness). *Let Prog = Decls; E be well-typed, that is,* $\Gamma \vdash E : X$ *for some reordering* Γ *of Decls and some type* X*. Then for any* $\mathbf{S}$ *such that* $([], E) \longrightarrow^* \mathbf{S}$ *we have that* $\mathbf{S}$ *is not stuck and* $[\mathbf{S}] \subseteq X^i$*.*

Proof. First, configuration $([], E)$ is well-typed by Definition 2.3.1. Then the first conclusion, $\mathbf{S}$ is not stuck, follows by Lemma 2.3.4 and Lemma 2.3.5. For the second conclusion, since $[\mathbf{S}] \subseteq \mathsf{maxins}(\mathbf{S}|_{\mathsf{hi}(\mathbf{S})})$ and $\mathsf{maxins}(([], E)) = X^i$, the upper bounds of instances, $[\mathbf{S}] \subseteq X^i$, follow by Lemma 2.3.6 and the transitivity of $\subseteq$. □

Termination. The last property to be considered is the termination of well-typed programs. The common tool for proving the termination of programs (cf. [14, 31]) is to find a *termination function* (also called *termination measure* [31]) which maps program states to a *well-founded set*. A well-founded set is a set S with an ordering $>$ on elements of S such that there can be no infinite descending sequences of elements.

Since we do not allow recursion and mutual recursion in the declarations, any well-typed program terminates after a finite number of transition steps. We can even calculate the maximum number of steps that a well-typed program takes to terminate. The function mts below is defined for an expression, but we will overload it for a configuration. We choose the set of natural numbers $\mathbb{N}$ and the usual ordering $>$ to be a well-founded set $(\mathbb{N}, >)$. The function mts, which takes an expression and returns a natural number, is defined recursively as follows.

$$\mathsf{mts}(E) = \begin{cases} 0, & \text{if } E = \epsilon \\ \mathsf{mts}(A) + \mathsf{mts}(B), & \text{if } E = AB \\ 1 + \mathsf{mts}(A), & \text{if } E = \texttt{new}\, x \text{ and } x \prec A \in \mathit{Decls} \\ 1 + max(\mathsf{mts}(A), \mathsf{mts}(B)), & \text{if } E = (A + B) \\ 2 + \mathsf{mts}(A), & \text{if } E = \{A\} \end{cases}$$

The integers $0, 1$ and 2 in the definition are the corresponding steps of the operational semantics. This function is well-defined for any well-typed expression and terminates as we will see in the analysis in Section 2.4 below. Now we overload the function for the configuration $\mathbf{S} = (M_1, E_1) \circ \ldots \circ (M_n, E_n)$:

$$\mathsf{mts}(\mathbf{S}) = \sum_{i=1}^{n} \mathsf{mts}(E_i) + n - 1$$

In the definition, $n - 1$ is the number of osPop steps.

Note that, if E is the main expression of a well-typed program, then $\mathsf{mts}(E)$ is the maximum transition steps that the program takes to terminate in any run, not all possible runs of the program because there may be an exponential number of such runs. The following theorem guarantees the termination of all well-typed programs.

Theorem 2.3.15 (Termination).

1. *If* $\Gamma \models \mathbf{S}$ *and* $\mathbf{S} \longrightarrow \mathbf{S}'$, *then* $\mathsf{mts}(\mathbf{S}) > \mathsf{mts}(\mathbf{S}')$.
2. *A well-typed program always terminates in a finite number of steps.*

Proof. The second clause follows by the first one since the initial configuration of a well-typed program is well-typed, as in the proof of Theorem 2.3.7. The proof of the first clause proceeds by case analysis on the transition relation $\longrightarrow$.

Let $\mathbf{S} = (M_1, C_1) \circ \ldots \circ (M_n, C_n)$ and $\mathbf{S}' = (N_1, E_1) \circ \ldots \circ (N_m, E_m)$.

- Case osNew:

  $$\frac{(\mathsf{osNew}) \quad x \prec A \in \mathit{Decls}}{\mathbf{S}_1 \circ (M, \texttt{new}\, x E) \longrightarrow \mathbf{S}_1 \circ (M + x, AE)}$$

  We have $m = n$ and $\mathbf{S}$ and $\mathbf{S}'$ are only different at their tops. In addition, since $\mathsf{mts}(C_n) = \mathsf{mts}(\texttt{new}\, xE) = \mathsf{mts}(\texttt{new}\, x) + \mathsf{mts}(E) > \mathsf{mts}(A) + \mathsf{mts}(E) = \mathsf{mts}(AE) = \mathsf{mts}(E_m)$, the clause follows.

- Case osChoice:

  $$\frac{(\mathsf{osChoice}) \quad i \in \{1, 2\}}{\mathbf{S}_1 \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S}_1 \circ (M, A_i E)}$$

We treat the case $i = 1$, the other case is symmetric.

We have $m = n$ and $\mathbf{S}$ and $\mathbf{S}'$ are only different at their tops. In addition, since $\mathsf{mts}(C_n) = \mathsf{mts}((A_1 + A_2)E) = \mathsf{mts}((A_1 + A_2)) + \mathsf{mts}(E) > \mathsf{mts}(A_1) + \mathsf{mts}(E) = \mathsf{mts}(A_1E) = \mathsf{mts}(E_m)$, the clause follows.

- Case osPush:

$$\begin{array}{l}(\mathsf{osPush})\\ \mathbf{S}_1 \circ (M, \{A\}E) \longrightarrow \mathbf{S}_1 \circ (M, E) \circ ([], A)\end{array}$$

We have $m = n + 1$ and two stacks $\mathbf{S}|_{n-1}$ and $\mathbf{S}'|_{n-1}$ are identical. In addition, since $\mathsf{mts}((M_n, C_n)) = \mathsf{mts}(C_n) = \mathsf{mts}(\{A\}E) = \mathsf{mts}(\{A\}) + \mathsf{mts}(E) = 2 + \mathsf{mts}(A) + \mathsf{mts}(E) > 1 + \mathsf{mts}(A) + \mathsf{mts}(E) = \mathsf{mts}((N_n, E_n) \circ (N_m, E_m))$, so $\mathsf{mts}(\mathbf{S}) > \mathsf{mts}(\mathbf{S}')$ and then the clause follows.

- Case osPop:

$$\begin{array}{l}(\mathsf{osPop})\\ \mathbf{S}_1 \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S}_1 \circ (M, E)\end{array}$$

We have $m = n - 1$ and the clause holds easily since the two stacks $\mathbf{S}|_m$ and $\mathbf{S}'|_m$ are the same.

□

2.4 Type Inference

So far we know that a well-typed program is safe to execute. Now given a well-formed program, if we can derive the type of its main expression, then we know whether the program is safe to execute on a given system. The problem of finding a type/derivation of an expression, given a set of declarations, is the *type inference problem* [9,31] or typability problem [4]. Types inferred give information about component programs such as memory and resources they may use, and hence guide the design of the component system.

Now let us see a solution for our type inference problem. Let *Prog* be the component program and E be the expression we need to find the type of. A necessary (but not sufficient) condition for type inference is that the declarations in *Prog* can be reordered into a basis Γ such that for any declaration $x \prec A$ in Γ, the variables occurring in A are already declared previously in Γ. In other words:

$$\text{if } \Gamma = \Delta, x \prec A, \Delta' \text{ then } \mathsf{var}(A) \subseteq \mathsf{dom}(\Delta) \tag{2.6}$$

The existence of such a reordering can be detected in polynomial time by an analysis of the dependency graph associated with the declarations in *Prog*. From now on we assume that Γ is a basis consisting of all declarations in *Prog* and satisfying (2.6). The considerations below are independent of which particular ordering is used as long as it satisfies (2.6).

The basic idea behind the type inference algorithm is to exploit the fact that the typing rules are syntax-directed, or, in other words, to use the Generation Lemma 2.3.11 reversely.

We can break down the problem of finding type for E by finding types of $\mathtt{new}\,x$ for all $x \in \mathsf{var}(E)$. Why? First, we can recursively break down expression E into $E_1, \ldots, E_m$ for some m such that E_i is one of the forms: $\mathtt{new}\,x$, $(E + E)$, $\{E\}$ and $E = E_1 \ldots E_m$. By Definition 2.2.2 we can easily calculate the type of E if we know types of all E_i. Moreover, $\mathsf{var}(E_i) \subseteq \mathsf{var}(E)$ so if we know types of $\mathtt{new}\,x$ for all $x \in \mathsf{var}(E)$ we can calculate types of E_i by doing some multisets operations in Definition 2.2.2. The type inference problem for E now becomes type inference problems of $\mathtt{new}\,x$ for all $x \in \mathsf{var}(E)$.

To find the type of $\mathtt{new}\,x$ we can look up the declaration of x in the basis Γ. If no declaration of x can be found then no type can be inferred. Otherwise $\Gamma = \Delta, x \prec A, \Delta'$

for some Δ, Δ' and A and clause 1 of the Generation Lemma 2.3.11 allows us to reduce the problem to inferring the type of A in Δ, together with the additional task of checking if Δ' legally extends $\Delta, x \prec A$. Here some care has to be taken in order to stay polynomial. A naive recursive algorithm could behave exponentially by generating recursively duplicate instances of the same type inference problem. Duplication can, however, be avoided by storing solved instances.

Observe that all instances are of the form: infer the type of A in Δ, where Δ is an initial segment of the basis of the original type inference problem and A is a sub-expression of one of its constituents. There are polynomially many such instances and hence type inference can be done in polynomial time.

Chapter 3

Explicit Deallocation

In the previous chapter, to put an expression in a scope is the only way to deallocate instances. When the control exits a scope, all the instances in the current local store are discarded. In this chapter, we extend the language of the previous chapter with an explicit deallocation primitive, which can remove from the local store a single instance at a time.

The chapter is organized as in Chapter 2. First, the syntax and the operational semantics of the language are defined. Then we develop a type system which can find the (sharp) upper bounds of resources for a class of programs. Last, we prove some important properties of the type system.

3.1 Language

Adding the explicit deallocation primitive causes only a few small changes to the syntax and the operational semantics. We will focus on explaining the new features in this section.

3.1.1 Syntax

Table 3.1 defines the syntax of the language with an explicit deallocation primitive. We use the extended Backus-Naur Form as in Section 2.1.1. The only new syntax for expressions is the primitive `del` for deallocating an instance of a component. The other ingredients of the language (programs, declarations) are the same as in Chapter 2.

Table 3.1: Syntax of the language with `del`

Prog	$::=$	$\mathit{Decls}; E$	Program
Decls	$::=$	$\overline{x \prec E}$	Declarations
E	$::=$		Expression
		ϵ	Empty
	$\mid$	$\mathtt{new}\, x$	Instantiation
	$\mid$	$\mathtt{del}\, x$	Deallocation
	$\mid$	$E\ E$	Sequencing
	$\mid$	$(E + E)$	Choice
	$\mid$	$\{E\}$	Scope

We give an example program for demonstrating the operational semantics and typing derivations in the subsequent sections. In this example, d and e are primitive components. Component a first creates an instance of component e, then it creates an instance of d in a new scope. Component b has a choice expression before deleting an instance of

component e.

$$d \prec \epsilon \quad e \prec \epsilon$$
$$a \prec \texttt{new}\, e \{ \texttt{new}\, d \}$$
$$b \prec (\texttt{new}\, a + \texttt{new}\, e\, \texttt{new}\, d)\, \texttt{del}\, e;$$
$$\texttt{new}\, b$$

3.1.2 Operational Semantics

Table 3.2 defines a small-step operational semantics. As in Section 2.1.2, the operational semantics is defined in terms of a transition system of configurations and a configuration is a stack of pairs (M, E) of a multiset M and an expression E defined in Table 3.1. The notation of stacks and the notion of *terminal* configuration and the multi-step transition relation $\longrightarrow^*$ are the same as in Chapter 2.

In Table 3.2, the transition rules for $\texttt{new}\, x$, choice, scope, and sequential composition are the same as in Section 2.1.2. Executing $\texttt{new}\, x$ adds an x to the local store then continues with the 'body' A of x. Executing $(A_1 + A_2)$ chooses either A_1 or A_2 to execute with the same local store. Executing $\{A\}$ pushes a new pair $([], A)$ on the top of the stack and the execution is continued with the new pair. When the new pair terminates in form (M, ϵ), it is popped from the stack and the execution is continued with the new top of the stack.

The only new transition rule is the rule osDel. Executing the deallocation primitive $\texttt{del}\, x$ removes an instance of x from the local store, if there exists at least one x in the store. Then the execution continues with the next command. If there exists no instance of x in the local store, the execution is stuck. This happening is one of the runtime errors that the type system in the next section will check.

Note that the operational semantics of `new` and `del` is not symmetric. In the rule osNew, after creating a new instance of x we continue executing the body A of x, while in the rule osDel, after removing an instance of x, we do not continue with the body of x.

Also, here we have abstracted from the specific instance that is deleted. That is, if there are several instances of a component x, we do not care about which x is deleted; We only care that one x is removed from the store. For the purpose of finding the resource bounds, this abstraction is fine. Type systems for the safety of deallocating a specific instance have been studied elsewhere, cf. [32, 44].

Table 3.2: Transition rules of the language with `del`

(osNew) $\quad x \prec A \in \mathit{Decls}$
$$\mathbf{S} \circ (M, \texttt{new}\, xE) \longrightarrow \mathbf{S} \circ (M + x, AE)$$

(osDel) $\quad x \in M$
$$\mathbf{S} \circ (M, \texttt{del}\, xE) \longrightarrow \mathbf{S} \circ (M - x, E)$$

(osChoice) $\quad i \in \{1, 2\}$
$$\mathbf{S} \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S} \circ (M, A_i E)$$

(osPush)
$$\mathbf{S} \circ (M, \{A\}E) \longrightarrow \mathbf{S} \circ (M, E) \circ ([], A)$$

(osPop)
$$\mathbf{S} \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S} \circ (M, E)$$

The example at the end of Section 3.1.1 is used to illustrate the operational semantics. For this simple program there are two possible runs of the program due to the choice

composition. Here we only show one of the possible runs.

$$
\begin{array}{rll}
(\mathsf{Start}) & ([\,], \texttt{new}\, b) & \\
(\mathsf{osNew}) \longrightarrow & ([b], (\texttt{new}\, a + \texttt{new}\, e\, \texttt{new}\, d)\, \texttt{del}\, e) & \\
(\mathsf{osChoice}) \longrightarrow & ([b], \texttt{new}\, a\, \texttt{del}\, e) & (\text{or } ([b], \texttt{new}\, e\, \texttt{new}\, d\, \texttt{del}\, e)) \\
(\mathsf{osNew}) \longrightarrow & ([b, a], \texttt{new}\, e\{\texttt{new}\, d\}\, \texttt{del}\, e) & \\
(\mathsf{osNew}) \longrightarrow & ([b, a, e], \{\texttt{new}\, d\}\, \texttt{del}\, e) & \\
(\mathsf{osPush}) \longrightarrow & ([b, a, e], \texttt{del}\, e) \circ ([\,], \texttt{new}\, d) & \\
(\mathsf{osNew}) \longrightarrow & ([b, a, e], \texttt{del}\, e) \circ ([d], \epsilon) & \\
(\mathsf{osPop}) \longrightarrow & ([b, a, e], \texttt{del}\, e) & \\
(\mathsf{osDel}) \longrightarrow & ([b, a], \epsilon) & (\text{terminal})
\end{array}
$$

3.2 Type System

As in the previous chapter, we have two main goals in designing the type system. For the first one, the soundness, besides checking the declarations of all variables as in the previous chapter, we need the safety of deallocation operation in the rule osDel. In this rule, the execution is stuck if the next operation is a deallocation of an instance of a component and there exists no instance of that component in the local store. We solve the problem by keeping a store in the typing environment, a technique inspired by linear type systems [32, 44]. For the second goal, we want to find the maximum number of instances that a program can create, as in Chapter 2.

Before defining types, we extend the notion of multiset to the notion of *signed multiset*. Recall that a multiset over a set S can be viewed as a map from S to the set of natural numbers $\mathbb{N}$. Similarly, a signed multiset M, also denoted by $[\ldots]$, over a set S is a map from S to the set of integers $\mathbb{Z}$. For a negative occurrence of an element, we put the sign '$-$' before the element. For example, $[x, -y, -y]$ is a signed multiset where the multiplicity of x is 1 and the multiplicity of y is -2.

The analogous operations of multisets are overloaded for signed multisets. Let M, N be signed multisets. Then $M(x)$ is the *multiplicity* (can be negative) of x; $M(x) = 0$ when x is not an element of M, notation $x \notin M$. Domain of M, notation $\mathsf{dom}(M)$, is the set of elements in M: $\mathsf{dom}(M) = \{x \mid M(x) \neq 0\}$. Other operations defined for two signed multisets are the same as those for multisets except the substraction; additive union: $(M + N)(x) = M(x) + N(x)$; substraction: $(M - N)(x) = M(x) - N(x)$; union: $(M \cup N)(x) = max(M(x), N(x))$; intersection: $(M \cap N)(x) = min(M(x), N(x))$; inclusion: $M \subseteq N$ if $M(x) \leq N(x)$ for all $x \in M$.

Now we can define types, which are triples of a multiset and two signed multisets.

Definition 3.2.1 (Types). *Types of component expressions are tuples*

$$X = \langle X^i, X^o, X^l \rangle$$

where X^i is a multiset (no negative occurrences) and X^o, X^l are signed multisets over $\mathbb{C}$.

Intuitively, the meaning of each part of a type triple is as follows. Suppose X is the type of an expression E. When executing E alone, X^i is the upper bound of the number of simultaneously active instances for all components during the execution of E, and X^o is the maximum number of instances that 'survive' at the end of the execution, as in Chapter 2. Here we have the deallocation primitive with its behaviour opposite to instantiation so we use a signed multiset for X^o. In composition, X^o is the maximal net effect (with respect to the change in the number of instances) to the runtime environment *before* and *after* the execution of E. Similarly, X^i in composition is the effect on the maximum during the execution. The pair $\langle X^i, X^o \rangle$ is enough to calculate the upper bound.

In addition, we want the safety of the deallocation primitives in composition. When sequencing E with $\texttt{del}\,x$, the safety of $\texttt{del}\,x$ depends on the minimum outcome of E. Therefore, we need X^l, which is the minimum number of instances that survive the execution of E. Analogous to X^o, in composition, X^l is the minimal net effect to the runtime environment before and after the execution of E. The difference between X^o and X^l is caused by the choice composition. More explanation is given shortly in the exposition of typing rules below.

A *basis* is a list of declarations: $x_1 \prec E_1, \ldots, x_n \prec E_n$, as in Chapter 2. A *store* σ is a multiset (no negative multiplicities) over $\mathbb{C}$, the set of component names. An *environment* is now extended with a store, so an environment is a pair of a store and a basis. A *typing judgment* is a tuple of the form:

$$\sigma, \Gamma \vdash E : X$$

and it asserts that expression E has type X in the environment σ, Γ. The idea of the store is that it should contain enough instances for the safety of `del` commands during the execution of the expression.

Definition 3.2.2 (Valid typing judgments). *Valid typing judgments $\sigma, \Gamma \vdash A : X$ are derived by applying the typing rules in Table 3.3 in the usual inductive way.*

Table 3.3: Typing rules of the language with `del`

(Axiom)
$$\frac{}{[], \varnothing \vdash \epsilon : \langle [], [], [] \rangle}$$

(WeakenB)
$$\frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma)}{\sigma_1, \Gamma, x \prec B \vdash A : X}$$

(WeakenS)
$$\frac{\sigma, \Gamma \vdash A : X \quad \sigma \subseteq \sigma_1}{\sigma_1, \Gamma \vdash A : X}$$

(New)
$$\frac{\sigma, \Gamma \vdash A : X \quad x \notin \mathsf{dom}(\Gamma)}{\sigma, \Gamma, x \prec A \vdash \texttt{new}\,x : \langle X^i + x, X^o + x, X^l + x \rangle}$$

(Del)
$$\frac{\sigma, \Gamma \vdash A : X \quad x \in \mathsf{dom}(\Gamma)}{[x], \Gamma \vdash \texttt{del}\,x : \langle [], [-x], [-x] \rangle}$$

(Seq)
$$\frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y \quad A, B \neq \epsilon}{\sigma_1 \cup (\sigma_2 - X^l), \Gamma \vdash AB : \langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l \rangle}$$

(Choice)
$$\frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y}{\sigma_1 \cup \sigma_2, \Gamma \vdash (A + B) : \langle X^i \cup Y^i, X^o \cup Y^o, X^l \cap Y^l \rangle}$$

(Scope)
$$\frac{[], \Gamma \vdash A : X}{[], \Gamma \vdash \{A\} : \langle X^i, [], [] \rangle}$$

These typing rules deserve some further explanation. The most critical rule is Seq because sequencing two expressions can lead to an increase in instances of the composed expression. First, the semantics of the store in the typing judgment requires that the store always has enough elements for deallocation commands in the expression. So we need to increase the store when the minimum outcome of A and its store, $X^l + \sigma_1$, is not enough for σ_2. In particular, consider a component x, the first premise of the rule Seq tells us that we need a store σ_1 for executing A. Thereafter, we have at least $X^l(x)$ instances of x, where $X^l(x) \in \mathbb{Z}$. Again, by the second premise of the rule Seq, we need $\sigma_2(x)$ instances for safely executing B. Therefore, we must start the execution of AB with at least $(\sigma_2 - X^l)(x)$ in the store (more than $\sigma_2(x)$ if $X^l(x) < 0$). Second, as in the rule Seq in the previous chapter, in the type expression of AB, the maximum of AB is the maximum of A or of the outcome of A together with the maximum of B. The remaining parts, $X^o + Y^o$ and $X^l + Y^l$, are easy referring to the semantics of these parts of the types.

Other typing rules are straightforward. The rule Axiom is used for startup. The rule WeakenB allows us to extend the type environments so that the rules Seq and

Choice may be applied. As we implicitly enlarge the store in the rule Seq and Choice, the rule WeakenS plays a technical role in some proofs and is a natural rule anyway: enlarging the store should preserve typing. The rule New accumulates a new instance in type expression while the rule Del reduces one instance. The first signed multiset in the type of $\texttt{del}\, x$ is empty since it has no effect on the maximum in composition, but the last two multisets are both $[-x]$ since $\texttt{del}\, x$ reduces the local store by one x. The side condition $\sigma, \Gamma \vdash A : X$ in the premise of this rule only guarantees that the basis Γ is legal, and $x \in \mathsf{dom}(\Gamma)$ only guarantees that x has been used somewhere. The rule Scope requires an empty store in the environment because the semantics of deallocation applies to the local store only.

Last, we define the class of *well-typed programs* with respect to the type system. Then we present some typing derivations for the expressions of the program in Section 3.1.1.

Definition 3.2.3 (Well-typed programs). *Program $Prog = Decls; E$ is well-typed if there exist a reordering Γ of declarations in $Decls$ and a type X such that $[], \Gamma \vdash E : X$.*

Using the example in Section 3.1.1 we derive type for $\texttt{new}\, b$. Note that we omit some side conditions as they can be checked easily and we shorten the rule name WeakenS to Wea. The rule Axiom is also simplified.

$$\textsc{Wea}\frac{\textsc{Scope}\dfrac{\textsc{New}\dfrac{\vdash \epsilon : \langle [], [], [] \rangle}{[], d \mathrel{-\!\!\prec} \epsilon \vdash \texttt{new}\, d : \langle [d], [d], [d] \rangle}}{[], d \mathrel{-\!\!\prec} \epsilon \vdash \{\texttt{new}\, d\} : \langle [d], [], [] \rangle} \quad \textsc{Wea}\dfrac{\vdash \epsilon : \langle [], [], [] \rangle \quad \vdash \epsilon : \langle [], [], [] \rangle}{[], d \mathrel{-\!\!\prec} \epsilon \vdash \epsilon : \langle [], [], [] \rangle}}{[], d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon \vdash \{\texttt{new}\, d\} : \langle [d], [], [] \rangle} \tag{3.1}$$

$$\textsc{New}\frac{\textsc{Seq}\dfrac{\textsc{New}\dfrac{\textsc{Wea}\dfrac{\vdash \epsilon : \langle [], [], [] \rangle \quad \vdash \epsilon : \langle [], [], [] \rangle}{[], d \mathrel{-\!\!\prec} \epsilon \vdash \epsilon : \langle [], [], [] \rangle}}{[], d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon \vdash \texttt{new}\, e : \langle [e], [e], [e] \rangle} \quad (3.1)}{[], d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon \vdash \texttt{new}\, e \{\texttt{new}\, d\} : \langle [d, e], [e], [e] \rangle}}{[], d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} \texttt{new}\, e \{\texttt{new}\, d\} \vdash \texttt{new}\, a : \langle [a, d, e], [a, e], [a, e] \rangle} \tag{3.2}$$

Similarly, we can derive $[], \Gamma_1 \vdash \texttt{new}\, e\, \texttt{new}\, d : \langle [d, e], [d, e], [d, e] \rangle$ where $\Gamma_1 = d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} \texttt{new}\, e \{\texttt{new}\, d\}$.

$$\textsc{Choice}\frac{(3.2) \quad [], \Gamma_1 \vdash \texttt{new}\, e\, \texttt{new}\, d : \langle [d, e], [d, e], [d, e] \rangle}{[], \Gamma_1 \vdash (\texttt{new}\, a + \texttt{new}\, e\, \texttt{new}\, d) : \langle [a, d, e], [a, d, e], [e] \rangle} \tag{3.3}$$

$$\textsc{New}\frac{\textsc{Seq}\dfrac{(3.3) \quad \textsc{Del}\dfrac{(3.3) \quad e \in \mathsf{dom}(d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon)}{[e], \Gamma_1 \vdash \texttt{del}\, e : \langle [], [-e], [-e] \rangle}}{[], \Gamma_1 \vdash (\texttt{new}\, a + \texttt{new}\, e\, \texttt{new}\, d)\, \texttt{del}\, e : \langle [a, d, e], [a, d], [] \rangle}}{[], \Gamma_1, b \mathrel{-\!\!\prec} (\texttt{new}\, a + \texttt{new}\, e\, \texttt{new}\, d)\, \texttt{del}\, e \vdash \texttt{new}\, b : \langle [a, b, d, e], [a, b, d], [b] \rangle}$$

3.3 Properties

As in Chapter 2, this section first presents the type soundness property without proofs. Then we prove some technical lemmas and the type soundness. Last, we discuss the termination of well-typed programs and the type inference algorithm.

3.3.1 Type Soundness

In this model, type errors occur when a program tries to delete an instance of a component x but the local store has no x, or when it tries to instantiate a component x but there is no declaration of x. We will prove that these two situations will not happen. Besides, we will prove an additional important property which guarantees that a well-typed program

will not create more instances than a certain maximum stated in the type of its main expression.

As in Chapter 2, for the type soundness, we will prove two main lemmas: Preservation and Progress. Before that we define some auxiliary notions and definitions.

We repeat some notations for a stack $\mathbf{S}$: $\mathsf{hi}(\mathbf{S})$ denotes the height of the stack $\mathbf{S}$, $\mathbf{S}(k)$ denotes the pair at position k from the bottom of the stack, $[\mathbf{S}(k)]$ denotes the local store at position k, $[\mathbf{S}]$ denotes the multiset of all active instances of $\mathbf{S}$, and $\mathbf{S}|_k$ denotes the stack from the bottom of $\mathbf{S}$ up to k element.

Definition 3.3.1 (Well-typed configurations). *Configuration $\mathbf{S}$ is well-typed with respect to a basis Γ, notation $\Gamma \models \mathbf{S}$, if for all $1 \leq k \leq \mathsf{hi}(\mathbf{S})$ such that $\mathbf{S}(k) = (M, E)$, there exists X such that*

$$M, \Gamma \vdash E : X$$

Definition 3.3.2 (Terminal configurations). *A configuration $\mathbf{S}$ is terminal if it has the form (M, ϵ).*

Definition 3.3.3 (Stuck states). *A configuration $\mathbf{S}$ is stuck if no transition rule applies and $\mathbf{S}$ is not terminal.*

Lemma 3.3.4 (Preservation). *If $\Gamma \models \mathbf{S}$ and $\mathbf{S} \longrightarrow \mathbf{S}'$, then $\Gamma \models \mathbf{S}'$.*

Lemma 3.3.5 (Progress). *If $\Gamma \models \mathbf{S}$, then either $\mathbf{S}$ is terminal or there exists a configuration $\mathbf{S}'$ such that $\mathbf{S} \longrightarrow \mathbf{S}'$.*

Next, we show an additional invariant which allows us to infer the resource bounds of well-typed programs. The invariant is about the monotonicity of the maximum number of instances that a well-typed configuration can reach. We calculate the maximum number as follows.

$$\mathsf{maxins}(\mathbf{S}) = \bigcup_{k=1}^{\mathsf{hi}(\mathbf{S})} ([\mathbf{S}|_k] + X_k^i)$$

where X_k is the type of the expression at position k. During transition, this maximum number of instances does not increase.

Lemma 3.3.6 (Invariant of maxins). *If $\Gamma \models \mathbf{S}$ and $\mathbf{S} \longrightarrow \mathbf{S}'$, then*

$$\mathsf{maxins}(\mathbf{S}) \supseteq \mathsf{maxins}(\mathbf{S}')$$

Now we can state the type soundness together with the upper bound of instances that a well-typed program always respects.

Theorem 3.3.7 (Soundness). *Let $Prog = Decls; E$ be well-typed, that is, $\Gamma \vdash E : X$ for some reordering Γ of Decls and some type X. Then for any $\mathbf{S}$ such that $([], E) \longrightarrow^* \mathbf{S}$ we have that $\mathbf{S}$ is not stuck and $[\mathbf{S}] \subseteq X^i$.*

3.3.2 Typing Properties

This section lists some typing properties of the type system needed to prove the lemmas and theorem in the previous section. These properties are analogous to those in the previous chapter. First, we update some terminology on bases and the function var.

Definition 3.3.8 (Bases). *Let $\Gamma = x_1 \prec A_1, \ldots, x_n \prec A_n$ be a basis.*

- *Γ is called* legal *if $\sigma, \Gamma \vdash A : X$ for some store σ, expression A and type X.*
- *A declaration $x \prec A$* is in *Γ, notation $x \prec A \in \Gamma$, if $x \equiv x_i$ and $A \equiv A_i$ for some i.*
- *Δ is an* initial segment *of Γ, if $\Delta = x_1 \prec A_1, \ldots, x_j \prec A_j$ for some $1 \leq j \leq n$.*

The function var is extended with the new form of expressions:

$$\begin{aligned}&\mathsf{var}(\epsilon) = \emptyset,\\ &\mathsf{var}(\,\mathtt{new}\,x) = \mathsf{var}(\,\mathtt{del}\,x) = \{x\}, \quad \mathsf{var}(\{A\}) = \mathsf{var}(A),\\ &\mathsf{var}(AB) = \mathsf{var}((A+B)) = \mathsf{var}(A) \cup \mathsf{var}(B)\end{aligned}$$

We use X^* for any of X^i, X^o and X^l.

Lemma 3.3.9 (Valid typing judgment). *If* $\sigma, \Gamma \vdash A\!:\!X$*, then*

1. $\mathsf{var}(A) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{dom}(X^*) \subseteq \mathsf{dom}(\Gamma)$,
2. *every variable in* $\mathsf{dom}(\Gamma)$ *is declared only once in* Γ,
3. $X^i \supseteq X^o \supseteq X^l$, $X^i \supseteq []$,
4. $\sigma + X^* \supseteq []$.

Proof. By induction on typing derivations.

- Base case Axiom:

$$\text{(Axiom)}\quad \frac{}{[], \varnothing \vdash \epsilon\!:\!\langle [], [], [] \rangle}$$

Then $\mathsf{var}(\epsilon) = \mathsf{dom}(X^*) = \mathsf{dom}(\varnothing) = \emptyset$ and all the clauses are trivial.

- Case WeakenB:

$$\text{(WeakenB)}\quad \frac{\sigma, \Gamma' \vdash A\!:\!X \quad \sigma_1, \Gamma' \vdash B\!:\!Y \quad x \notin \mathsf{dom}(\Gamma')}{\sigma, \Gamma', x{\prec} B \vdash A\!:\!X}$$

Clause 1 holds by the induction hypothesis and $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma', x{\prec} B) \supset \mathsf{dom}(\Gamma')$. Clause 2 holds by the side condition. Clauses 3 and 4 hold by the induction hypothesis.

- Case WeakenS:

$$\text{(WeakenS)}\quad \frac{\sigma_1, \Gamma \vdash A\!:\!X \quad \sigma_1 \subseteq \sigma}{\sigma, \Gamma \vdash A\!:\!X}$$

Clauses 1, 2 and 3 hold by the induction hypothesis. Clause 4 holds by the side condition.

- Case New:

$$\text{(New)}\quad \frac{\sigma, \Gamma' \vdash B\!:\!Y \quad x \notin \mathsf{dom}(\Gamma')}{\sigma, \Gamma', x{\prec} B \vdash \,\mathtt{new}\,x\!:\!\langle Y^i + x, Y^o + x, Y^l + x \rangle}$$

Clause 1 holds by the induction hypothesis and $x \in \mathsf{var}(\,\mathtt{new}\,x)$, $x \in X^*$, and $x \in \mathsf{dom}(\Gamma)$. Clause 2 holds by the side condition. Clause 3 and 4 hold by the induction hypothesis and $X^*(x) = 1$.

- Case Del:

$$\text{(Del)}\quad \frac{\sigma_1, \Gamma \vdash B\!:\!Y \quad x \in \mathsf{dom}(\Gamma)}{[x], \Gamma \vdash \,\mathtt{del}\,x\!:\!\langle [], [-x], [-x] \rangle}$$

Clause 1 holds by $x \in \mathsf{dom}(\Gamma)$. Clause 2 holds by the induction hypothesis. Clause 3 follows easily by the type expression $X = \langle [], [-x], [-x] \rangle$. Clause 4 holds by $\sigma(x) = 1$ and $X^*(x) \geq -1$.

- Case SEQ:

$$\text{(SEQ)}\ \frac{\sigma_1, \Gamma \vdash B : Y \quad \sigma_2, \Gamma \vdash C : Z \quad B, C \neq \epsilon}{\sigma_1 \cup (\sigma_2 - Y^l), \Gamma \vdash BC : \langle Y^i \cup (Y^o + Z^i), Y^o + Z^o, Y^l + Z^l \rangle}$$

Clauses 1 holds by the induction hypothesis and $\mathsf{var}(BC) = \mathsf{var}(B) \cup \mathsf{var}(C)$. Clause 2 holds by the induction hypothesis. Clause 3 also follows easily by the induction hypothesis. For clause 4, by clause 3 we only need to prove that $\sigma + X^l \supseteq []$. We have

$$\begin{aligned}
\sigma + X^l &= (\sigma_1 \cup (\sigma_2 - Y^l)) + X^l \\
&\supseteq (\sigma_2 - Y^l) + X^l \\
&= (\sigma_2 - Y^l) + (Y^l + Z^l) && (X^l = Y^l + Z^l) \\
&= \sigma_2 - Y^l + Y^l + Z^l \\
&= \sigma_2 + Z^l \\
&\supseteq [] && \text{(IH)}
\end{aligned}$$

- Case CHOICE:

$$\text{(CHOICE)}\ \frac{\sigma_1, \Gamma \vdash B : Y \quad \sigma_2, \Gamma \vdash C : Z}{\sigma_1 \cup \sigma_2, \Gamma \vdash (B + C) : \langle Y^i \cup Z^i, Y^o \cup Z^o, Y^l \cap Z^l \rangle}$$

All clauses follow easily by the induction hypothesis.

- Case SCOPE:

$$\text{(SCOPE)}\ \frac{[], \Gamma \vdash B : Y}{[], \Gamma \vdash \{B\} : \langle Y^i, [], [] \rangle}$$

All clauses follow easily by the induction hypothesis.

□

Lemma 3.3.10 (Associativity). *If $\sigma_i, \Gamma \vdash A_i : X_i$, for $i \in \{1, 2, 3\}$, then the typing judgments for $(A_1 A_2) A_3$ and $A_1 (A_2 A_3)$ are the same.*

Proof.

$$\text{(SEQ)}\ \frac{\sigma_1, \Gamma \vdash A_1 : X_1 \quad \sigma_2, \Gamma \vdash A_2 : X_2 \quad A_1, A_2 \neq \epsilon}{\sigma_1 \cup (\sigma_2 - X_1^l), \Gamma \vdash A_1 A_2 : \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o, X_1^l + X_2^l \rangle}$$

By the rule SEQ, we have $\sigma_Y, \Gamma \vdash A_1 A_2 : Y$ with $\sigma_Y = \sigma_1 \cup (\sigma_2 - X_1^l)$ and

$$Y = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o, X_1^l + X_2^l \rangle$$

Similarly, we have $\sigma_Z, \Gamma \vdash A_2 A_3 : Z$ with $\sigma_Z = \sigma_2 \cup (\sigma_3 - X_2^l)$ and

$$Z = \langle X_2^i \cup (X_2^o + X_3^i), X_2^o + X_3^o, X_2^l + X_3^l \rangle$$

Continue to apply the rule SEQ, we get the typing judgments for $(A_1 A_2) A_3$ and $A_1 (A_2 A_3)$:

$$\sigma_Y \cup (\sigma_3 - Y^l), \Gamma \vdash (A_1 A_2) A_3 : \langle Y^i \cup (Y^o + X_3^i), Y^o + X_3^o, Y^l + X_3^l \rangle$$

$$\sigma_1 \cup (\sigma_Z - X_1^l), \Gamma \vdash A_1 (A_2 A_3) : \langle X_1^i \cup (X_1^o + Z^i), X_1^o + Z^o, X_1^l + Z^l \rangle$$

Then to prove that the two judgments are the same, we need to prove the following equations:

$$\sigma_Y \cup (\sigma_3 - Y^l) = \sigma_1 \cup (\sigma_Z - X_1^l)$$
$$Y^i \cup (Y^o + X_3^i) = X_1^i \cup (X_1^o + Z^i)$$
$$Y^o + X_3^o = X_1^o + Z^o$$
$$Y^l + X_3^l = X_1^l + Z^l$$

The first equation is proved as follows.

$$\begin{aligned}
&\sigma_Y \cup (\sigma_3 - Y^l) \\
&= \sigma_1 \cup (\sigma_2 - X_1^l) \cup (\sigma_3 - Y^l) && (\sigma_Y = \sigma_1 \cup (\sigma_2 - X_1^l)) \\
&= \sigma_1 \cup (\sigma_2 - X_1^l) \cup (\sigma_3 - X_1^l - X_2^l) && (Y^l = X_1^l + X_2^l) \\
&= \sigma_1 \cup ((\sigma_2 \cup (\sigma_3 - X_2^l)) - X_1^l) \\
&= \sigma_1 \cup (\sigma_Z - X_1^l) && (\sigma_Z = \sigma_2 \cup (\sigma_3 - X_2^l))
\end{aligned}$$

The second equation is also straightforward.

$$\begin{aligned}
&Y^i \cup (Y^o + X_3^i) \\
&= X_1^i \cup (X_1^o + X_2^i) \cup (Y^o + X_3^i) && (Y^i = X_1^i \cup (X_1^o + X_2^i)) \\
&= X_1^i \cup (X_1^o + X_2^i) \cup (X_1^o + X_2^o + X_3^i) && (Y^o = X_1^o + X_2^o) \\
&= X_1^i \cup (X_1^o + (X_2^i \cup (X_2^o + X_3^i))) \\
&= X_1^i \cup (X_1^o + Z^i) && (Z^i = X_2^i \cup (X_2^o + X_3^i))
\end{aligned}$$

The last two equations are easy. □

Lemma 3.3.11 (Generation).

1. *If $\sigma, \Gamma \vdash \texttt{new}\, x : X$, then there exist Δ, Δ', A and Y such that $\Gamma = \Delta, x \prec A, \Delta'$ and $\sigma, \Delta \vdash A : Y$ and $X = \langle Y^i + x, Y^o + x, Y^l + x \rangle$.*
2. *If $\sigma, \Gamma \vdash \texttt{del}\, x : X$, then $x \in \sigma$, $x \in \mathsf{dom}(\Gamma)$ and $X = \langle [], [-x], [-x] \rangle$.*
3. *If $\sigma, \Gamma \vdash AB : Z$ with $A, B \neq \epsilon$, then there exist X, Y such that $\sigma, \Gamma \vdash A : X$, $\sigma + X^l, \Gamma \vdash B : Y$ and $Z = \langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l \rangle$.*
4. *If $\sigma, \Gamma \vdash (A + B) : Z$, then there exist X, Y such that $\sigma, \Gamma \vdash A : X$ and $\sigma, \Gamma \vdash B : Y$ and $Z = \langle X^i \cup Y^i, X^o \cup Y^o, X^l \cap Y^l \rangle$.*
5. *If $\sigma, \Gamma \vdash \{A\} : Z$, then there exists X such that $[], \Gamma \vdash A : X$ and $Z = \langle X^i, [], [] \rangle$.*

Proof. By induction on typing derivations.

1. $\sigma, \Gamma \vdash \texttt{new}\, x : X$ can only be derived by the rule New or WeakenB or WeakenS. If it is derived by the rule New, then there is only one possibility:

$$\text{(New)}\ \frac{\sigma, \Delta \vdash B : Y \quad x \notin \mathsf{dom}(\Delta)}{\sigma, \Delta, x \prec B \vdash \texttt{new}\, x : X}$$

   with $X = \langle Y^i + x, Y^o + x, Y^l + x \rangle$ and $\Gamma = \Delta, x \prec B$, so that Δ' is empty.

   If $\sigma, \Gamma \vdash \texttt{new}\, x : X$ is derived by the rule WeakenB:

$$\text{(WeakenB)}\ \frac{\sigma, \Gamma' \vdash \texttt{new}\, x : X \quad \sigma_1, \Gamma' \vdash B : Y \quad y \notin \mathsf{dom}(\Gamma)}{\sigma, \Gamma', y \prec B \vdash \texttt{new}\, x : X}$$

then $\sigma, \Gamma' \vdash \texttt{new}\, x : X$ and by the induction hypothesis applied to $\sigma, \Gamma' \vdash \texttt{new}\, x : X$, we have $\Gamma' = \Delta_1, x \prec A, \Delta_2$ and $\sigma, \Delta_1 \vdash A : Y$ for some Δ_1, Δ_2, A, and $X = \langle Y^i + x, Y^o + x, Y^l + x \rangle$. Take $\Delta = \Delta_1$, $\Delta' = \Delta_2, y \prec B$, we have all the conclusions.

If it is derived by the rule WeakenS:

$$\text{(WeakenS)}\quad \frac{\sigma_1, \Gamma \vdash \texttt{new}\, x : X \quad \sigma_1 \subseteq \sigma}{\sigma, \Gamma \vdash \texttt{new}\, x : X}$$

the clause holds by the induction hypothesis and WeakenS.

2. $\sigma, \Gamma \vdash \texttt{del}\, x : X$ can only be derived by the rule Del or WeakenB or WeakenS. If it is derived by the rule Del, then there is only one possibility:

$$\text{(Del)}\quad \frac{\sigma, \Gamma \vdash A : X \quad x \in \mathsf{dom}(\Gamma)}{[x], \Gamma \vdash \texttt{del}\, x : \langle [], [-x], [-x] \rangle}$$

and the conclusions hold easily.

If $\sigma, \Gamma \vdash \texttt{del}\, x : X$ is derived by the rule WeakenB:

$$\text{(WeakenB)}\quad \frac{\sigma, \Gamma' \vdash \texttt{del}\, x : X \quad \sigma_2, \Gamma' \vdash B : Y \quad y \notin \mathsf{dom}(\Gamma')}{\sigma, \Gamma', y \prec B \vdash \texttt{del}\, x : X}$$

then $\sigma, \Gamma' \vdash \texttt{del}\, x : X$ and by the induction hypothesis applied to $\sigma, \Gamma' \vdash \texttt{del}\, x : X$, we get all the conclusions.

If it is derived by the rule WeakenS, the clause holds by the induction hypothesis.

3. $\sigma, \Gamma \vdash AB : Z$ with $A, B \neq \epsilon$ can only be derived by the rule Seq or WeakenB or WeakenS. If $\sigma, \Gamma \vdash AB : Z$ is derived by the rule Seq with two component expressions A and B in the premise of the typing rule:

$$\text{(Seq)}\quad \frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y \quad A, B \neq \epsilon}{\sigma_1 \cup (\sigma_2 - X^l), \Gamma \vdash AB : \langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l \rangle}$$

then $\sigma = \sigma_1 \cup (\sigma_2 - X^l)$. We need to prove that $\sigma, \Gamma \vdash A : X$ and $\sigma + X^l, \Gamma \vdash B : Y$. Since $\sigma \supseteq \sigma_1$, the first one holds by applying the rule WeakenS to the induction hypothesis $\sigma_1, \Gamma \vdash A : X$. Similarly, since $\sigma \supseteq \sigma_2 - X^l$ implies $\sigma + X^l \supseteq \sigma_2$, the second one holds by applying the rule WeakenS to the induction hypothesis $\sigma_2, \Gamma \vdash B : Y$.

If $\sigma, \Gamma \vdash AB : Z$ is derived by the rule Seq with two component expressions $A_1 \neq A$ and $B_1 \neq B$ such that $A_1 B_1 = AB$:

$$\text{(Seq)}\quad \frac{\sigma_1, \Gamma \vdash A_1 : X_1 \quad \sigma_2, \Gamma \vdash B_1 : Y_1 \quad A_1, B_1 \neq \epsilon}{\sigma_1 \cup (\sigma_2 - X_1^l), \Gamma \vdash A_1 B_1 : \langle X_1^i \cup (X_1^o + Y_1^i), X_1^o + Y_1^o, X_1^l + Y_1^l \rangle}$$

then $\sigma = \sigma_1 \cup (\sigma_2 - X_1^l)$, $Z = \langle X_1^i \cup (X_1^o + Y_1^i), X_1^o + Y_1^o, X_1^l + Y_1^l \rangle$. There are two possibilities:

- $A = A_1 A_2$ for some $A_2 \neq \epsilon$: then $B_1 = A_2 B$ and we have $\sigma_2, \Gamma \vdash A_2 B : Y_1$.

  By the induction hypothesis applied to $\sigma_2, \Gamma \vdash A_2 B : Y_1$, we get $\sigma_2, \Gamma \vdash A_2 : X_2$ and $\sigma_2 + X_2^l, \Gamma \vdash B : Y$ and

$$Y_1 = \langle X_2^i \cup (X_2^o + Y^i), X_2^o + Y^o, X_2^l + Y^l \rangle$$

Now we can apply the rule Seq to $\sigma_1, \Gamma \vdash A_1 : X_1$ and $\sigma_2, \Gamma \vdash A_2 : X_2$:

$$\text{(Seq)} \quad \frac{\sigma_1, \Gamma \vdash A_1 : X_1 \quad \sigma_2, \Gamma \vdash A_2 : X_2 \quad A_1, A_2 \neq \epsilon}{\sigma_1 \cup (\sigma_2 - X_1^l), \Gamma \vdash A_1 A_2 : \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o, X_1^l + X_2^l \rangle}$$

and we get $\sigma, \Gamma \vdash A : X$ with

$$X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o, X_1^l + X_2^l \rangle$$

Continue applying the rule Seq to $\sigma, \Gamma \vdash A : X$ and $\sigma_2 + X_2^l, \Gamma \vdash B : Y$, we get

$$\sigma \cup (\sigma_2 + X_2^l - X^l), \Gamma \vdash AB : \langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l \rangle$$

We have $\sigma \cup (\sigma_2 + X_2^l - X^l) = \sigma \cup (\sigma_2 - X_1^l) = \sigma$. So we only need to show that $Z = \langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l \rangle$. This holds by Lemma 3.3.10.

- $B = B_0 B_1$: analogous to the previous subcase.

If $\sigma, \Gamma \vdash AB : Z$ is derived by the rule WeakenB:

$$\text{(WeakenB)} \quad \frac{\sigma, \Gamma' \vdash AB : Z \quad \sigma_2, \Gamma' \vdash C : V \quad y \notin \mathsf{dom}(\Gamma')}{\sigma, \Gamma', y \prec C \vdash AB : Z}$$

with $\Gamma = \Gamma', y \prec C$ then by the induction hypothesis applied to $\sigma, \Gamma' \vdash AB : Z$, we have $\sigma, \Gamma' \vdash A : X$, $\sigma + X^l, \Gamma' \vdash B : Y$ and $Z = \langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l \rangle$. Now applying the rule WeakenB to $\sigma, \Gamma' \vdash A : X$ and $\sigma + X^l, \Gamma' \vdash B : Y$ with $\sigma_2, \Gamma' \vdash C : V$, we get the conclusions.

If it is derived by the rule WeakenS, the proof is analogous to the previous subcase.

4. $\sigma, \Gamma \vdash (A + B) : Z$ can only be derived by the rule Choice or WeakenB or WeakenS.

If it is derived by the rule Choice, then there is only one possibility:

$$\text{(Choice)} \quad \frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y}{\sigma_1 \cup \sigma_2, \Gamma \vdash (A + B) : \langle X^i \cup Y^i, X^o \cup Y^o, X^l \cap Y^l \rangle}$$

with $\sigma = \sigma_1 \cup \sigma_2$ and $Z = \langle X^i \cup Y^i, X^o \cup Y^o, X^l \cap Y^l \rangle$. By the induction hypothesis, we have $\sigma_1, \Gamma \vdash A : X$ and $\sigma_2, \Gamma \vdash B : Y$. Since $\sigma_1 \subseteq \sigma$ and $\sigma_2 \subseteq \sigma$, we can apply the rule WeakenS and get the conclusions.

If $\sigma, \Gamma \vdash (A + B) : Z$ is derived by the rule WeakenB:

$$\text{(WeakenB)} \quad \frac{\sigma, \Gamma' \vdash (A + B) : Z \quad \sigma_2, \Gamma' \vdash E : V \quad x \notin \mathsf{dom}(\Gamma')}{\sigma, \Gamma', x \prec E \vdash (A + B) : Z}$$

then we apply the induction hypothesis to $\sigma_1, \Gamma' \vdash (A + B) : Z$ and then WeakenB.

If it is derived by the rule WeakenS, the proof is analogous to the previous subcase.

5. $\sigma, \Gamma \vdash \{A\} : Z$ can only be derived by the rule Scope or WeakenB or WeakenS.

If it is derived by the rule Scope, then there is only one possibility:

$$\text{(Scope)} \quad \frac{[], \Gamma \vdash A : X}{[], \Gamma \vdash \{A\} : \langle X^i, [], [] \rangle}$$

with $Z = \langle X^i, [], [] \rangle$. The conclusion follows immediately.

If $\sigma, \Gamma \vdash \{A\} : Z$ is derived by the rule WEAKENB:

$$\frac{\sigma, \Gamma' \vdash \{A\} : X \quad \sigma, \Gamma' \vdash E : V \quad x \notin \mathsf{dom}(\Gamma')}{\sigma, \Gamma', x \prec E \vdash \{A\} : X} \ (\text{WEAKENB})$$

then the proof is analogous to the subcase WEAKENB of case 4.

If it is derived by the rule WEAKENS, the proof is analogous to the subcase WEAKENS of case 4.

□

Lemma 3.3.12 (Weakening).

1. *If $\Gamma = \Delta, x \prec E, \Delta'$ is legal, then $\sigma, \Delta \vdash E : Z$ for some Z, σ.*
2. *If $\sigma, \Gamma \vdash E : Z$ and Γ is an initial segment of a legal basis Γ', then $\sigma, \Gamma' \vdash E : Z$.*

Proof. 1. Since Γ is legal, by Definition 3.3.8, there exist σ', B, Y such that $\sigma', \Gamma \vdash B : Y$. In the typing derivation tree for B, the only way to extend Δ to $\Delta, x \prec E$ is by applying the rule NEW, or WEAKENB.

$$\frac{\sigma, \Delta \vdash E : Z \quad x \notin \mathsf{dom}(\Delta)}{\sigma, \Delta, x \prec E \vdash \mathtt{new}\, x : \langle Z^i + x, Z^o + x, Z^l + x \rangle} \ (\text{NEW})$$

$$\frac{\sigma_1, \Delta \vdash A : X \quad \sigma, \Delta \vdash E : Z \quad x \notin \mathsf{dom}(\Delta)}{\sigma_1, \Delta, x \prec E \vdash A : X} \ (\text{WEAKENB})$$

Each of the rules has $\sigma, \Delta \vdash E : Z$ as a premise.

2. Since Γ is an initial segment of Γ', suppose that $\Gamma' = \Gamma, x_1 \prec A_1, \ldots, x_n \prec A_n$. By clause 1 we have for all $k = 1..n$, there exist σ_{k-1}, X_k such that

$$\sigma_{k-1}, \Gamma, x_1 \prec A_1, \ldots, x_{k-1} \prec A_{k-1} \vdash A_k : X_k$$

By applying the rule WEAKENB for $k = 1$, we get

$$\frac{\sigma, \Gamma \vdash E : Z \quad \sigma_0, \Gamma \vdash A_1 : X_1 \quad x_1 \notin \mathsf{dom}(\Gamma)}{\sigma, \Gamma, x_1 \prec A_1 \vdash E : Z} \ (\text{WEAKENB})$$

Reiterate applying the rule WEAKENB for $k = 2, \ldots, n$, we get the conclusion.

□

Lemma 3.3.13 (Strengthening). *If $\sigma, \Gamma, x \prec A \vdash B : Y$ and $x \notin \mathsf{var}(B)$, then $\sigma, \Gamma \vdash B : Y$ and $x \notin Y^i$.*

Proof. By induction on typing derivations. Let $\Gamma' = \Gamma, x \prec A$.

- Case AXIOM: $B = \epsilon$, does not apply since the basis is not empty.
- Case NEW: $B = \mathtt{new}\, x$, does not apply since $\mathsf{var}(B) = \mathsf{var}(\mathtt{new}\, x) = \{x\}$.
- Case DEL: if $B = \mathtt{del}\, x$, then the rule does not apply since $\mathsf{var}(B) = \mathsf{var}(\mathtt{del}\, x) = \{x\}$. If $B = \mathtt{del}\, y$ (and $y \neq x$):

$$\frac{\sigma_1, \Gamma, x \prec A \vdash C : Z \quad y \in \mathsf{dom}(\Gamma) \cup \{x\}}{[y], \Gamma, x \prec A \vdash \mathtt{del}\, y : \langle [], [-y], [-y] \rangle} \ (\text{DEL})$$

then $\sigma = [y]$. By Lemma 3.3.12, clause 1 applied to the legal basis $\Gamma, x \prec A$ in the premise, we get $\sigma_2, \Gamma \vdash A : X$ for some σ_2 and X. Moreover, $y \neq x$ so we have $y \in \mathsf{dom}(\Gamma)$ and then we can apply the rule Del and WeakenS to get the conclusion:

$$\text{(Del)}\ \frac{\sigma_2, \Gamma \vdash A : X \quad y \in \mathsf{dom}(\Gamma)}{[y], \Gamma \vdash \mathtt{del}\, y : \langle [], [-y], [-y] \rangle}$$

- Case WeakenB:

$$\text{(WeakenB)}\ \frac{\sigma, \Gamma \vdash B : Y \quad \sigma_1, \Gamma \vdash A : X \quad x \notin \mathsf{dom}(\Gamma)}{\sigma, \Gamma, x \prec A \vdash B : Y}$$

  We have $\sigma, \Gamma \vdash B : Y$ in the premise. By Lemma 3.3.9, $\mathsf{dom}(Y^*) \subseteq \mathsf{dom}(\Gamma, x \prec A)$ and $x \notin \mathsf{dom}(\Gamma)$, we get $x \notin Y^i$.

- Case WeakenS:

$$\text{(WeakenS)}\ \frac{\sigma_1, \Gamma' \vdash B : Y \quad \sigma_1 \subseteq \sigma}{\sigma, \Gamma' \vdash B : Y}$$

  By the induction hypothesis, we have $\sigma_1, \Gamma \vdash B : Y$. By the rule WeakenS applied to $\sigma_1, \Gamma \vdash B : Y$, we get the conclusion $\sigma, \Gamma \vdash B : Y$.

- Case Seq:

$$\text{(Seq)}\ \frac{\sigma_1, \Gamma' \vdash B_1 : Y_1 \quad \sigma_2, \Gamma' \vdash B_2 : Y_2 \quad B_1, B_2 \neq \epsilon}{\sigma_1 \cup (\sigma_2 - Y_1^l), \Gamma' \vdash B_1 B_2 : \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o, Y_1^l + Y_2^l \rangle}$$

  with $Y = \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o, Y_1^l + Y_2^l \rangle$, $\sigma = \sigma_1 \cup (\sigma_2 - X^l)$, and $B = B_1 B_2$. Since $x \notin \mathsf{var}(B_1 B_2) = \mathsf{var}(B_1) \cup \mathsf{var}(B_2)$, we have $x \neq \mathsf{var}(B_1)$ and $x \notin \mathsf{var}(B_2)$. By the induction hypothesis, we get $\sigma_1, \Gamma \vdash B_1 : Y_1$ and $x \notin Y_1^i$ and $\sigma_2, \Gamma \vdash B_2 : Y_2$ and $x \notin Y_2^i$. Now we can apply the rule Seq to get the conclusion.

- Case Choice, $B = (B_1 + B_2)$: analogous to the case Seq.
- Case Scope, $B = \{C\}$: analogous to the case Seq.

□

Proposition 3.3.14 (Uniqueness of types). *If $\sigma, \Gamma \vdash A : X$ and $\sigma, \Gamma \vdash A : Y$, then $X^i = Y^i$, $X^o = Y^o$, and $X^l = Y^l$.*

Proof. By induction on typing derivations.

- Base case Axiom: We have $A = \epsilon$ and Γ is empty, so that only Axiom is applicable. Hence, $X = Y = \langle [], [], [] \rangle$.

- Case New:

$$\text{(New)}\ \frac{\sigma, \Gamma' \vdash B : U \quad x \notin \mathsf{dom}(\Gamma)}{\sigma, \Gamma', x \prec B \vdash \mathtt{new}\, x : X}$$

  with $X = \langle U^i + x, U^o + x, U^l + x \rangle$ and $\Gamma = \Gamma', x \prec B$.

  Assume Proposition 3.3.14 holds for the premise of this rule and let $\sigma, \Gamma \vdash \mathtt{new}\, x : Y$. By Generation Lemma 3.3.11, clause 1, we get $\Gamma = \Delta_1, x \prec C, \Delta_2$ and $\sigma, \Delta_1 \vdash C : V$ for some Δ_1, Δ_2, C, and $Y = \langle V^i + x, V^o + x, V^l + x \rangle$.

  By Lemma 3.3.9, there is only one declaration of x in Γ. This means $\Delta_1 = \Gamma'$, $C = B$ and Δ_2 is empty, so $\sigma, \Gamma' \vdash B : V$. By the induction hypothesis we have $U^* = V^*$, thus $X^* = Y^*$.

- Case Del: We have $A = \mathtt{del}\,x$. The clause holds immediately since $X = Y = \langle [], [-x], [-x] \rangle$.

- Case WeakenB:

$$\frac{\sigma, \Gamma' \vdash A : X \quad \sigma_1, \Gamma' \vdash B : Z \quad x \notin \mathsf{dom}(\Gamma')}{\sigma, \Gamma', x \prec B \vdash A : X}\ (\text{WeakenB})$$

  with $\Gamma = \Gamma', x \prec B$.

  Assume Proposition 3.3.14 holds for the two premises and let $\sigma, \Gamma \vdash A : Y$. Since $\sigma, \Gamma' \vdash A : X$ and $x \notin \mathsf{dom}(\Gamma')$, we have $x \notin \mathsf{var}(A)$ by Lemma 3.3.9. In addition, by Lemma 3.3.13 applied to $\sigma, \Gamma', x \prec B \vdash A : Y$, we get $\sigma, \Gamma' \vdash A : Y$. Now by the induction hypothesis, we have the conclusion $X = Y$.

- Case Seq:

$$\frac{\sigma_1, \Gamma \vdash B_1 : Y_1 \quad \sigma_2, \Gamma \vdash B_2 : Y_2 \quad B_1, B_2 \neq \epsilon}{\sigma_1 \cup (\sigma_2 - Y_1^l), \Gamma \vdash B_1 B_2 : \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o, Y_1^l + Y_2^l \rangle}\ (\text{Seq})$$

  with $A = B_1 B_2$, $\sigma = \sigma_1 \cup (\sigma_2 - X^l)$ and $X = \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o, Y_1^l + Y_2^l \rangle$.

  First, by the rule WeakenS applied to the premises of the rule, we get $\sigma, \Gamma \vdash B_1 : Y_1$, $\sigma, \Gamma \vdash B_2 : Y_2$. Second, by Generation Lemma 3.3.11, clause 3 applied to $\sigma, \Gamma \vdash B_1 B_2 : Y$, we get $\sigma, \Gamma \vdash B_1 : V_1$, $\sigma, \Gamma \vdash B_2 : V_2$ and $Y = \langle V_1^i \cup (V_1^o + V_2^i), V_1^o + V_2^o, V_1^l + V_2^l \rangle$. Now by the induction hypothesis, we have $Y_1 = V_1$ and $Y_2 = V_2$. Hence, we get $X = Y = \langle Y_1^i \cup (Y_1^o + Y_2^i), Y_1^o + Y_2^o, Y_1^l + Y_2^l \rangle$.

- Case Choice: analogous to the case Seq.

- Case Scope: analogous to the case Seq.

□

3.3.3 Soundness Proofs

Proof of Lemma 3.3.4 (Preservation). *If* $\Gamma \models \mathbf{S}$ *and* $\mathbf{S} \longrightarrow \mathbf{S}'$, *then* $\Gamma \models \mathbf{S}'$.

Proof. By Definition 3.3.1 of well-typed configurations, we need to prove that for all pairs (M, E) in $\mathbf{S}'$, there exists X such that $M, \Gamma \vdash E : X$.

The proof proceeds by case analysis on the transition relation $\longrightarrow$. In each case, we only need to prove for positions k in $\mathbf{S}'$ but not in $\mathbf{S}$, or positions k where the pair $\mathbf{S}(k)$ and the pair $\mathbf{S}'(k)$ are different. Assume that $\mathsf{hi}(\mathbf{S}) = n$.

- Case osNew:

$$(\mathsf{osNew}) \quad x \prec A \in \mathit{Decls}$$
$$\mathbf{S}_1 \circ (M, \mathtt{new}\,xE) \longrightarrow \mathbf{S}_1 \circ (M + x, AE)$$

  We only need to prove that $M + x, \Gamma \vdash AE : Z$ for some Z.

  Since $\Gamma \models \mathbf{S}$, we have $M, \Gamma \vdash \mathtt{new}\,xE : X$. By Generation Lemma 3.3.11, clause 3, we have $M, \Gamma \vdash \mathtt{new}\,x : X_1$ and $M + X_1^l, \Gamma \vdash E : X_2$ with

$$X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o, X_1^l + X_2^l \rangle$$

  Also by Generation Lemma 3.3.11, clause 1 applied to $M, \Gamma \vdash \mathtt{new}\,x : X_1$ and Lemma 3.3.12, we get $M, \Gamma \vdash A : Y$ with $X_1 = \langle Y^i + x, Y^o + x, Y^l + x \rangle$.

Now sequencing $M, \Gamma \vdash A\!:\!Y$ and $M + X_1^l, \Gamma \vdash E\!:\!X_2$, we get $M \cup (M + X_1^l - Y^l), \Gamma \vdash AE\!:\!Z$ with

$$Z = \langle Y^i \cup (Y^o + X_2^i), Y^o + X_2^o, Y^l + X_2^l \rangle$$

Since $X_1^l = Y^l + x$, we have $M \cup (M + X_1^l - Y^l) = M \cup (M + x) = M + x$ and we get the conclusion.

- Case osDel:

$$\begin{array}{l}(\mathsf{osDel}) \quad x \in M \\ \mathbf{S}_1 \circ (M, \mathtt{del}\, xE) \longrightarrow \mathbf{S}_1 \circ (M - x, E)\end{array}$$

We only need to prove that $M - x, \Gamma \vdash E\!:\!Z$ for some Z.

Since $\Gamma \models \mathbf{S}$, we have $M, \Gamma \vdash \mathtt{del}\, xE\!:\!X$. By Generation Lemma 3.3.11, clause 3, we have $M, \Gamma \vdash \mathtt{del}\, x\!:\!X_1$ and $M + X_1^l, \Gamma \vdash E\!:\!Z$ with

$$X = \langle X_1^i \cup (X_1^o + Z^i), X_1^o + Z^o, X_1^l + Z^l \rangle$$

Also by Generation Lemma 3.3.11, clause 2 applied to $M, \Gamma \vdash \mathtt{del}\, x : X_1$, we get $x \in M$ and $X_1 = \langle [], [-x], [-x] \rangle$. Hence, we have $M - x, \Gamma \vdash E\!:\!Z$.

- Case osChoice:

$$\begin{array}{l}(\mathsf{osChoice}) \quad i \in \{1, 2\} \\ \mathbf{S}_1 \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S}_1 \circ (M, A_i E)\end{array}$$

We treat the case $i = 1$, the other case is symmetric.

We only need to prove that $M, \Gamma \vdash A_1 E : Z$ for some Z. Since $\Gamma \models \mathbf{S}$, we have $M, \Gamma \vdash (A_1 + A_2)E\!:\!X$. By Generation Lemma 3.3.11, clause 3 applied to $M, \Gamma \vdash (A_1 + A_2)E\!:\!X$, we get $M, \Gamma \vdash (A_1 + A_2)\!:\!X_1$ and $M + X_1^l, \Gamma \vdash E\!:\!X_2$ with

$$X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o, X_1^l + X_2^l \rangle$$

Also by Generation Lemma 3.3.11, clause 4 applied to $M, \Gamma \vdash (A_1 + A_2)\!:\!X_1$, we get $M, \Gamma \vdash A_1\!:\!Y_1$, $M, \Gamma \vdash A_2\!:\!Y_2$ with $X_1 = \langle Y_1^i \cup Y_2^i, Y_1^o \cup Y_2^o, Y_1^l \cap Y_2^l \rangle$. Then we can apply the rule Seq and get $N, \Gamma \vdash A_1 E\!:\!Z$ with $N = M \cup (M + X_1^l - Y_1^l)$ and

$$Z = \langle Y_1^i \cup (Y_1^o + X_2^i), Y_1^o + X_2^o, Y_1^l + X_2^l \rangle$$

Since $X_1^l = Y_1^l \cap Y_2^l \subseteq Y_1^l$, we get $N = M$ and the clause holds.

- Case osPush:

$$\begin{array}{l}(\mathsf{osPush}) \\ \mathbf{S}_1 \circ (M, \{A\}E) \longrightarrow \mathbf{S}_1 \circ (M, E) \circ ([], A)\end{array}$$

We need to prove that $\Gamma \vdash A\!:\!Y$ and $\Gamma \vdash E\!:\!Z$ for some Y, Z.

Since $\Gamma \models \mathbf{S}$, we have $\Gamma \vdash \{A\}E\!:\!X$. By Generation Lemma 3.3.11, clause 3 and 5, we have $M, \Gamma \vdash \{A\}\!:\!X_1$ with $X_1^o = X_1^l = []$ and $M, \Gamma \vdash E\!:\!Z$ with

$$\begin{aligned} X &= \langle X_1^i \cup (X_1^o + Z^i), X_1^o + Z^o, X_1^l + Z^l \rangle \\ &= \langle X_1^i \cup Z^i, Z^o, Z^l \rangle \end{aligned}$$

Also by Generation Lemma 3.3.11, clause 5 applied to $M, \Gamma \vdash \{A\} : X_1$, we have $[], \Gamma \vdash A\!:\!Y$ with $Y^i = X_1^i$.

- Case osPop:

$$\begin{array}{l}(\mathsf{osPop}) \\ \mathbf{S}_1 \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S}_1 \circ (M, E)\end{array}$$

The clause holds by the hypothesis.

□

Proof of Lemma 3.3.5 (Progress). *If $\Gamma \models \mathbf{S}$, then either $\mathbf{S}$ is terminal or there exists a configuration $\mathbf{S}'$ such that $\mathbf{S} \longrightarrow \mathbf{S}'$.*

Proof. Since $\Gamma \models \mathbf{S}$, for the expression E at the top of $\mathbf{S}$, we have $M, \Gamma \vdash E : X$, where M is the local store of E. Among all the transition rules, there are two cases where the execution may get stuck. First, the execution is stuck if E has the form $\mathtt{new}\, xA$ and $x \notin \mathsf{dom}(\Gamma)$. But this has been guaranteed by Lemma 3.3.9. Second, the execution is stuck if E has the form $\mathtt{del}\, xA$ and $x \notin M$. But $x \in M$ follows by Lemma 3.3.4 and Generation Lemma 3.3.11, clause 2 and 3. □

Proof of Lemma 3.3.6 (Invariants of maxins). *If $\Gamma \models \mathbf{S}$ and $\mathbf{S} \longrightarrow \mathbf{S}'$, then*

$$\mathsf{maxins}(\mathbf{S}) \supseteq \mathsf{maxins}(\mathbf{S}')$$

Proof. The proof proceeds by case analysis on the transition relation $\longrightarrow$. Assume that $\mathsf{hi}(\mathbf{S}) = n$.

Let $\mathbf{S} = (M_1, C_1) \circ \ldots \circ (M_n, C_n)$ and $\mathbf{S}' = (N_1, E_1) \circ \ldots \circ (N_m, E_m)$. In the proof of Lemma 3.3.4, we have proved that all expressions in $\mathbf{S}'$ are well-typed. We assume that C_k has type V_k for $k = 1..n$ and E_k has type Z_k for $k = 1..m$.

To prove the clause, we will prove that for each element in $\{[\mathbf{S}'|_k] + Z_k^i \mid k = 1..m\}$ there exists an element in $\{[\mathbf{S}|_k] + V_k^i \mid k = 1..n\}$ such that the latter multiset includes the former.

- Case osNew:

$$\begin{array}{l}(\mathsf{osNew}) \quad x \prec A \in \mathit{Decls} \\ \mathbf{S}_1 \circ (M, \mathtt{new}\, xE) \longrightarrow \mathbf{S}_1 \circ (M + x, AE)\end{array}$$

  We have $m = n$. For $k < n$ we have $[\mathbf{S}|_k] + V_k^i = [\mathbf{S}'|_k] + Z_k^i$ since the two stacks $\mathbf{S}|_k$ and $\mathbf{S}'|_k$ are the same.

  For $k = n$ we will prove that $[\mathbf{S}|_n] + V_n^i = [\mathbf{S}'|_n] + Z_n^i$. Using the proof of Lemma 3.3.4 we have:

$$\begin{aligned}
[\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && \text{(Lemma 3.3.4)} \\
&= [\mathbf{S}|_n] + X_1^i \cup (X_1^o + X_2^i) && (X^i = X_1^i \cup (X_1^o + X_2^i)) \\
&= [\mathbf{S}|_n] + ((Y^i + x) \cup (Y^o + x + X_2^i)) && (X_1^* = Y^* + x) \\
&= [\mathbf{S}|_n] + x + (Y^i \cup (Y^o + X_2^i)) && \\
&= [\mathbf{S}|_n] + x + Z^i && (Z^i = Y^i \cup (Y^o + X_2^i)) \\
&= [\mathbf{S}'|_n] + Z^i && ([\mathbf{S}'|_n] = [\mathbf{S}|_n] + x) \\
&= [\mathbf{S}'|_n] + Z_n^i && (Z_n \text{ is } Z)
\end{aligned}$$

- Case osDel:

$$\begin{array}{l}(\mathsf{osDel}) \quad x \in M \\ \mathbf{S}_1 \circ (M, \mathtt{del}\, xE) \longrightarrow \mathbf{S}_1 \circ (M - x, E)\end{array}$$

  We have $m = n$. For $k < n$ we have $[\mathbf{S}|_k] + V_k^i = [\mathbf{S}'|_k] + Z_k^i$ since the two stacks $\mathbf{S}|_k$ and $\mathbf{S}'|_k$ are the same.

For $k = n$ we will prove that $[\mathbf{S}|_n] + V_n^i = [\mathbf{S}'|_n] + Z_n^i$. Using the proof of Lemma 3.3.4 we have:

$$\begin{aligned}
[\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && \text{(Lemma 3.3.4)}\\
&= [\mathbf{S}|_n] + X_1^i \cup (X_1^o + Z^i) && (X^i = X_1^i \cup (X_1^o + Z^i))\\
&= [\mathbf{S}|_n] + ([] \cup ([-x] + Z^i)) && (X_1^* = \langle [], [-x], [-x] \rangle)\\
&\supseteq [\mathbf{S}|_n] - x + Z^i\\
&= [\mathbf{S}'|_n] + Z^i && ([\mathbf{S}'|_n] = [\mathbf{S}|_n] - x)\\
&= [\mathbf{S}'|_n] + Z_n^i && (Z_n \text{ is } Z)
\end{aligned}$$

- Case osChoice:

$$\frac{(\mathsf{osChoice}) \quad i \in \{1,2\}}{\mathbf{S}_1 \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S}_1 \circ (M, A_i E)}$$

We treat the case $i = 1$, the other case is symmetric.

We have $m = n$. For $k < n$ we have $[\mathbf{S}|_k] + V_k^i = [\mathbf{S}'|_k] + Z_k^i$ since the two stacks $\mathbf{S}|_k$ and $\mathbf{S}'|_k$ are the same.

For $k = n$ we will prove that $[\mathbf{S}|_n] + V_n^i \supseteq [\mathbf{S}'|_n] + Z_n^i$. Using the proof of Lemma 3.3.4 we have:

$$\begin{aligned}
[\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && \text{(Lemma 3.3.4)}\\
&= [\mathbf{S}|_n] + X_1^i \cup (X_1^o + X_2^i) && (X^i = X_1^i \cup (X_1^o + X_2^i))\\
&\supseteq [\mathbf{S}|_n] + (Y_1^i \cup (Y_1^o + X_2^i)) && (X_1^* = Y_1^* \cup Y_2^*)\\
&= [\mathbf{S}|_n] + Z^i && (Z^i = Y_1^i \cup (Y_1^o + X_2^i))\\
&= [\mathbf{S}'|_n] + Z^i && ([\mathbf{S}'|_n] = [\mathbf{S}_1] + M = [\mathbf{S}|_n])\\
&= [\mathbf{S}'|_n] + Z_n^i && (Z_n \text{ is } Z)
\end{aligned}$$

- Case osPush:

$$\frac{(\mathsf{osPush})}{\mathbf{S}_1 \circ (M, \{A\}E) \longrightarrow \mathbf{S}_1 \circ (M, E) \circ ([], A)}$$

We have $m = n + 1$. For $k < n$ we have $[\mathbf{S}|_k] + V_k^i = [\mathbf{S}'|_k] + Z_k^i$ since the two stacks $\mathbf{S}|_k$ and $\mathbf{S}'|_k$ are the same.

For $k = n$ we will prove that $[\mathbf{S}|_n] + V_n^i \supseteq [\mathbf{S}'|_n] + Z_n^i$. Using the proof of Lemma 3.3.4 we have:

$$\begin{aligned}
[\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && \text{(Lemma 3.3.4)}\\
&= [\mathbf{S}|_n] + (X_1^i \cup Z^i) && (X^i = X_1^i \cup Z^i)\\
&\supseteq [\mathbf{S}|_n] + Z^i\\
&= [\mathbf{S}'|_n] + Z_n^i && (Z_n \text{ is } Z)
\end{aligned}$$

For $k = n + 1$ we will prove that $[\mathbf{S}|_n] + V_n^i \supseteq [\mathbf{S}'|_{n+1}] + Z_{n+1}^i$. Using the proof of Lemma 3.3.4 we have:

$$\begin{aligned}
[\mathbf{S}|_n] + V_n^i &= [\mathbf{S}|_n] + X^i && \text{(Lemma 3.3.4)}\\
&= [\mathbf{S}|_n] + (X_1^i \cup Z^i) && (X^i = X_1^i \cup Z^i)\\
&\supseteq [\mathbf{S}'|_n] + Y_1^i && (X_1^i = Y^i)\\
&= [\mathbf{S}'|_{n+1}] + Z_{n+1}^i && (\mathbf{S}'(n+1) = [])
\end{aligned}$$

- Case osPop:

$$\begin{array}{l}(\mathsf{osPop})\\ \mathbf{S}_1 \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S}_1 \circ (M, E)\end{array}$$

  We have $m = n - 1$. The clause holds easily since the two stacks $\mathbf{S}|_{n-1}$ and $\mathbf{S}'|_{n-1}$ are the same.

$\square$

Proof of Theorem 3.3.7 (Soundness). *Let Prog = Decls; E be well-typed, that is,* $\Gamma \vdash E : X$ *for some reordering* Γ *of Decls and some type* X*. Then for any* $\mathbf{S}$ *such that* $([], E) \longrightarrow^* \mathbf{S}$ *we have that* $\mathbf{S}$ *is not stuck and* $[\mathbf{S}] \subseteq X^i$.

Proof. The same as the proof of Soundness Theorem 2.3.7 in Chapter 2. $\square$

Termination. As in Chapter 2, all well-typed programs terminate after a finite number of transition steps. We can prove this property by the same method of Chapter 2, with the function mts extended for the new form of expressions: $\mathtt{del}\, x$. Since executing $\mathtt{del}\, x$ takes only one step and we do not continue executing the body of x, the function for $\mathtt{del}\, x$ always returns 1.

$$\mathsf{mts}(\,\mathtt{del}\, x) = 1$$

3.4 Type Inference

The type inference for programs of this chapter is almost the same as in Chapter 2. Note that type for expression $\mathtt{del}\, x$ is $\langle [], [-x], [-x] \rangle$ for every x.

Chapter 4

Explicit Deallocation and Parallel Composition

In this chapter, we extend the language in Chapter 3 by adding parallel composition that allows many configuration stacks in Chapter 3 running in parallel. The syntax and the type system of this chapter need only a few updates from the ones of Chapter 3. However, the operational semantics needs a substantial change and the type soundness is more sophisticated. The type inference for programs of this chapter is almost the same as in Chapter 2, therefore, we leave it out here for brevity.

4.1 Language

4.1.1 Syntax

Table 4.1 defines the syntax of the language using the extended Backus-Naur Form as in the previous chapters. Comparing to the language of Chapter 3, the syntax has only one new form of expressions created by the parallel composition. The notions of programs and declarations are the same as in Section 2.1.1.

Table 4.1: Syntax of the language with `del` and parallel composition

Prog	::=	$\mathit{Decls}; E$	Program
Decls	::=	$\overline{x \prec E}$	Declarations
E	::=		Expression
		ϵ	Empty
	|	$\mathtt{new}\,x$	Instantiation
	|	$\mathtt{del}\,x$	Deallocation
	|	$E\ E$	Sequencing
	|	$(E + E)$	Choice
	|	$(E \parallel E)$	Parallel
	|	$\{E\}$	Scope

We give an example program to demonstrate the operational semantics and typing derivations in the subsequent sections. In this example, d and e are primitive components. Component a is the parallel composition of $\{\mathtt{new}\,d\}\,\mathtt{new}\,e$ and $\mathtt{new}\,d$ followed by a

deallocation of d. Component b has a choice expression before deleting an instance of e.

$$
\begin{aligned}
& d \prec \epsilon \quad e \prec \epsilon \\
& a \prec (\{\, \texttt{new}\, d\} \,\texttt{new}\, e \parallel \texttt{new}\, d)\, \texttt{del}\, d \\
& b \prec (\,\texttt{new}\, a + \texttt{new}\, e\, \texttt{new}\, d)\, \texttt{del}\, e; \\
& \texttt{new}\, b
\end{aligned}
$$

4.1.2 Operational Semantics

Informally, expression E can be viewed as a sequence of commands of the form $\texttt{new}\, x$, $\texttt{del}\, x, (A + B), \{A\}$ and $(A \parallel B)$ in imperative programming languages and the execution is sequential from left to right. As in previous chapters, E is paired with a multiset (local store). The first four commands behave the same as in Chapter 3. Executing $(E_1 \parallel E_2)$ suspends the execution of the commands after it and creates two pairs $([], E_1)$ and $([], E_2)$, called child threads. These child threads are executed concurrently and independently. When a thread terminates in the pair (M, ϵ), the instances in M are *returned* to the store at the top of its parent stack. When all the child threads terminate, the execution resumes to the parent thread.

The formal operational semantics is defined by a rewriting system [40] of *configuration*s. A configuration is a binary tree $\mathbb{T}$ of *thread*s. A thread is a stack $\mathbf{S}$ of pairs of a local store and an expression (M, E), where M is a multiset over component names $\mathbb{C}$, and E is an expression as defined in Table 4.1. A thread is *active* if it is a leaf thread. A configuration is *terminal* if it has only one (root) thread of the form (M, ϵ). Figure 4.1 illustrates stacks and configurations. Stacks and configuration trees are denoted by the following syntax:

$\mathbf{S}$	$::=$	$(M_1, E_1) \circ ... \circ (M_n, E_n)$	Stack
$\mathbb{T}$, $\mathbb{R}$	$::=$		Configurations
		$\mathsf{Lf}(\mathbf{S})$	Leaf
	$\mid$	$\mathsf{Nd}(\mathbf{S}, \mathbb{T})$	Node with one branch
	$\mid$	$\mathsf{Nd}(\mathbf{S}, \mathbb{T}, \mathbb{T})$	Node with two branches

The above stack $\mathbf{S}$ has n elements where (M_1, E_1) is the bottom, (M_n, E_n) is the top of the stack, and '$\circ$' is the stack separator. A node in the binary trees may have no child nodes $\mathsf{Lf}(\mathbf{S})$—a leaf, or one branch $\mathsf{Nd}(\mathbf{S}, \mathbb{T})$, or two branches $\mathsf{Nd}(\mathbf{S}, \mathbb{T}, \mathbb{T})$.

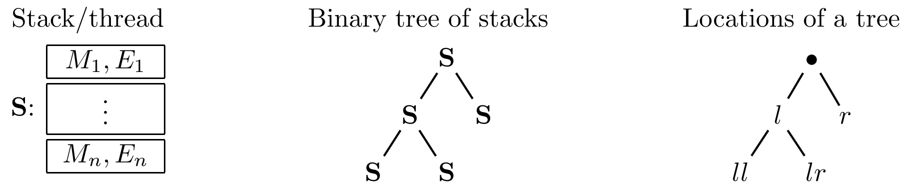


Figure 4.1: Illustration of a tree of stacks

We assign a *location* to each node in a tree, as illustrated in Figure 4.1. A location is a sequence over $\{l, r\}$. Let α, β range over locations. The root is assigned the empty sequence, notation $\bullet$, but in most cases we will ignore this notation. The locations of two direct nodes from the root are l and r. The locations of the two direct child nodes of l are ll and lr, and so on. In general, αl and αr are the locations of the direct children of α. We write $\alpha \in \mathbb{T}$ when α is a location of a node in the tree $\mathbb{T}$. Whenever a new node is created, a location is assigned to it and this location will not be changed until the node is removed from the tree.

By $\mathbb{T}[\,]_\alpha$ we denote a tree with a hole at the leaf location α. Filling this hole with another tree $\mathbb{R}$ is denoted by $\mathbb{T}[\![\mathbb{R}]\!]_\alpha$. One-step reduction is defined first by choosing an

arbitrary active thread. Then depending on the pattern of the chosen thread and the state of the configuration, the appropriate rewrite rule can be applied. The rewriting rules for these patterns or subconfigurations, notation $\mathbb{R} \rightsquigarrow \mathbb{R}'$, are called the basic reduction relation. The configuration $\mathbb{T}[\![\mathbb{R}]\!]_\alpha$ can take a step to $\mathbb{T}[\![\mathbb{R}']\!]_\alpha$, notation $\mathbb{T}[\![\mathbb{R}]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathbb{R}']\!]_\alpha$, if $\mathbb{R} \rightsquigarrow \mathbb{R}'$. As usual, $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

Table 4.2: Basic reduction rules of the language with `del` and parallel composition

(osNew) $x \prec A \in Decls$
$\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{new}\, xE)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M + x, AE))$

(osDel) $x \in M$
$\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{del}\, xE)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M - x, E))$

(osChoice) $i \in \{1, 2\}$
$\mathsf{Lf}(\mathbf{S} \circ (M, (A_1 + A_2)E)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M, A_i E))$

(osPush)
$\mathsf{Lf}(\mathbf{S} \circ (M, \{A\}E)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M, E) \circ ([], A))$

(osPop)
$\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ (M', \epsilon)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M, E))$

(osParIntr)
$\mathsf{Lf}(\mathbf{S} \circ (M, (A \parallel B)E)) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B)))$

(osParElimL)
$\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)), \mathbb{R}) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})$

(osParElimR)
$\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathbb{R}, \mathsf{Lf}((M', \epsilon))) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})$

(osParElim)
$\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon))) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M + M', E))$

Table 4.2 defines the basic reduction relation. Each basic reduction rule has two lines. The first line contains a rule name followed by a list of conditions. The second line has the form $\mathbb{R} \rightsquigarrow \mathbb{R}'$, which states that if a configuration $\mathbb{T}$ has a subconfiguration of the form $\mathbb{R}$ and all the conditions in the first line hold, then we can replace the subconfiguration $\mathbb{R}$ of $\mathbb{T}$ by subconfiguration $\mathbb{R}'$ and the system moves to the new state $\mathbb{T}[\![\mathbb{R}']\!]$.

The rules osNew, osDel, and osChoice only cause local changes to the pair at the top of the leaf stack. The rule osNew first creates a new instance of component x in the local store. Then if x is a primitive component, it continues to execute the remaining expression E; otherwise, it continues to execute A before executing the remaining expression E. The rule osDel deallocates an instance of x in the local store if there exists one. If there exists no instance of x in the local store, the execution is stuck. Note that, again, we have abstracted away the specific instance that is deleted. The rule osChoice selects a branch to execute.

The next two rules, osPush and osPop, behave as in the previous chapters. The rule osPush pushes an element on the top of the leaf stack. The rule osPop pops the stack when the stack has at least two elements. That means no stack in any configuration is empty. The last four rules change the tree structure of the configuration. By the rule osParIntr, a leaf is replaced by a branch of a node and two leaves. In contrast, by the rules osParElimR, osParElimL, osParElim, a leaf is removed from the tree and the instances surviving at the leaf are returned to the store at the top of the parent stack. When appropriate, the parent node becomes a leaf (see the rule osParElim).

The example at the end of Section 4.1.1 is used to illustrate the operational semantics. There are many possible runs of the program due to the choice composition and when a

configuration has more than one leaf thread, the number of possible runs can be exponential as active threads have the same priority. Here we only show one of the possible runs. To make it easier to follow, we represent the trees graphically instead of using the formal syntax; '$\leftharpoondown$' and '$\langle$' denote branches with one and two child nodes, respectively. At the starting point, the configuration has one leaf $\mathsf{Lf}([], \mathtt{new}\, b)$. After the first step, there are two possibilities by the rule $\mathsf{osChoice}$.

$$\begin{array}{rl}
(\mathsf{Start}) & ([], \mathtt{new}\, b) \\
(\mathsf{osNew}) \longrightarrow & ([b], (\mathtt{new}\, a + \mathtt{new}\, e\, \mathtt{new}\, d)\, \mathtt{del}\, e) \\
(\mathsf{osChoice}) \longrightarrow & ([b], \mathtt{new}\, a\, \mathtt{del}\, e) \qquad \qquad (\text{or } ([b], \mathtt{new}\, e\, \mathtt{new}\, d\, \mathtt{del}\, e))
\end{array}$$

Now we continue with the first possibility. When the tree grows two more leaves, we draw a box around the leaf that will be executed in the next step.

$$\begin{array}{rl}
 & ([b], \mathtt{new}\, a\, \mathtt{del}\, e) \\
(\mathsf{osNew}) \longrightarrow & ([b, a], (\{\, \mathtt{new}\, d\}\, \mathtt{new}\, e \parallel \mathtt{new}\, d)\, \mathtt{del}\, d\, \mathtt{del}\, e) \\
(\mathsf{osParIntr}) \longrightarrow & ([b, a], \mathtt{del}\, d\, \mathtt{del}\, e) \left\langle \begin{array}{l} ([], \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e) \\ \boxed{([], \mathtt{new}\, d)} \end{array} \right. \\
(\mathsf{osNew}) \longrightarrow & ([b, a], \mathtt{del}\, d\, \mathtt{del}\, e) \left\langle \begin{array}{l} \boxed{([], \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e)} \\ ([d], \epsilon) \end{array} \right. \\
(\mathsf{osPush}) \longrightarrow & ([b, a], \mathtt{del}\, d\, \mathtt{del}\, e) \left\langle \begin{array}{l} \boxed{([], \mathtt{new}\, e) \circ ([], \mathtt{new}\, d)} \\ ([d], \epsilon) \end{array} \right. \\
(\mathsf{osNew}) \longrightarrow & ([b, a], \mathtt{del}\, d\, \mathtt{del}\, e) \left\langle \begin{array}{l} ([], \mathtt{new}\, e) \circ ([d], \epsilon) \\ \boxed{([d], \epsilon)} \end{array} \right. \\
(\mathsf{osParElimL}) \longrightarrow & ([b, a, d], \mathtt{del}\, d\, \mathtt{del}\, e) \leftharpoondown ([], \mathtt{new}\, e) \circ ([d], \epsilon) \\
(\mathsf{osPop}) \longrightarrow & ([b, a, d], \mathtt{del}\, d\, \mathtt{del}\, e) \leftharpoondown ([], \mathtt{new}\, e) \\
(\mathsf{osNew}) \longrightarrow & ([b, a, d], \mathtt{del}\, d\, \mathtt{del}\, e) \leftharpoondown ([e], \epsilon) \\
(\mathsf{osParElim}) \longrightarrow & ([b, a, d, e], \mathtt{del}\, d\, \mathtt{del}\, e) \\
(\mathsf{osDel}) \longrightarrow & ([b, a, e], \mathtt{del}\, e) \\
(\mathsf{osDel}) \longrightarrow & ([b, a], \epsilon) \qquad \qquad (\text{terminal})
\end{array}$$

4.2 Type System

The type system for the language with the additional parallel composition is almost the same as the type system of the previous chapter (Section 3.2). We have only new typing rule for the parallel composition, other notions of types, environments, typing judgments are the same. Therefore, we will only repeat some important definitions and update the typing relation in Table 4.3.

Definition 4.2.1 (Types). *Types of component expressions are tuples*

$$X = \langle X^i, X^o, X^l \rangle$$

where X^i is a multiset (no negative occurrences) and X^o, X^l are signed multisets over $\mathbb{C}$.

A *typing judgment* is also a tuple of the form $\sigma, \Gamma \vdash E : X$ and it asserts that expression E has type X in the environment σ, Γ. Valid typing judgments are defined as follows.

Definition 4.2.2 (Valid typing judgments). *Valid typing judgments $\sigma, \Gamma \vdash A : X$ are derived by applying the typing rules in Table 4.3 in the usual inductive way.*

Table 4.3: Typing rules of the language with `del` and parallel composition

(Axiom)
$$\frac{}{[], \varnothing \vdash \epsilon : \langle [], [], [] \rangle}$$

(WeakenB)
$$\frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma)}{\sigma_1, \Gamma, x \prec B \vdash A : X}$$

(WeakenS)
$$\frac{\sigma, \Gamma \vdash A : X \quad \sigma \subseteq \sigma_1}{\sigma_1, \Gamma \vdash A : X}$$

(New)
$$\frac{\sigma, \Gamma \vdash A : X \quad x \notin \mathsf{dom}(\Gamma)}{\sigma, \Gamma, x \prec A \vdash \mathtt{new}\, x : \langle X^i + x, X^o + x, X^l + x \rangle}$$

(Del)
$$\frac{\sigma, \Gamma \vdash A : X \quad x \in \mathsf{dom}(\Gamma)}{[x], \Gamma \vdash \mathtt{del}\, x : \langle [], [-x], [-x] \rangle}$$

(Seq)
$$\frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y \quad A, B \neq \epsilon}{\sigma_1 \cup (\sigma_2 - X^l), \Gamma \vdash AB : \langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l \rangle}$$

(Choice)
$$\frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y}{\sigma_1 \cup \sigma_2, \Gamma \vdash (A + B) : \langle X^i \cup Y^i, X^o \cup Y^o, X^l \cap Y^l \rangle}$$

(Scope)
$$\frac{[], \Gamma \vdash A : X}{[], \Gamma \vdash \{A\} : \langle X^i, [], [] \rangle}$$

(Parallel)
$$\frac{[], \Gamma \vdash A : X \quad [], \Gamma \vdash B : Y}{[], \Gamma \vdash (A \parallel B) : \langle X^i + Y^i, X^o + Y^o, X^l + Y^l \rangle}$$

All the old typing rules are the same as in the previous chapter. We have only an additional typing rule Parallel for the parallel composition. Like the rule Scope, the rule Parallel requires the empty stores in the environments of the premises because the semantics of deallocation applies to the local store only (see the rule osDel).

The notion of *well-typed program* is also the same as in Chapter 3.

Definition 4.2.3 (Well-typed programs). *Program $Prog = Decls; E$ is well-typed if there exist a reordering Γ of declarations in $Decls$ and a type X such that $[], \Gamma \vdash E : X$.*

Using the example in Section 4.1.1, we derive type for $\mathtt{new}\, b$. Note that we omitted some side conditions as they can be checked easily and we shortened the rule names to the first two characters. Since we do not use the rule WeakenS, We stands for WeakenB. The rule Axiom is also simplified.

$$\text{We}\frac{\text{Sc}\dfrac{\text{Ne}\dfrac{[], \varnothing \vdash \epsilon : \langle [], [], [] \rangle}{[], d \prec \epsilon \vdash \mathtt{new}\, d : \langle [d], [d], [d] \rangle}}{[], d \prec \epsilon \vdash \{\mathtt{new}\, d\} : \langle [d], [], [] \rangle} \quad \text{We}\dfrac{[], \varnothing \vdash \epsilon : \langle [], [], [] \rangle \quad [], \varnothing \vdash \epsilon : \langle [], [], [] \rangle}{[], d \prec \epsilon \vdash \epsilon : \langle [], [], [] \rangle}}{[], d \prec \epsilon, e \prec \epsilon \vdash \{\mathtt{new}\, d\} : \langle [d], [], [] \rangle} \tag{4.1}$$

$$\text{Se}\frac{(4.1) \quad \text{Ne}\dfrac{\text{We}\dfrac{[], \varnothing \vdash \epsilon : \langle [], [], [] \rangle \quad [], \varnothing \vdash \epsilon : \langle [], [], [] \rangle}{[], d \prec \epsilon \vdash \epsilon : \langle [], [], [] \rangle}}{[], d \prec \epsilon, e \prec \epsilon \vdash \mathtt{new}\, e : \langle [e], [e], [e] \rangle}}{[], d \prec \epsilon, e \prec \epsilon \vdash \{\mathtt{new}\, d\}\, \mathtt{new}\, e : \langle [d, e], [e], [e] \rangle} \tag{4.2}$$

$$\text{Pa}\frac{(4.2) \quad \text{We}\dfrac{\text{Ne}\dfrac{[], \varnothing \vdash \epsilon : \langle [], [], [] \rangle}{[], d \prec \epsilon \vdash \mathtt{new}\, d : \langle [d], [d], [d] \rangle} \quad [], \varnothing \vdash \epsilon : \langle [], [], [] \rangle}{[], d \prec \epsilon, e \prec \epsilon \vdash \mathtt{new}\, d : \langle [d], [d], [d] \rangle}}{[], d \prec \epsilon, e \prec \epsilon \vdash (\{\mathtt{new}\, d\}\, \mathtt{new}\, e \parallel \mathtt{new}\, d) : \langle [d, d, e], [d, e], [d, e] \rangle} \tag{4.3}$$

$$\textsc{Ne}\dfrac{\textsc{Se}\dfrac{(4.3) \qquad \textsc{De}\dfrac{(4.3) \quad d \in \mathsf{dom}(d\prec \epsilon, e\prec \epsilon)}{[d], d\prec \epsilon, e\prec \epsilon \vdash \texttt{del}\, d : \langle [], [-d], [-d] \rangle}}{[], d\prec \epsilon, e\prec \epsilon \vdash (\{\texttt{new}\, d\}\, \texttt{new}\, e \parallel \texttt{new}\, d)\, \texttt{del}\, d : \langle [d, d, e], [e], [e] \rangle}}{[], d\prec \epsilon, e\prec \epsilon, a\prec (\{\texttt{new}\, d\}\, \texttt{new}\, e \parallel \texttt{new}\, d)\, \texttt{del}\, d \vdash \texttt{new}\, a : \langle [a, d, d, e], [a, e], [a, e] \rangle}$$

Similarly, we can derive $\Gamma \vdash \texttt{new}\, b : \langle [a, b, d, d, e], [a, b, d], [b] \rangle$ where $\Gamma = d\prec \epsilon, e\prec \epsilon, a\prec (\{\texttt{new}\, d\}\, \texttt{new}\, e \parallel \texttt{new}\, d)\, \texttt{del}\, d, b\prec (\texttt{new}\, a + \texttt{new}\, e\, \texttt{new}\, d)\, \texttt{del}\, e$.

4.3 Properties

As in Chapter 2, first we state the type soundness property, then we prove some typing properties, and finally we prove the type soundness. The type soundness proof is a bit more sophisticated because of the semantics of the parallel composition. Proofs are delegated to Section 4.3.2 and 4.3.3.

4.3.1 Type Soundness

As in Chapter 3, besides proving the type soundness, we will prove an additional property which guarantees that a well-typed program will not create more instances than a bound given by the type of the main expression of the program. We start with some auxiliary definitions.

First, since the location of a parent node is a subsequence of the location of its children (direct and indirect), we define the following binary prefix ordering relation $\preccurlyeq$ over locations. For two locations α and α', we define $(\alpha, \alpha') \in \preccurlyeq$, also written $\alpha \preccurlyeq \alpha'$, if and only if α is an initial subsequence of α'. That is, if $\alpha' = s_0 s_1..s_n$, then $\alpha = s_0 s_1..s_m$ for some $0 \leq m \leq n$. The set of all locations in a tree and this binary relation form a partially ordered set [17]. A maximal element of this partially ordered set is the location of a leaf. We denote by $\mathsf{leaves}(\mathbb{T})$ the set of locations of all the leaves of tree $\mathbb{T}$ and $\mathbb{T}(\alpha)$ the stack at location $\alpha \in \mathbb{T}$.

Second, we call $\alpha.k$ the *position* of the kth element from the bottom of the stack $\mathbb{T}(\alpha)$. Again the set of all positions $\alpha.k$ in tree $\mathbb{T}$ together with the binary relation: $\alpha_1.k_1 \preccurlyeq \alpha_2.k_2$ if either $\alpha_1 = \alpha_2$ and $k_1 \leq k_2$, or $\alpha_1 \prec \alpha_2$, forms a partially ordered set.

Next, we formalize the notion of *subtree*. Given a tree $\mathbb{T}$. The set of positions $\mathcal{L} = \{\alpha_i.k_i \in \mathbb{T} \mid 1 \leq i \leq m\}$ is *valid* if $\alpha_i.k_i \not\preccurlyeq \alpha_j.k_j$ for all $i \neq j$. Tree $\mathbb{T}'$ obtained from $\mathbb{T}$ by removing all elements at positions $\alpha.k \succcurlyeq \alpha_i.k_i$ for all $1 \leq i \leq m$ is a subtree of $\mathbb{T}$, notation $\mathbb{T}' \sqsubseteq_{\mathcal{L}} \mathbb{T}$ or $\mathbb{T}' = \mathbb{T}|_{\mathcal{L}}$. Consequently, $\mathbb{T}'$ has the same root as $\mathbb{T}$. When $\mathcal{L}$ is empty, we get $\mathbb{T}' = \mathbb{T}$.

More denotations: $\mathsf{hi}(\mathbf{S})$ is the height of the stack $\mathbf{S}$; $\mathbb{T}(\alpha.k) = (M, E)$ denotes that the pair at position $\alpha.k$ is the pair (M, E); $[\mathbb{T}(\alpha.k)]$ is the store M at position $\alpha.k$; $[\mathbb{T}(\alpha)]$ is the additive union of all stores in the stack at location α; and $[\mathbb{T}]$ is the multiset of all active instances in the tree $\mathbb{T}$, that is, $[\mathbb{T}] = \biguplus_{\alpha \in \mathbb{T}} [\mathbb{T}(\alpha)]$.

Now we calculate the maximum number of instances that a configuration can generate. Unlike the semantics of scope (the rule osPop) that discards all the instances in the local store upon exiting the scope, when a thread terminates all the instances in the local store are added to the store at the top of the parent thread (see the rules $\mathsf{osParElimL}$, $\mathsf{osParElimR}$ and $\mathsf{osParElim}$). Therefore, in order to calculate the maximum number of instances that a configuration can reach, we need to calculate the collection of instances that will be returned to a position $\alpha.k$.

Due to the non-determinism of the rule $\mathsf{osChoice}$, we can only calculate the upper bound and the lower bound of the collection. The minimal number of instances that will be returned to the store a position $\alpha.k$, denoted by function $\mathsf{retl}_{\mathbb{T}}(\alpha.k)$, is zero if k is not the top of the stack at location α, or α is a leaf. Otherwise, it contains those elements in

the multisets at the bottom of child nodes of $\alpha.k$ and the minimal number of instances that the expressions in the child nodes generate. Since the bottom of the child nodes of $\alpha.k$ may receive instances from its child nodes and so on, we need to call the function recursively.

$$\mathsf{retl}_{\mathbb{T}}(\alpha.k) = \begin{cases} [], & \text{if } k < \mathsf{hi}(\mathbb{T}(\alpha)) \text{ or } \alpha \in \mathsf{leaves}(\mathbb{T}) \\ \biguplus_{\beta \in \{\alpha l, \alpha r\}} (M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.1)), & \text{otherwise} \end{cases}$$

where $\mathbb{T}(\beta.1) = (M, E)$ and $M + \mathsf{retl}_{\mathbb{T}}(\beta.1), \Gamma \vdash E : X$. This recursive definition is non-circular since first it is well-defined for all the positions at all leaves. Then it is well-defined for the top position of the parents of all leaves. And so on until the root.

The maximal number of instances that will be returned to a position $\alpha.k$, denoted by function $\mathsf{retop}_{\mathbb{T}}(\alpha.k)$, is calculated analogously.

$$\mathsf{retop}_{\mathbb{T}}(\alpha.k) = \begin{cases} [], & \text{if } k < \mathsf{hi}(\mathbb{T}(\alpha)) \text{ or } \alpha \in \mathsf{leaves}(\mathbb{T}) \\ \biguplus_{\beta \in \{\alpha l, \alpha r\}} (M + X^o + \mathsf{retop}_{\mathbb{T}}(\beta.1)), & \text{otherwise} \end{cases}$$

where $\mathbb{T}(\beta.1) = (M, E)$ and $M + \mathsf{retl}_{\mathbb{T}}(\beta.1), \Gamma \vdash E : X$.

By Lemma 4.3.9 below, these two functions always return multisets even though signed multisets X^l and X^o appear in their definitions.

Now we can define the notion of well-typed configuration. It guarantees that the local store always has enough elements for typing its executing expression. Hence, deallocation operations are always safe to execute. For a position $\alpha.k$ that is not at the top of a leaf, the local store is at least $[\mathbb{T}(\alpha.k)] + \mathsf{retl}_{\mathbb{T}}(\alpha.k)$ when the expression at $\alpha.k$ is executed.

Definition 4.3.1 (Well-typed configurations). *Configuration $\mathbb{T}$ is well-typed with respect to a basis Γ, notation $\Gamma \models \mathbb{T}$, if for each pair (M, E) at position $\alpha.k \in \mathbb{T}$ there exists X such that*

$$M + \mathsf{retl}_{\mathbb{T}}(\alpha.k), \Gamma \vdash E : X$$

The formal definitions of terminal configurations and stuck states are the same as in previous chapters.

Definition 4.3.2 (Terminal configurations). *A configuration $\mathbb{T}$ is terminal if it has the form (M, ϵ).*

Definition 4.3.3 (Stuck states). *A configuration $\mathbb{T}$ is stuck if no transition rule applies and $\mathbb{T}$ is not terminal.*

Now, we state the two main lemmas mentioned at the beginning of the section.

Lemma 4.3.4 (Preservation). *If $\Gamma \models \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then $\Gamma \models \mathbb{T}'$.*

Lemma 4.3.5 (Progress). *If $\Gamma \models \mathbb{T}$, then either $\mathbb{T}$ is terminal or there exists a configuration $\mathbb{T}'$ such that $\mathbb{T} \longrightarrow \mathbb{T}'$.*

Next, we show some additional invariants which allow us to infer the upper resource bounds of well-typed programs. Then we state the Soundness Theorem which contains both goals mentioned at the beginning of the section.

Consider the pair (M, E) at position $\alpha.k$ in a well-typed configuration $\mathbb{T}$, by Definition 4.3.1, we have $M + \mathsf{retl}_{\mathbb{T}}(\alpha.k), \Gamma \vdash E : X$ for some X. The maximum number of instances that the pair $\mathbb{T}(\alpha.k)$ can reach is computed by:

$$\mathsf{io}_{\mathbb{T}}(\alpha.k) = M + \mathsf{retop}_{\mathbb{T}}(\alpha.k) + X^i$$

The following lemma shows the monotonicity of the above three functions with respect to one-step reduction $\longrightarrow$. In particular, at nodes $\alpha.k$ that are the same in the two

trees, $\mathbb{T}(\alpha.k) = \mathbb{T}'(\alpha.k)$, the function retl and retop are non-descending and non-ascending monotonic, resepectively. The function io is non-ascending monotonic at all positions that are on both trees.

Lemma 4.3.6 (Invariants of retl, retop, and io). *If $\Gamma \models \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then for all positions $\alpha.k$ in both configurations $\mathbb{T}$ and $\mathbb{T}'$ we have:*

1. $\mathsf{retl}_{\mathbb{T}}(\alpha.k) \subseteq \mathsf{retl}_{\mathbb{T}'}(\alpha.k)$ *if* $\mathbb{T}(\alpha.k) = \mathbb{T}'(\alpha.k)$,
2. $\mathsf{retop}_{\mathbb{T}}(\alpha.k) \supseteq \mathsf{retop}_{\mathbb{T}'}(\alpha.k)$ *if* $\mathbb{T}(\alpha.k) = \mathbb{T}'(\alpha.k)$,
3. $\mathsf{io}_{\mathbb{T}}(\alpha.k) \supseteq \mathsf{io}_{\mathbb{T}'}(\alpha.k)$.

The maximum number of instances that a subtree $\mathbb{T}|_{\mathcal{L}}$ can reach includes the maximum number of instances that can be created by its leaves and all the existing instances in all the stores inside the subtree.

$$\mathsf{maxins}(\mathbb{T}|_{\mathcal{L}}) = \biguplus_{\alpha.k \prec \mathcal{L}'} [\mathbb{T}(\alpha.k)] + \biguplus_{\alpha.k \in \mathcal{L}'} \mathsf{io}_{\mathbb{T}}(\alpha.k)$$

where $\mathcal{L}'$ is the set of all positions at the top of leaves of the subtree $\mathbb{T}|_{\mathcal{L}}$:

$$\mathcal{L}' = \{\alpha.\mathsf{hi}(\mathbb{T}|_{\mathcal{L}}(\alpha)) \mid \alpha \in \mathsf{leaves}(\mathbb{T}|_{\mathcal{L}})\}$$

By the monotonicity of the function io, the function maxins is also monotonic.

Lemma 4.3.7 (Invariant of maxins). *If $\Gamma \models \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then for all valid sets of positions $\mathcal{L}'$ of $\mathbb{T}'$ there exists a valid set of positions $\mathcal{L}$ of $\mathbb{T}$ such that*

$$\mathsf{maxins}(\mathbb{T}|_{\mathcal{L}}) \supseteq \mathsf{maxins}(\mathbb{T}'|_{\mathcal{L}'})$$

Now we can state the type soundness together with the upper bound of instances that a well-typed program always respects.

Theorem 4.3.8 (Soundness). *Let $Prog = Decls; E$ be well-typed, that is, $\Gamma \vdash E : X$ for some reordering Γ of Decls and some type X. Then for any $\mathbb{T}$ such that $\mathsf{Lf}([], E) \longrightarrow^* \mathbb{T}$ we have that $\mathbb{T}$ is not stuck and $[\mathbb{T}] \subseteq X^i$.*

4.3.2 Typing Properties

The following typing properties are analogous to the ones in Chapter 3. First, we update some definitions. The terminology on bases are the same as in Definition 3.3.8. Recall, $\mathsf{dom}(M) = \{x \mid M(x) \neq 0\}$ for a signed multiset M. Function $\mathsf{var}(E)$ is extended from the definition in Section 3.3.2 with the new parallel composition as follows.

$$\mathsf{var}((A \parallel B)) = \mathsf{var}(A) \cup \mathsf{var}(B)$$

Lemma 4.3.9 (Valid typing judgment). *If $\sigma, \Gamma \vdash A : X$, then*

1. $\mathsf{var}(A) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{dom}(X^*) \subseteq \mathsf{dom}(\Gamma)$,
2. *every variable in* $\mathsf{dom}(\Gamma)$ *is declared only once in* Γ,
3. $X^i \supseteq X^o \supseteq X^l$, $X^i \supseteq []$,
4. $\sigma + X^* \supseteq []$.

Proof. By induction on typing derivations. We will only show the proof for the case where $\sigma, \Gamma \vdash A : X$ is derived by the rule Parallel. The proofs for the other cases are the same as in the proof of Lemma 3.3.9.

Suppose that $\sigma, \Gamma \vdash A{:}X$ is derived by the rule Parallel:

$$\frac{[], \Gamma \vdash B{:}Y \quad [], \Gamma \vdash C{:}Z}{[], \Gamma \vdash (B \parallel C){:}\langle Y^i + Z^i, Y^o + Z^o, Y^l + Z^l\rangle}\ (\textsc{Parallel})$$

By the induction hypothesis, we have $\mathsf{var}(B) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{dom}(Y^*) \subseteq \mathsf{dom}(\Gamma)$, and $\mathsf{var}(C) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{dom}(Z^*) \subseteq \mathsf{dom}(\Gamma)$. So the first clause follows easily since $\mathsf{var}(B \parallel C) = \mathsf{var}(B) \cup \mathsf{var}(C)$ and $\mathsf{dom}(Y^* + Z^*) \subseteq \mathsf{dom}(Z^*) \cup \mathsf{dom}(Z^*)$.

Clause 2 holds by the induction hypothesis. Clause 3 and clause 4 also follow by the induction hypothesis and the definition of the additive unions on multisets and signed multisets.

□

Lemma 4.3.10 (Associativity). *If $\sigma_i, \Gamma \vdash A_i : X_i$, for $i \in \{1, 2, 3\}$, then the typing judgments for $(A_1A_2)A_3$ and $A_1(A_2A_3)$ are the same.*

Proof. The proof is the same as in the proof of Lemma 3.3.10.

□

Lemma 4.3.11 (Generation).

1. *If $\sigma, \Gamma \vdash \mathtt{new}\, x{:}X$, then there exist Δ, Δ', A and Y such that $\Gamma = \Delta, x{\prec}A, \Delta'$ and $\sigma, \Delta \vdash A{:}Y$ and $X = \langle Y^i + x, Y^o + x, Y^l + x\rangle$.*
2. *If $\sigma, \Gamma \vdash \mathtt{del}\, x{:}X$, then $x \in \sigma$, $x \in \mathsf{dom}(\Gamma)$ and $X = \langle [], [-x], [-x]\rangle$.*
3. *If $\sigma, \Gamma \vdash AB : Z$ with $A, B \neq \epsilon$, then there exist X, Y such that $\sigma, \Gamma \vdash A : X$ and $\sigma + X^l, \Gamma \vdash B{:}Y$ and $Z = \langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l\rangle$.*
4. *If $\sigma, \Gamma \vdash (A + B){:}Z$, then there exist X, Y such that $\sigma, \Gamma \vdash A{:}X$ and $\sigma, \Gamma \vdash B{:}Y$ and $Z = \langle X^i \cup Y^i, X^o \cup Y^o, X^l \cap Y^l\rangle$.*
5. *If $\sigma, \Gamma \vdash (A \parallel B){:}Z$, then there exist X, Y such that $[], \Gamma \vdash A{:}X$ and $[], \Gamma \vdash B{:}Y$, and $Z = \langle X^i + Y^i, X^o + Y^o, X^l + Y^l\rangle$.*
6. *If $\sigma, \Gamma \vdash \{A\}{:}Z$, then there exists X such that $[], \Gamma \vdash A{:}X$ and $Z = \langle X^i, [], []\rangle$.*

Proof. By induction on typing derivations. We will only show the proof for clause 5. The other cases are the same as in the proof of Lemma 3.3.11. The proof for clause 5 is analogous to the proof of Lemma 3.3.11, clause 4.

Note that $\sigma, \Gamma \vdash (A \parallel B){:}Z$ can only be derived by the rule Parallel or WeakenB or WeakenS.

If it is derived by the rule Parallel, then there is only one possibility:

$$\frac{[], \Gamma \vdash A{:}X \quad [], \Gamma \vdash B{:}Y}{[], \Gamma \vdash (A \parallel B){:}\langle X^i + Y^i, X^o + Y^o, X^l + Y^l\rangle}\ (\textsc{Parallel})$$

with $Z = \langle X^i + Y^i, X^o + Y^o, X^l + Y^l\rangle$. The conclusion follows immediately.

If $\sigma, \Gamma \vdash (A \parallel B){:}Z$ is derived by the rule WeakenB:

$$\frac{\sigma, \Gamma' \vdash (A \parallel B){:}Z \quad \sigma_2, \Gamma' \vdash E{:}V \quad x \notin \mathsf{dom}(\Gamma')}{\sigma, \Gamma', x{\prec}E \vdash (A \parallel B){:}Z}\ (\textsc{WeakenB})$$

then we apply the induction hypothesis to $\sigma_1, \Gamma' \vdash (A \parallel B){:}Z$ and then WeakenB.

If it is derived by the rule WeakenS

$$\frac{\sigma', \Gamma \vdash (A \parallel B) : Z \quad \sigma' \subseteq \sigma}{\sigma, \Gamma \vdash (A \parallel B) : Z} \text{ (WeakenS)}$$

then we apply the induction hypothesis to $\sigma', \Gamma \vdash (A \parallel B) : Z$ and then WeakenS.

□

Lemma 4.3.12 (Weakening).

1. *If $\Gamma = \Delta, x \prec E, \Delta'$ is legal, then $\sigma, \Delta \vdash E : X$ for some X, σ.*
2. *If $\sigma, \Gamma \vdash E : X$ and Γ is an initial segment of a legal basis Γ', then $\sigma, \Gamma' \vdash E : X$.*

Proof. The same as in the proof of Lemma 3.3.12. □

Lemma 4.3.13 (Strengthening). *If $\sigma, \Gamma, x \prec A \vdash B : Y$ and $x \notin \mathsf{var}(B)$, then $\sigma, \Gamma \vdash B : Y$ and $x \notin Y^i$.*

Proof. By induction on typing derivations. The proof for the all cases except Parallel is the same as in the proof of Lemma 3.3.13. The case Parallel is proved analogously to the case Seq.

□

Proposition 4.3.14 (Uniqueness of types). *If $\sigma, \Gamma \vdash A : X$ and $\sigma, \Gamma \vdash A : Y$, then $X^i = Y^i$, $X^o = Y^o$, and $X^l = Y^l$.*

Proof. By induction on typing derivations. The proof for all cases except Parallel is the same as in the proof of Lemma 3.3.14. The case Parallel is proved analogously to the case Seq.

□

4.3.3 Soundness Proofs

Since Lemma 4.3.7 is closely related to Lemma 4.3.6, we prove both lemmas together. During the proof, we also show the well-typeness of the new expressions in the new configuration. These results will be used in the proof of Lemma 4.3.4.

Proof of Lemma 4.3.6 and Lemma 4.3.7 (Invariants of retl, retop, io, and maxins). *If $\Gamma \models \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then for all positions $\alpha.k$ in both configurations $\mathbb{T}$ and $\mathbb{T}'$ we have:*

1. $\mathsf{retl}_{\mathbb{T}}(\alpha.k) \subseteq \mathsf{retl}_{\mathbb{T}'}(\alpha.k)$ *if* $\mathbb{T}(\alpha.k) = \mathbb{T}'(\alpha.k)$,
2. $\mathsf{retop}_{\mathbb{T}}(\alpha.k) \supseteq \mathsf{retop}_{\mathbb{T}'}(\alpha.k)$ *if* $\mathbb{T}(\alpha.k) = \mathbb{T}'(\alpha.k)$,
3. $\mathsf{io}_{\mathbb{T}}(\alpha.k) \supseteq \mathsf{io}_{\mathbb{T}'}(\alpha.k)$,
4. *for any valid set of positions $\mathcal{L}'$ of $\mathbb{T}'$ there exists a valid set of positions $\mathcal{L}$ of $\mathbb{T}$ such that*
$$\mathsf{maxins}(\mathbb{T}|_{\mathcal{L}}) \supseteq \mathsf{maxins}(\mathbb{T}'|_{\mathcal{L}'})$$

Proof. The proof proceeds by case analysis on the reduction relation $\longrightarrow$. Suppose that the reduction occurs at position $\beta.n$.

Let $\mathbb{T}(\beta.n) = (M, E_1)$. Since $\mathbb{T}$ is well-typed, there exist X and Γ such that $M + \mathsf{retl}_{\mathbb{T}}(\beta.n), \Gamma \vdash E_1 : X$.

- Case osNew:

$$\text{(osNew)} \quad x \prec A \in \mathit{Decls}$$
$$\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{new}\, xE)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M + x, AE))$$

0. First, we prove that $M + x, \Gamma \vdash AE : Z$ for some Z.

   Since $\beta.n$ is in a leaf stack, $\mathsf{retl}_{\mathbb{T}}(\beta.n) = []$ and we get $M, \Gamma \vdash \mathtt{new}\,x E : X$. By Generation Lemma 4.3.11, clause 3, we have $M, \Gamma \vdash \mathtt{new}\,x : X_1$ and $M + X_1^l, \Gamma \vdash E : X_2$ with

$$X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o, X_1^l + X_2^l \rangle$$

   Also by Lemma 4.3.11, clause 1 applied to $M, \Gamma \vdash \mathtt{new}\,x : X_1$ and by Lemma 4.3.12, we get $M, \Gamma \vdash A : Y$ with $X_1 = \langle Y^i + x, Y^o + x, Y^l + x \rangle$.

   Now sequencing $M, \Gamma \vdash A : Y$ and $M + X_1^l, \Gamma \vdash E : X_2$, we get $M \cup (M + X_1^l - Y^l), \Gamma \vdash AE : Z$ with

$$Z = \langle Y^i \cup (Y^o + X_2^i), Y^o + X_2^o, Y^l + X_2^l \rangle$$

   We still need to show that $M \cup (M + X_1^l - Y^l) \subseteq M + x$. Since $X_1^l = Y^l + x$, we have $M \cup (M + X_1^l - Y^l) = M + x$ and thence the clause holds.

1. The clause holds for all positions $\beta.k$ in any leaf β since by the definition of the function retl, we have $\mathsf{retl}_{\mathbb{T}}(\beta.k) = \mathsf{retl}_{\mathbb{T}'}(\beta.k) = []$. If β is also the root, then the proof completes. Otherwise, for the position $\alpha.k$ at the top of the parent node of β, function $\mathsf{retl}_{\mathbb{T}}(\alpha.k)$ is additive union of $M_1 + Y_1^l + \mathsf{retl}_{\mathbb{T}}(\beta.1) = M_1 + Y_1^l$ and M_2 where $\mathbb{T}(\beta.1) = (M_1, B_1 : Y_1)$ and M_2 is the instances returned from the (possible) other branch of $\alpha.k$. Function $\mathsf{retl}_{\mathbb{T}'}(\alpha.k)$ is calculated in the same way. Therefore, if $n > 1$, then $\mathsf{retl}_{\mathbb{T}'}(\alpha.k)$ is the same as $\mathsf{retl}_{\mathbb{T}}(\alpha.k)$ and the clause holds. If $n = 1$, we need to show that $M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) \subseteq (M + x) + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n)$. We have:

$$\begin{aligned}
M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) &= M + X^l && (\mathsf{retl}_{\mathbb{T}}(\beta.n) = []) \\
&= M + X_1^l + X_2^l && (X^l = X_1^l + X_2^l) \\
&= M + (Y^l + x) + X_2^l && (X_1^l = Y^l + x) \\
&= (M + x) + (Y^l + X_2^l) \\
&= (M + x) + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n) && (\mathsf{retl}_{\mathbb{T}'}(\beta.n) = [])
\end{aligned}$$

   so the clause holds for the position at the top of the parent stack of β. By the recursive definition of function retl and since the two trees $\mathbb{T}$ and $\mathbb{T}'$ are only different at $\beta.n$, the clause follows for all other positions.

2. Analogous to the previous subcase, note that here we also have equality: $M + X^o + \mathsf{retop}_{\mathbb{T}}(\beta.n) = (M + x) + Z^o + \mathsf{retop}_{\mathbb{T}'}(\beta.n)$.

3. The two trees $\mathbb{T}$ and $\mathbb{T}'$ are only different at $\beta.n$, so we only need to prove for this position. For other positions the clause holds by the monotonicity of retop. We have:

$$\begin{aligned}
\mathsf{io}_{\mathbb{T}}(\beta.n) &= M + X^i && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = []) \\
&= M + (X_1^i \cup (X_1^o + X_2^i)) && (X^i = X_1^i \cup (X_1^o + X_2^i)) \\
&= M + ((Y^i + x) \cup ((Y^o + x) + X_2^i)) && (X_1^i = Y^i + x) \\
&= M + x + (Y^i \cup (Y^o + X_2^i)) \\
&= M + x + Z^i && (Z^i = Y^i \cup (Y^o + X_2^i)) \\
&= \mathsf{io}_{\mathbb{T}'}(\beta.n) && (\mathsf{retop}_{\mathbb{T}'}(\beta.n) = [])
\end{aligned}$$

4. We choose $\mathcal{L} = \mathcal{L}'$, then the clause follows by clause 3.

- Case osDel:

$$\begin{array}{l}(\mathsf{osDel}) \quad x \in M \\ \mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{del}\, x E)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M - x, E))\end{array}$$

0. First, we prove that $M - x, \Gamma \vdash E : Z$ for some Z.

   As in the case osNew, since β is a leaf, we get $M, \Gamma \vdash \mathtt{del}\, x E : X$. By Generation Lemma 4.3.11, clause 3, we have $M, \Gamma \vdash \mathtt{del}\, x : X_1$ and $M + X_1^l, \Gamma \vdash E : Z$ with

   $$X = \langle X_1^i \cup (X_1^o + Z^i), X_1^o + Z^o, X_1^l + Z^l \rangle$$

   Also by Lemma 4.3.11, clause 2 applied to $M, \Gamma \vdash \mathtt{del}\, x : X_1$, we have $x \in M$ and $X_1 = \langle [], [-x], [-x] \rangle$. Hence, we get $M - x, \Gamma \vdash E : Z$.

1. By analogous reasoning as in the case osNew, we only need to prove that $M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) \subseteq (M - x) + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n)$. We have:

   $$\begin{aligned} M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) &= M + X_1^l + Z^l && (\mathsf{retl}_{\mathbb{T}}(\beta.n) = []) \\ &= M - x + Z^l && (X_1^l = [-x]) \\ &= (M - x) + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n) && (\mathsf{retl}_{\mathbb{T}'}(\beta.n) = []) \end{aligned}$$

2. Analogous to the previous clause.

3. As in the case osNew, we only need to prove for position $\beta.n$. We have:

   $$\begin{aligned} \mathsf{io}_{\mathbb{T}}(\beta.n) &= M + X^i && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = []) \\ &= M + ([] \cup (Z^i - x)) && (X^i = [] \cup (Z^i - x)) \\ &= M \cup (M + Z^i - x) \\ &\supseteq M + Z^i - x \\ &= \mathsf{io}_{\mathbb{T}'}(\beta.n) && (\mathsf{retop}_{\mathbb{T}'}(\beta.n) = []) \end{aligned}$$

4. We choose $\mathcal{L} = \mathcal{L}'$, then the clause follows by clause 3.

- Case osChoice:

$$\begin{array}{l}(\mathsf{osChoice}) \quad i \in \{1, 2\} \\ \mathsf{Lf}(\mathbf{S} \circ (M, (A_1 + A_2)E)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M, A_i E))\end{array}$$

We treat the case $i = 1$, the case $i = 2$ is symmetric.

0. First, we prove that $M, \Gamma \vdash A_1 E : Z$ for some Z. As in the case osNew, since β is a leaf, we get $M, \Gamma \vdash (A_1 + A_2)E : X$. By Generation Lemma 4.3.11, clause 3 applied to $M, \Gamma \vdash (A_1 + A_2)E : X$, we get $M, \Gamma \vdash (A_1 + A_2) : X_1$ and $M + X_1^l, \Gamma \vdash E : X_2$ with

   $$X = \langle X_1^i \cup (X_1^o + X_2^i), X_1^o + X_2^o, X_1^l + X_2^l \rangle$$

   Also by Lemma 4.3.11, clause 4 applied to $M, \Gamma \vdash (A_1 + A_2) : X_1$, we get $M, \Gamma \vdash A_1 : Y_1$, $M, \Gamma \vdash A_2 : Y_2$ with $X_1 = \langle Y_1^i \cup Y_2^i, Y_1^o \cup Y_2^o, Y_1^l \cap Y_2^l \rangle$. Then we can apply the rule Seq and get $N, \Gamma \vdash A_1 E : Z$ with $N = M \cup (M + X_1^l - Y_1^l)$ and

   $$Z = \langle Y_1^i \cup (Y_1^o + X_2^i), Y_1^o + X_2^o, Y_1^l + X_2^l \rangle$$

   Since $X_1^l = Y_1^l \cap Y_2^l \subseteq Y_1^l$, we get $N = M$ and the clause holds.

1. By similar reasoning as in the case osNew, we only need to prove that $M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) \subseteq M + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n)$. We have:

$$\begin{aligned} M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) &= M + X_1^l + X_2^l && (\mathsf{retl}_{\mathbb{T}}(\beta.n) = []) \\ &= M + (Y_1^l \cap Y_2^l) + X_2^l && (X_1^l = Y_1^l \cap Y_2^l) \\ &\subseteq M + Y_1^l + X_2^l \\ &= M + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n) && (\mathsf{retl}_{\mathbb{T}'}(\beta.n) = []) \end{aligned}$$

2. By similar reasoning as in the case osNew, we only need to prove that $M + X^o + \mathsf{retop}_{\mathbb{T}}(\beta.n) \supseteq M + Z^o + \mathsf{retop}_{\mathbb{T}'}(\beta.n)$. We have:

$$\begin{aligned} M + X^o + \mathsf{retop}_{\mathbb{T}}(\beta.n) &= M + X_1^o + X_2^o && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = []) \\ &= M + (Y_1^o \cup Y_2^o) + X_2^o && (X_1^o = Y_1^o \cup Y_2^o) \\ &\supseteq M + Y_1^o + X_2^o \\ &= M + Z^o + \mathsf{retop}_{\mathbb{T}'}(\beta.n) && (\mathsf{retop}_{\mathbb{T}'}(\beta.n) = []) \end{aligned}$$

3. As in the case osNew, we only need to prove for position $\beta.n$. We have:

$$\begin{aligned} \mathsf{io}_{\mathbb{T}}(\beta.n) &= M + X^i && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = []) \\ &= M + (X_1^i \cup (X_1^o + X_2^i)) && (X^i = X_1^i \cup (X_1^o + X_2^i)) \\ &\supseteq M + (Y_1^i \cup (Y_1^o + X_2^i)) && (X_1^i \supseteq Y_1^i,\ X_1^o \supseteq Y_1^o) \\ &= M + Z^i && (Z^i = Y_1^i \cup (Y_1^o + X_2^i)) \\ &= \mathsf{io}_{\mathbb{T}'}(\beta.n) && (\mathsf{retop}_{\mathbb{T}'}(\beta.n) = []) \end{aligned}$$

4. We choose $\mathcal{L} = \mathcal{L}'$, then the clause follows by clause 3.

- Case osPush:

$$\begin{array}{l} (\mathsf{osPush}) \\ \mathsf{Lf}(\mathbf{S} \circ (M, \{A\}E)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M, E) \circ ([], A)) \end{array}$$

0. First we prove that $[], \Gamma \vdash A : Y$ and $M, \Gamma \vdash E : Z$ for some Y, Z. As in the case osNew, since β is a leaf, we have $M, \Gamma \vdash \{A\}E : X$. By Generation Lemma 4.3.11, clause 3 and 6, we get $M, \Gamma \vdash \{A\} : X_1$ with $X_1^o = X_1^l = []$ and $M, \Gamma \vdash E : Z$ with

$$\begin{aligned} X &= \langle X_1^i \cup (X_1^o + Z^i), X_1^o + Z^o, X_1^l + Z^l \rangle \\ &= \langle X_1^i \cup Z^i, Z^o, Z^l \rangle \end{aligned}$$

Also by Lemma 4.3.11, clause 6 applied to $M, \Gamma \vdash \{A\} : X_1$, we have $[], \Gamma \vdash A : Y$ with $Y^i = X_1^i$.

1. Since $\beta.(n+1) \notin \mathbb{T}$, we do not have to consider this position. For other positions, by similar reasoning as in the case osNew, we only need to prove that $M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) \subseteq M + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n)$. This holds immediately by $X^l = Z^l$ and $\mathsf{retl}_{\mathbb{T}}(\beta.n) = \mathsf{retl}_{\mathbb{T}'}(\beta.n) = []$.
2. Analogous to the previous clause.
3. As in the case osNew, we only need to prove for position $\beta.n$. We have:

$$\begin{aligned} \mathsf{io}_{\mathbb{T}}(\beta.n) &= M + X^i && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = []) \\ &= M + (X_1^i \cup Z^i) && (X^i = X_1^i \cup Z^i) \\ &\supseteq M + Z^i \\ &= \mathsf{io}_{\mathbb{T}'}(\beta.n) && (\mathsf{retop}_{\mathbb{T}'}(\beta.n) = []) \end{aligned}$$

4. If $\beta.(n+1) \in \mathcal{L}'$, we choose $\mathcal{L} = \mathcal{L}' \setminus \{\beta.(n+1)\}$. We have $\beta.n$ is the top of leaf β in both $\mathbb{T}|_{\mathcal{L}}$ and $\mathbb{T}'|_{\mathcal{L}'}$ and the two trees are different only at this position. Therefore the clause follows by clause 3.

   Otherwise we choose $\mathcal{L} = \mathcal{L}'$, then the clause also follows by clause 3.

- Case osPop:

$$\begin{array}{l}(\mathsf{osPop})\\ \mathsf{Lf}(\mathbf{S} \circ (M, E) \circ (M', \epsilon)) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M, E))\end{array}$$

  Since $\beta.n \notin \mathbb{T}'$, we do not have to prove for this position. For other position, the first three clauses hold easily as in the case osNew. The last clause follows by choosing $\mathcal{L} = \mathcal{L}'$.

- Case osParIntr:

$$\begin{array}{l}(\mathsf{osParIntr})\\ \mathsf{Lf}(\mathbf{S} \circ (M, (A \parallel B)E)) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B)))\end{array}$$

0. First, we prove that $[], \Gamma \vdash A : Y_1$ and $[], \Gamma \vdash B : Y_2$, and $M + Y_1^l + Y_2^l, \Gamma \vdash E : Z$ for some Y_1, Y_2, Z.

   As in the case osNew, since β is a leaf of $\mathbb{T}$, we get $M, \Gamma \vdash (A \parallel B)E : X$. By Generation Lemma 4.3.11, clause 3 applied to $M, \Gamma \vdash (A \parallel B)E : X$, we get $M, \Gamma \vdash (A \parallel B) : X_1$ and $M + X_1^l, \Gamma \vdash E : Z$ with

$$X = \langle X_1^i \cup (X_1^o + Z^i), X_1^o + Z^o, X_1^l + Z^l \rangle$$

   Also by Lemma 4.3.11, clause 5 applied to $M, \Gamma \vdash (A \parallel B) : X_1$, we have $[], \Gamma \vdash A : Y_1$, $[], \Gamma \vdash B : Y_2$ with $X_1 = \langle Y_1^i + Y_2^i, Y_1^o + Y_2^o, Y_1^l + Y_2^l \rangle$. Hence, we get the last conclusion $M + Y_1^l + Y_2^l, \Gamma \vdash E : Z$.

1. Since $\beta r.1, \beta l.1 \notin \mathbb{T}$ and $\mathbb{T}(\beta.n) \neq \mathbb{T}'(\beta.n)$, we do not have to consider these positions. For other positions, by similar reasoning as in the case osNew, we only need to prove that $M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) \subseteq M + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n)$. We have:

$$\begin{aligned} M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) &= M + X^l && (\mathsf{retl}_{\mathbb{T}}(\beta.n) = []) \\ &= M + X_1^l + Z^l && (X^l = X_1^l + Z^l) \\ &= M + (Y_1^l + Y_2^l) + Z^l && (X_1^l = Y_1^l + Y_2^l) \\ &= M + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n) && (\mathsf{retl}_{\mathbb{T}'}(\beta.n) = Y_1^l + Y_2^l) \end{aligned}$$

2. Analogous to the previous clause (for we have equality in the previous clause).

3. As in the case osNew, we prove for position $\beta.n$ as follows.

$$\begin{aligned} \mathsf{io}_{\mathbb{T}}(\beta.n) &= M + X^i && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = []) \\ &= M + (X_1^i \cup (X_1^o + Z^i)) && (X^i = X_1^i \cup (X_1^o + Z^i)) \\ &\supseteq M + X_1^o + Z^i \\ &= M + (Y_1^o + Y_2^o) + Z^i && (X_1^o = Y_1^o + Y_2^o) \\ &= M + \mathsf{retop}_{\mathbb{T}'}(\beta.n) + Z^i && (\mathsf{retop}_{\mathbb{T}'}(\beta.n) = Y_1^o + Y_2^o) \\ &= \mathsf{io}_{\mathbb{T}'}(\beta.n) \end{aligned}$$

4. If both $\beta r.1, \beta l.1 \in \mathcal{L}'$ we choose $\mathcal{L} = \mathcal{L}' \setminus \{\beta r.1, \beta l.1\}$. Then the two subtrees $\mathbb{T}|_{\mathcal{L}}$ and $\mathbb{T}'|_{\mathcal{L}'}$ have the same tree structure and they are only different at $\beta.n$, the top of the leaf β of both subtrees. Hence, the clause follows by clause 3.

If $\beta l.1 \in \mathcal{L}'$ and $\beta r.1 \notin \mathcal{L}'$ we choose $\mathcal{L} = \mathcal{L}' \setminus \{\beta l.1\}$. Then $\beta.n \in \mathbb{T}|_{\mathcal{L}}$ and to prove the clause, we need to prove that $\mathsf{io}_{\mathbb{T}}(\beta.n) \supseteq M + \mathsf{io}_{\mathbb{T}'}(\beta r.1)$. We have:

$$\begin{aligned}
\mathsf{io}_{\mathbb{T}}(\beta.n) &= M + X^i && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = []) \\
&= M + (X_1^i \cup (X_1^o + Z^i)) && (X^i = X_1^i \cup (X_1^o + Z^i)) \\
&\supseteq M + X_1^i \\
&\supseteq M + Y_2^i && (X_1^i \supseteq Y_2^i) \\
&= M + \mathsf{io}_{\mathbb{T}'}(\beta r.1) && (\mathsf{io}_{\mathbb{T}'}(\beta r.1) = Y_2^i)
\end{aligned}$$

Case $\beta r.1 \in \mathcal{L}'$ and $\beta l.1 \notin \mathcal{L}'$ is symmetric to the previous subcase.

Otherwise, we choose $\mathcal{L} = \mathcal{L}'$, then $\beta.n \in \mathbb{T}$ and $\beta.n, \beta l.1, \beta r.1 \in \mathbb{T}'$ and the two trees $\mathbb{T}$ and $\mathbb{T}'$ are only different at these positions. Hence, to prove the clause we only need to prove that $\mathsf{io}_{\mathbb{T}}(\beta.n) \supseteq M + \mathsf{io}_{\mathbb{T}'}(\beta l.1) + \mathsf{io}_{\mathbb{T}'}(\beta r.1)$. We have:

$$\begin{aligned}
\mathsf{io}_{\mathbb{T}}(\beta.n) &= M + X^i && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = []) \\
&= M + (X_1^i \cup (X_1^o + Z^i)) && (X^i = X_1^i \cup (X_1^o + Z^i)) \\
&\supseteq M + X_1^i \\
&= M + Y_1^i + Y_2^i && (X_1^i = Y_1^i + Y_2^i) \\
&= M + \mathsf{io}_{\mathbb{T}'}(\beta l.1) + \mathsf{io}_{\mathbb{T}'}(\beta r.1) && (\mathsf{io}_{\mathbb{T}'}(\beta l.1) = Y_1^i, \mathsf{io}_{\mathbb{T}'}(\beta r.1) = Y_2^i)
\end{aligned}$$

- Case osParElimL:

$$\begin{array}{l}
(\mathsf{osParElimL}) \\
\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)), \mathbb{R}) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})
\end{array}$$

Suppose that the location of (M', ϵ) is βl and let $M_o = [\mathbb{T}(\beta r.1)] + X_1^o + \mathsf{retop}_{\mathbb{T}}(\beta r.1)$ and $M_l = [\mathbb{T}(\beta r.1)] + X_1^l + \mathsf{retl}_{\mathbb{T}}(\beta r.1)$ where X_1 is the type of the expression at position $\beta r.1 \in \mathbb{R}$. (βr is the root of $\mathbb{R}$ and M_l (resp. M_o) is minimal (resp. maximal) number of instances returned by $\mathbb{R}$ to $\beta.n$.)

0. First we prove that $(M + M') + \mathsf{retl}_{\mathbb{T}'}(\beta.n), \Gamma \vdash E : X$.
   By the hypothesis we have $M + \mathsf{retl}_{\mathbb{T}}(\beta.n), \Gamma \vdash E : X$. In addition, we have $M + \mathsf{retl}_{\mathbb{T}}(\beta.n) = M + M' + M_l = M + M' + \mathsf{retl}_{\mathbb{T}'}(\beta.n)$. Therefore, we get $M + M' + \mathsf{retl}_{\mathbb{T}'}(\beta.n), \Gamma \vdash E : X$.
1. For $\alpha.k \in \mathbb{R}$, the clause holds easily by the definition of retl and since the two branches from $\alpha.k$ in $\mathbb{T}$ and $\mathbb{T}'$ are the same. For the other positions $\alpha.k \neq \beta.n$, by similar reasoning as in the case osNew, we only need to prove that $M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) \subseteq (M + M') + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n)$. We have:

$$\begin{aligned}
M + X^l + \mathsf{retl}_{\mathbb{T}}(\beta.n) &= M + X^l + M' + M_l && (\mathsf{retl}_{\mathbb{T}}(\beta.n) = M' + M_l) \\
&= (M + M') + X^l + M_l \\
&= (M + M') + Z^l + \mathsf{retl}_{\mathbb{T}'}(\beta.n) \\
& && (\mathsf{retl}_{\mathbb{T}'}(\beta.n) = M_l, X = Z)
\end{aligned}$$

2. Analogous to the previous clause but with M_o instead of M_l.

3. As in the case osNew, we prove for position $\beta.n$ as follows.

$$\begin{aligned}\mathsf{io}_{\mathbb{T}}(\beta.n) &= M + X^i + M' + M_o && (\mathsf{retop}_{\mathbb{T}}(\beta.n) = M_o + M')\\ &= (M + M') + Z^i + M_o && (X = Z)\\ &= (M + M') + Z^i + \mathsf{retop}_{\mathbb{T}'}(\beta.n) && (\mathsf{retop}_{\mathbb{T}'}(\beta.n) = M_o)\\ &= \mathsf{io}_{\mathbb{T}'}(\beta.n)\end{aligned}$$

4. We choose $\mathcal{L} = \mathcal{L}'$, then the clause follows by clause 3.

- Case $\mathsf{osParElimR}$:

$$\begin{array}{l}(\mathsf{osParElimR})\\ \mathsf{Nd}(\mathbf{S} \circ (M, E), \mathbb{R}, \mathsf{Lf}((M', \epsilon))) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})\end{array}$$

  Analogous to case $\mathsf{osParElimL}$.

- Case $\mathsf{osParElim}$:

$$\begin{array}{l}(\mathsf{osParElim})\\ \mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon))) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M + M', E))\end{array}$$

  Analogous to case $\mathsf{osParElimL}$.

□

Proof of Lemma 4.3.4 (Preservation). *If* $\Gamma \models \mathbb{T}$ *and* $\mathbb{T} \longrightarrow \mathbb{T}'$*, then* $\Gamma \models \mathbb{T}'$*.*

Proof. By Definition 4.3.1, we need to prove that for all (M, E) at position $\alpha.k \in \mathbb{T}'$ there exists X such that

$$M + \mathsf{retl}_{\mathbb{T}'}(\alpha.k), \Gamma \vdash E : X$$

In clause 0 of the proof of Lemma 4.3.6, 4.3.7, we have treated all positions $\alpha.k$ such that $\mathbb{T}' \ni \alpha.k \notin \mathbb{T}$ or $\mathbb{T}(\alpha.k) \neq \mathbb{T}'(\alpha.k)$.

For other positions $\alpha.k$, where $\mathbb{T}(\alpha.k)$ and $\mathbb{T}'(\alpha.k)$ are the same, the clause follows by $\Gamma \models \mathbb{T}$, the monotonicity of the function retl, and the typing rule WeakenS. □

Proof of Lemma 4.3.5 (Progress). *If* $\Gamma \models \mathbb{T}$*, then either* $\mathbb{T}$ *is terminal or there exists a configuration* $\mathbb{T}'$ *such that* $\mathbb{T} \longrightarrow \mathbb{T}'$*.*

Proof. Since $\mathbb{T}$ is a well-typed configuration, for expression E at the top of a leaf of $\mathbb{T}$, the function retl at that top position always returns an empty multiset. Therefore, we have $M + [], \Gamma \vdash E : X$, where M is the local store of E. The rest of the proof is analogous to the proof of Lemma 3.3.5.

□

Proof of Theorem 4.3.8 (Soundness). *Let Prog = Decls; E be well-typed, that is,* $\Gamma \vdash E : X$ *for some reordering* Γ *of Decls and some type* X*. Then for any* $\mathbb{T}$ *such that* $\mathsf{Lf}([], E) \longrightarrow^* \mathbb{T}$ *we have that* $\mathbb{T}$ *is not stuck and* $[\mathbb{T}] \subseteq X^i$*.*

Proof. First, configuration $\mathsf{Lf}([], E)$ is well-typed by Definition 4.3.1. Then the first conclusion, $\mathbb{T}$ is not stuck, follows by Lemma 4.3.4 and Lemma 4.3.5. For the second conclusion, since $[\mathbb{T}] \subseteq \mathsf{maxins}(\mathbb{T}|_\emptyset)$ and $\mathsf{maxins}(\mathsf{Lf}([], E)|_\emptyset) = X^i$, the upper bound of instances, $[\mathbb{T}] \subseteq X^i$, follows by Lemma 4.3.7 and the transitivity of $\subseteq$. □

Termination. As in Chapter 2 and 3, all well-typed programs terminate after a finite number of reduction steps. We can prove this property by the same method of Chapter 2, with the function mts extended the new forms of expressions: $\texttt{del}\,x$ and $(A \parallel B)$.

$$\mathsf{mts}(E) = \begin{cases} 1, & \text{if } E = \texttt{del}\,x \\ 3 + \mathsf{mts}(A) + \mathsf{mts}(B), & \text{if } E = (A \parallel B) \end{cases}$$

We need the extra integer 3 in the definition because executing $(A \parallel B)$, let alone A and B, needs one osParIntr, then either osParElimL or osParElimR, and then one osParElim.

The function mts is updated configuration trees as follows.

$$\mathsf{mts}(\mathbb{T}) = \begin{cases} \mathsf{mts}(\mathbf{S}), & \text{if } \mathbb{T} = \mathsf{Lf}(\mathbf{S}) \\ 1 + \mathsf{mts}(\mathbf{S}) + \mathsf{mts}(\mathbb{R}_1), & \text{if } \mathbb{T} = \mathsf{Nd}(\mathbf{S}, \mathbb{R}_1) \\ 2 + \mathsf{mts}(\mathbf{S}) + \mathsf{mts}(\mathbb{R}_1) + \mathsf{mts}(\mathbb{R}_2), & \text{if } \mathbb{T} = \mathsf{Nd}(\mathbf{S}, \mathbb{R}_1, \mathbb{R}_2) \end{cases}$$

The Termination Theorem is the same as in previous chapters. Its proof is the same for the old rules, so we only show the proof for the new cases here.

Theorem 4.3.15 (Termination).

1. *If* $\Gamma \models \mathbb{T}$ *and* $\mathbb{T} \longrightarrow \mathbb{T}'$, *then* $\mathsf{mts}(\mathbb{T}) > \mathsf{mts}(\mathbb{T}')$.
2. *A well-typed program always terminates in a finite number of steps.*

Proof. Suppose that the reduction occurs at position $\beta.n$.

- Case osParIntr:

  $$\begin{array}{l}(\mathsf{osParIntr}) \\ \mathsf{Lf}(\mathbf{S} \circ (M, (A \parallel B)E)) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B)))\end{array}$$

  Since the two trees are different only at positions $\alpha.k \succ \beta.n$ we only need to prove that. $\mathsf{mts}(\mathsf{Lf}(\mathbf{S} \circ (M, (A \parallel B)E))) > \mathsf{mts}(\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B))))$. We have:

  $$\begin{array}{l}\mathsf{mts}(\mathsf{Lf}(\mathbf{S} \circ (M, (A \parallel B)E))) \\ = \mathsf{mts}(\mathbf{S} \circ (M, (A \parallel B)E)) \\ = \mathsf{mts}(\mathbf{S}) + 1 + \mathsf{mts}((A \parallel B)E) \\ = \mathsf{mts}(\mathbf{S}) + 1 + \mathsf{mts}((A \parallel B)) + \mathsf{mts}(E) \\ = \mathsf{mts}(\mathbf{S}) + 1 + 3 + \mathsf{mts}(A) + \mathsf{mts}(B) + \mathsf{mts}(E) \\ > 2 + (\mathsf{mts}(\mathbf{S}) + 1 + \mathsf{mts}(E)) + \mathsf{mts}(A) + \mathsf{mts}(B) \\ = 2 + \mathsf{mts}(\mathbf{S}) + 1 + \mathsf{mts}(E) + \mathsf{mts}(\mathsf{Lf}(([], A)) + \mathsf{mts}(\mathsf{Lf}(([], B)) \\ = 2 + \mathsf{mts}(\mathbf{S} \circ (M, E)) + \mathsf{mts}(\mathsf{Lf}(([], A)) + \mathsf{mts}(\mathsf{Lf}(([], B)) \\ = \mathsf{mts}(\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B))))\end{array}$$

- Case osParElimL:

  $$\begin{array}{l}(\mathsf{osParElimL}) \\ \mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)), \mathbb{R}) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})\end{array}$$

  Analogous to the case osParIntr, we only need to prove that:

  $$\mathsf{mts}(\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)), \mathbb{R})) \;>\; \mathsf{mts}(\mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R}))$$

We have:

$$\begin{aligned}
&\mathsf{mts}(\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)), \mathbb{R})) \\
&= 2 + \mathsf{mts}(\mathbf{S} \circ (M, E)) + \mathsf{mts}(\mathsf{Lf}((M', \epsilon)) + \mathsf{mts}(\mathbb{R}) \\
&= 2 + \mathsf{mts}(\mathbf{S} \circ (M, E)) + \mathsf{mts}(\mathbb{R}) \\
&= 2 + \mathsf{mts}(\mathbf{S} \circ (M + M', E)) + \mathsf{mts}(\mathbb{R}) \\
&> 1 + \mathsf{mts}(\mathbf{S} \circ (M + M', E)) + \mathsf{mts}(\mathbb{R}) \\
&= \mathsf{mts}(\mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R}))
\end{aligned}$$

- Case osParElimR:

$$\begin{array}{l}
(\mathsf{osParElimR}) \\
\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathbb{R}, \mathsf{Lf}((M', \epsilon))) \rightsquigarrow \mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})
\end{array}$$

Symmetric to case osParElimL.

- Case osParElim:

$$\begin{array}{l}
(\mathsf{osParElim}) \\
\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon))) \rightsquigarrow \mathsf{Lf}(\mathbf{S} \circ (M + M', E))
\end{array}$$

Analogous to case osParElimL.

□

Chapter 5

Reuse Instantiation

In the previous three chapters, we have only one instantiation primitive, `new`, for creating a new instance of a component. In this chapter and the next one, we consider, in addition, a conditional instantiation primitive, denoted by `reu` (short for *reuse*). Unlike the primitive `new` which always creates a new instance of a component once executed, the primitive `reu` checks the runtime environment for a reusable instance before creating a new one.

We will continue the goal of controlling the maximum resources that a program needs. However, in this chapter and the next we will explicitly specify the resource bound as a requirement and the type systems will statically check the programs against the requirement so that well-typed programs always respect the resource constraint at runtime.

To simplify the presentation, we exclude the explicit deallocation primitive of Chapters 3 and 4. This chapter extends the basic language of Chapter 2 with the reuse primitive `reu`, and the next chapter will extend the language of this chapter with the parallel composition of Chapter 4. We will discuss the combination of all features in Chapter 7. The type inference algorithm in Chapter 2 can be used to check the well-typedness of programs in this chapter.

5.1 Language

5.1.1 Syntax

Table 5.1 defines the syntax of the language. Now we have two primitives (`new` and `reu`) for creating and (if possible) reusing an instance of a component, and three primitives for composition: sequential, choice and scope. Compared to the basic language in Chapter 2, the only new form of expressions is $\mathtt{reu}\,x$. Other notions of programs and declarations are the same as before.

Table 5.1: Syntax of the language with reuse

$Prog$	$::=$	$Decls; E$	Program
$Decls$	$::=$	$\overline{x \prec E}$	Declarations
E	$::=$		Expression
		ϵ	Empty
	$\mid$	$\mathtt{new}\,x$	New instantiation
	$\mid$	$\mathtt{reu}\,x$	Reuse instantiation
	$\mid$	$E\ E$	Sequencing
	$\mid$	$(E + E)$	Choice
	$\mid$	$\{E\}$	Scope

The following example program will be used to illustrate the operational semantics and typing derivations in the subsequent sections.

$$d \prec \epsilon \quad e \prec \epsilon \quad a \prec \texttt{new}\, d$$
$$b \prec (\, \texttt{reu}\, d\{\, \texttt{new}\, a\} + \texttt{new}\, e\, \texttt{new}\, a)\, \texttt{reu}\, d;$$
$$\texttt{new}\, b$$

In this example, d and e are primitive components. Component a uses one instance of component d. Component b has a choice expression before reuse of an instance of d.

5.1.2 Operational Semantics

Table 5.2 defines a small-step operational semantics. As in Section 2.1.2, the operational semantics is a transition system of configurations and a configuration is a stack of pairs (M, E), where M is a multiset over $\mathbb{C}$, and E is an expression defined in Table 5.1. A configuration is *terminal* if it has the form (M, ϵ) and the multi-step transition relation $\longrightarrow^*$ is the reflective and transitive closure of $\longrightarrow$.

Most of the transition rules in Table 5.2 are the same as in Chapter 2. The rule osNew always adds a new instance to the local store, the rule osChoice selects one branch to execute, the rule osPush adds an element to the top of the leaf stack, and the rule osPop only removes the element at the top of the stack when the stack has at least two elements.

Table 5.2: Transition rules of the language with reuse

(osNew) if $x \prec A \in \mathit{Decls}$
$$\mathbf{S} \circ (M, \texttt{new}\, xE) \longrightarrow \mathbf{S} \circ (M + x, AE)$$

(osReu1) if $x \prec A \in \mathit{Decls}$ $x \notin [\mathbf{S}] + M$
$$\mathbf{S} \circ (M, \texttt{reu}\, xE) \longrightarrow \mathbf{S} \circ (M + x, AE)$$

(osReu2) if $x \prec A \in \mathit{Decls}$ $x \in [\mathbf{S}] + M$
$$\mathbf{S} \circ (M, \texttt{reu}\, xE) \longrightarrow \mathbf{S} \circ (M, AE)$$

(osChoice) $i \in \{1, 2\}$
$$\mathbf{S} \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S} \circ (M, A_i E)$$

(osPush)
$$\mathbf{S} \circ (M, \{A\}E) \longrightarrow \mathbf{S} \circ (M, E) \circ ([], A)$$

(osPop)
$$\mathbf{S} \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S} \circ (M, E)$$

Regarding the operational semantics of the primitive `reu`, first, we need the notion of *reusable instances*. For the moment, the reusable instances for an expression at the top of a stack are all the active instances in the stack. For instance, the reusable instances for E_n at the top of the stack $\mathbf{S} = (M_1, E_1) \circ \cdots \circ (M_n, E_n)$ are the multiset $[\mathbf{S}] = \biguplus_{j=1}^{n} M_j$.

The primitive `reu` has two transition rules: osReu1 and osReu2. When instantiating a component x by this primitive, first, we look for a reusable instance of x in the current configuration. If there is at least one x found, then, by the rule osReu2, we skip creating a new instance of x but continue executing the body A of x. Otherwise, by the rule osReu1, we create a new instance of x in the local store and then continue executing the body A of x—the same as the rule osNew.

Note that in this model we take an abstract view on the reusability, like the view on deleting an instance in Chapters 3 and 4. We care neither about how many times a component is reused, nor about which specific instance is reused (if there are more than

one). So the reusable instances can be thought of as a set—we only need to know whether there is a reusable instance or not.

The example at the end of Section 5.1.1 is used to illustrate the operational semantics. For this simple example program, there are only two possible runs. We show one of the possible runs.

$$
\begin{array}{rll}
& ([], \texttt{new}\, b) & \text{(startup)} \\
(\mathsf{osNew}) \longrightarrow & ([b], (\,\texttt{reu}\, d\{\,\texttt{new}\, a\} + \texttt{new}\, e\, \texttt{new}\, a)\, \texttt{reu}\, d) & \\
(\mathsf{osChoice}) \longrightarrow & ([b], \texttt{reu}\, d\{\,\texttt{new}\, a\}\, \texttt{reu}\, d) & \text{(a new } d \text{ is created)} \\
(\mathsf{osReu1}) \longrightarrow & ([b, d], \{\,\texttt{new}\, a\}\, \texttt{reu}\, d) & \\
(\mathsf{osPush}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([], \texttt{new}\, a) & \\
(\mathsf{osNew}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a], \texttt{new}\, d) & \\
(\mathsf{osNew}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a, d], \epsilon) & \text{(two } d\text{s)} \\
(\mathsf{osPop}) \longrightarrow & ([b, d], \texttt{reu}\, d) & (d \text{ is reused}) \\
(\mathsf{osReu2}) \longrightarrow & ([b, d], \epsilon) & \text{(terminal)}
\end{array}
$$

There are two $\texttt{reu}\, d$'s in the first execution and only the first one creates an instance of d. The maximum number of simultaneously active d's is two.

5.2 Type System

Unlike the previous three chapters where the resource bound errors are not explicitly described and the type systems then find the upper bound of resources of a program, here we will explicitly state the resource bound errors as a resource requirement. Then we develop a type system that, besides checking for missing declarations and loops in declarations, checks the resource constraint in the typing rules. We start by describing the resource constraint.

A requirement $\mathcal{R}$ states that some components in $\mathbb{C}$ can have at most a certain number of instances at runtime. So $\mathcal{R}$ can be viewed as a total map from $\mathbb{C}$ to $\mathbb{N} \cup \{\infty\}$, where $\mathbb{N}$ is the set of (positive) natural numbers and $\mathcal{R}(x) \in \mathbb{N}$ is the maximum allowed number of x's instances; other components which can have an unlimited number of instances are mapped to ∞. Be definition $n < \infty$ for all $n \in \mathbb{N}$. Since $\mathcal{R}(x) = 0$ means that x must not be used by the program and this property can be easily checked, we assume that $\mathcal{R}(x) > 0$. For a multiset M, we denote $M \subseteq \mathcal{R}$ when $M(x) \leq \mathcal{R}(x)$ for all $x \in M$.

Given a requirement $\mathcal{R}$, we say that a component program causes an error with respect to $\mathcal{R}$ if at some execution state of the program, the number of simultaneously active instances of a component x is greater than the allowed number $\mathcal{R}(x)$. In other words, a well-behaved program satisfies that, at any state during its execution, the number of simultaneously active instances of any component is smaller than or equals the allowed number.

Next, we define types, typing judgments, typing relation, and give some typing examples. Types of component expressions now are tuples of four finite multisets over the set of component names.

Definition 5.2.1 (Types). *Types of component expressions are tuples*

$$X = \langle X^i, X^o, X^j, X^p \rangle$$

where X^i, X^o, X^j and X^p are finite multisets over $\mathbb{C}$.

Let us explain informally these multisets. The first two multisets X^i and X^o have the same meaning as in Chapter 2. That is, X^i is the upper bound of the number of instances of any component *during* the execution of the expression, and X^o is the upper bound of the number of instances that are still active *after* the execution of the expression.

The next two multisets X^j and X^p express the same bounds as X^i and X^o, respectively, but with respect to executing the expression in a state where every component already has at least one reusable instance. The reason is that the semantics of reusing instances of components depends on whether there is already such an instance or not. More concretely, in a sequential composition EE' the behaviour of `reu`'s in E' depends on the instances that are active *after* the execution of E, which would violate *compositionality*—the principle that the type of an expression can be computed from the types of its subexpressions. In order to save compositionality, we need two more multisets X^j and X^p in the types. The compositionality of types will be clearer in the explanation of the typing rule SEQ below. Now we have to prepare with some preliminary definitions.

As in Chapter 2, a *basis* or an *environment*, ranged over by Γ, Δ, is a list of declarations and $\mathsf{dom}(\Gamma)$ is the set of variables occurring in Γ. A typing judgment is a tuple of the form:

$$\Gamma \vdash_{\mathcal{R}} A : X$$

and it asserts that expression A has type X in the environment Γ, with respect to the requirement $\mathcal{R}$. We ignore the subscript $\mathcal{R}$ in the typing judgment when $\mathcal{R}$ is clear from the context or some requirement $\mathcal{R}$ is assumed in the context.

Definition 5.2.2 (Valid typing judgments). *Let $\mathcal{R}$ be a requirement. Valid typing judgments $\Gamma \vdash_{\mathcal{R}} A : X$ are derived by applying the typing rules in Table 5.3 in the usual inductive way.*

Table 5.3: Typing rules of the language with reuse

(AXIOM)
$$\frac{}{\varnothing \vdash \epsilon : \langle [], [], [], [] \rangle}$$

(WEAKENB)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec B \vdash A : X}$$

(NEW)
$$\frac{\Gamma \vdash A : X \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{new}\, x : \langle X^i + x, X^o + x, X^j + x, X^p + x \rangle}$$

(REU)
$$\frac{\Gamma \vdash A : X \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{reu}\, x : \langle X^i + x, X^o + x, X^j, X^p \rangle}$$

(SEQ)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad X^o + Y^j \subseteq \mathcal{R} \quad A, B \neq \epsilon}{\Gamma \vdash AB : \langle X^i \cup (X^o + Y^j) \cup Y^i, (X^o + Y^p) \cup Y^o, X^j \cup (X^p + Y^j), X^p + Y^p \rangle}$$

(CHOICE)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y}{\Gamma \vdash (A + B) : \langle X^i \cup Y^i, X^o \cup Y^o, X^j \cup Y^j, X^p \cup Y^p \rangle}$$

(SCOPE)
$$\frac{\Gamma \vdash A : X}{\Gamma \vdash \{A\} : \langle X^i, [], X^j, [] \rangle}$$

Some further explanation of these typing rules is described next. The most critical rule is SEQ because sequencing two expressions can lead to an increase in instances of the composed expression. Let us start with the first and the third part of type expression for AB. After expression A has been executed, there are at most $X^o(x)$ instances of component x. Executing B can create at most $Y^i(x)$ instances of x if x is not in system state which is X^o. Otherwise $Y^j(x)$ instances of x will be created, meaning that there are at most $((X^o + Y^j) \cup Y^i)(x)$ instances of x after the execution of A and during the execution of B. So we require the side condition $X^o + Y^j \subseteq \mathcal{R}$. In addition, because during executing A there are at most $X^i(x)$ instances of x created, the first part of type of AB is the maximum of $X^i(x)$ and $((X^o + Y^j) \cup Y^i)(x)$. After executing AB it is easy to see that the surviving instances are total of those from A and B if we start from state with no instance of any component. By similar reasoning when we start with a

stack containing at least one instance of every component we can calculate the second and the last parts in the type expression for AB and the whole type expression of AB is $\langle X^i \cup (X^o + Y^j) \cup Y^i, (X^o + Y^p) \cup Y^o, X^j \cup (X^p + Y^j), X^p + Y^p \rangle$.

Other typing rules are straightforward. The rule Axiom requires no premise and is used to take off. The rules New and Reu allow us to type expressions $\texttt{new}\,x$ and $\texttt{reu}\,x$, respectively. The side condition $x \notin \mathsf{dom}(\Gamma)$ prevents ambiguity and circularity. The rule WeakenB is used to expand bases so that we can combine typings in other rules: Seq, Choice. The rules Choice and Scope are easy to understand recalling the corresponding rules osChoice and osScope of the operational semantics.

The definition of *well-typed programs* is almost the same as before.

Definition 5.2.3 (Well-typed programs). *Program $Prog = Decls; E$ is well-typed with respect to a requirement $\mathcal{R}$ if there exist a reordering Γ of declarations in Decls and a type X such that $\Gamma \vdash_{\mathcal{R}} E : X$.*

We end the section by giving some typing derivations for expressions in the sample program in Section 5.1.1. Assume that $\mathcal{R} = \{a \mapsto 2, b \mapsto 2, e \mapsto 3, c \mapsto \infty, d \mapsto \infty\}$, which means that components a, b can have at most two simultaneously active instances, component e can have at most three, and components c, d can have an unlimited number. Note that we omitted some side conditions as they can be checked easily and we shortened the rule names. The rule Axiom is also simplified.

$$\textsc{Wea}\,\dfrac{\textsc{New}\,\dfrac{\varnothing \vdash \epsilon : \langle [], [], [], [] \rangle}{d \mathrel{-\!\!\prec} \epsilon \vdash \texttt{reu}\,d : \langle [d], [d], [], [] \rangle} \qquad \textsc{New}\,\dfrac{\varnothing \vdash \epsilon : \langle [], [], [], [] \rangle}{d \mathrel{-\!\!\prec} \epsilon \vdash \texttt{new}\,d : \langle [d], [d], [d], [d] \rangle}}{d \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} \texttt{new}\,d \vdash \texttt{reu}\,d : \langle [d], [d], [], [] \rangle}$$

$$\textsc{Sco}\,\dfrac{\textsc{New}\,\dfrac{\textsc{New}\,\dfrac{\varnothing \vdash \epsilon : \langle [], [], [], [] \rangle}{d \mathrel{-\!\!\prec} \epsilon \vdash \texttt{new}\,d : \langle [d], [d], [d], [d] \rangle}}{d \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} \texttt{new}\,d \vdash \texttt{new}\,a : \langle [a,d], [a,d], [a,d], [a,d] \rangle}}{d \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} \texttt{new}\,d \vdash \{\,\texttt{new}\,a\} : \langle [a,d], [], [a,d], [] \rangle}$$

Sequencing the above two derivations we have:

$$d \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} \texttt{new}\,d \vdash \texttt{reu}\,d\{\,\texttt{new}\,a\} : \langle [a,d,d], [d], [a,d], [] \rangle$$

We can weaken the above derivation to get:

$$\Gamma \vdash \texttt{reu}\,d\{\,\texttt{new}\,a\} : \langle [a,d,d], [d], [a,d], [] \rangle$$

where $\Gamma = d \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} \texttt{new}\,d, e \mathrel{-\!\!\prec} \epsilon$. We can also derive:

$$\textsc{Seq}\,\dfrac{\dfrac{\cdots}{\Gamma \vdash \texttt{new}\,e : \langle [e], [e], [e], [e] \rangle} \qquad \dfrac{\cdots}{\Gamma \vdash \texttt{new}\,a : \langle [a,d], [a,d], [a,d], [a,d] \rangle}}{\Gamma \vdash \texttt{new}\,e\,\texttt{new}\,a : \langle [a,d,e], [a,d,e], [a,d,e], [a,d,e] \rangle}$$

and with $\Gamma' = \Gamma, b \mathrel{-\!\!\prec} (\,\texttt{reu}\,d\{\,\texttt{new}\,a\} + \texttt{new}\,e\,\texttt{new}\,a)\,\texttt{reu}\,d$, we have:

$$\Gamma' \vdash \texttt{new}\,b : \langle [a,b,d,d,e], [a,b,d,e], [a,b,d,e], [a,b,d,e] \rangle$$

In this example, expression $\texttt{new}\,b$ is typable with respect to the given requirement and the example program is well-typed with respect to the requirement. If $\mathcal{R}(d) = 1$, then the example program would not be typable as the side condition when sequencing $\texttt{reu}\,d$ and $\{\,\texttt{new}\,a\}$ would not be satisfied.

5.3 Properties

5.3.1 Type Soundness

As in the previous chapters, we will define the notion of *well-typed configuration* and prove two main lemmas: Preservation and Progress.

As before, we denote by $\mathsf{hi}(\mathbf{S})$ the height of the stack $\mathbf{S}$, by $\mathbf{S}(k)$ the pair at position k, by $[\mathbf{S}(k)]$ the store M at position k, by $[\mathbf{S}]$ the additive union of all stores in the stack, and by $\mathbf{S}|_k$ the stack from the bottom of $\mathbf{S}$ up to k.

In this system, type errors occur when a program tries to instantiate a component x but there is no declaration of x or when a configuration violates requirement $\mathcal{R}$, that is, there exists a component x whose number of active instances is greater than the allowed number, $\mathcal{R}(x)$. The latter error makes the definition of well-typed configurations a little different from the ones in the previous three chapters. As we have specified explicitly the resource bound requirement, the definition of well-typed configurations now contains the resource constraint.

Definition 5.3.1 (Well-typed configurations). *Configuration* $\mathbf{S} = (M_1, E_1) \circ \cdots \circ (M_n, E_n)$ *is well-typed with respect to a basis* Γ *and a requirement* $\mathcal{R}$*, notation* $\Gamma \models_{\mathcal{R}} \mathbf{S}$*, if for all* $1 \leq k \leq n$*, we have* $\Gamma \vdash_{\mathcal{R}} E_k : X_k$ *and*

$$[\mathbf{S}|_k] + X_k^j \subseteq \mathcal{R}$$

In the definition, to be more precise, the inequality should be $([\mathbf{S}|_k] + X_k^j) \cup X_k^i \subseteq \mathcal{R}$, but we know that $X_k^i \subseteq \mathcal{R}$ holds by Lemma 5.3.7 below. Therefore the inequality is simplified.

The formal definitions of terminal configurations and stuck states are the same as in previous chapters.

Definition 5.3.2 (Terminal configurations). *A configuration* $\mathbf{S}$ *is terminal if it has the form* (M, ϵ).

Definition 5.3.3 (Stuck states). *A configuration* $\mathbf{S}$ *is stuck if no transition rule applies and* $\mathbf{S}$ *is not terminal.*

The two standard lemmas Preservation and Progress are stated as follows.

Lemma 5.3.4 (Preservation). *If* $\Gamma \models_{\mathcal{R}} \mathbf{S}_1$ *and* $\mathbf{S}_1 \longrightarrow \mathbf{S}_2$*, then* $\Gamma \models_{\mathcal{R}} \mathbf{S}_2$.

Lemma 5.3.5 (Progress). *If* $\Gamma \models_{\mathcal{R}} \mathbf{S}$*, then either* $\mathbf{S}$ *is terminal or there exists configuration* $\mathbf{S}'$ *such that* $\mathbf{S} \longrightarrow \mathbf{S}'$.

Finally, the following theorem guarantees that well-typed programs are safe to execute. That is, during the execution of the programs the number of simultaneously active instances of any component never exceeds the allowed number.

Theorem 5.3.6 (Soundness). *If program* $Prog = Decls; E$ *is well-typed with respect to a requirement* $\mathcal{R}$*, then for any* $\mathbf{S}$ *such that* $([], E) \longrightarrow^* \mathbf{S}$ *we have* $\mathbf{S}$ *is not stuck and* $[\mathbf{S}] \subseteq \mathcal{R}$.

5.3.2 Typing Properties

This section lists some properties of the type system. These properties are analogous to the properties of the previous chapters. We start by updating some definitions.

The notions on bases: *legal*, *initial segment*, and *is in* are the same as in Section 2.3.2. We use X^* for any of X^i, X^o, X^j and X^p. The function var is updated from the definition in Section 2.3.2 as follows.

$$\mathsf{var}(\,\mathtt{reu}\, x) = \{x\}$$

The following lemma collects a number of simple properties of a valid typing judgment. It also shows some relations among multisets of types and any legal basis always has distinct declarations. The inclusion relations between multisets of a type expression can be visualized by the following picture. The arrows point to smaller multisets.

$$\begin{array}{ccccc} & & X^i & & \\ & \swarrow & \downarrow & \searrow & \\ X^o & \longrightarrow & X^p & \longleftarrow & X^j \end{array}$$

Lemma 5.3.7 (Valid typing judgment). *If* $\Gamma \vdash_{\mathcal{R}} A\!:\!X$*, then*

1. $\mathsf{var}(A) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{dom}(X^*) \subseteq \mathsf{dom}(\Gamma)$,
2. $\Gamma \vdash \epsilon\!:\!\langle [], [], [], [] \rangle$,
3. *every variable in* $\mathsf{dom}(\Gamma)$ *is declared only once in* Γ*,*
4. $X^o \subseteq X^i \subseteq \mathcal{R}$ *and* $X^p \subseteq X^j \subseteq \mathcal{R}$*,*
5. $0 \leq X^i(z) - X^j(z) \leq 1$ *and* $0 \leq X^o(z) - X^p(z) \leq 1$ *for all* z*.*

Proof. By induction on typing derivations.

- Base case Axiom: Since $\mathsf{var}(\epsilon) = \mathsf{dom}([]) = \mathsf{dom}(\varnothing) = \{\}$, the clause holds.
- Case WeakenB:

$$\text{(WeakenB)}\ \frac{\Gamma' \vdash A\!:\!X \quad \Gamma' \vdash B\!:\!Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x\!\prec\! B \vdash A\!:\!X}$$

  Clause 3 follows by the side condition and the induction hypothesis. The other clauses follow by the induction hypothesis.

- Case New:

$$\text{(New)}\ \frac{\Gamma' \vdash B\!:\!Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x\!\prec\! B \vdash \mathtt{new}\, x\!:\!\langle Y^i + x, Y^o + x, Y^j + x, Y^p + x\rangle}$$

  with $\Gamma = \Gamma', x\!\prec\! B$, $X = \langle Y^i + x, Y^o + x, Y^j + x, Y^p + x\rangle$.

  The first conclusion of clause 1 holds easily since $\mathsf{var}(\mathtt{new}\, x) = \{x\} \subseteq \mathsf{dom}(\Gamma') \cup \{x\} = \mathsf{dom}(\Gamma)$. The second one follows by the induction hypothesis $\mathsf{dom}(Y^*) \subseteq \mathsf{dom}(\Gamma')$ and $\mathsf{dom}(X^*) = \mathsf{dom}(Y^*) \cup \{x\} \subseteq \mathsf{dom}(\Gamma') \cup \{x\} = \mathsf{dom}(\Gamma)$. Clause 2, $\Gamma', x\!\prec\! B \vdash \epsilon\!:\! \langle [], [], [], [] \rangle$, follows by applying WeakenB:

$$\text{(WeakenB)}\ \frac{\Gamma' \vdash \epsilon\!:\!\langle [], [], [], [] \rangle \quad \Gamma' \vdash B\!:\!Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x\!\prec\! B \vdash \epsilon\!:\!\langle [], [], [], [] \rangle}$$

  Clause 3 follows by the side condition $x \notin \mathsf{dom}(\Gamma')$ and the induction hypothesis. Clause 4 follows by the induction hypothesis. Clause 5 holds by the induction hypothesis and $x \in X^*$.

- Case Reu:

$$\text{(Reu)}\ \frac{\Gamma' \vdash B\!:\!Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x\!\prec\! B \vdash \mathtt{reu}\, x\!:\!\langle Y^i + x, Y^o + x, Y^j, Y^p\rangle}$$

  with $\Gamma = \Gamma', x\!\prec\! B$ and $X = \langle Y^i + x, Y^o + x, Y^j, Y^p\rangle$. The last clause follows by the induction hypothesis, $x \notin Y^*$ and $x \in X^i$, $x \in X^o$, $x \notin X^j$, $x \notin X^p$. The other clauses are proved as in the case New.

- Case SEQ:

$$\text{(SEQ)}\ \frac{\Gamma \vdash B\!:\!Y \quad \Gamma \vdash C\!:\!Z \quad Y^o + Z^j \subseteq \mathcal{R} \quad B, C \neq \epsilon}{\Gamma \vdash BC\!:\!\langle Y^i \cup (Y^o + Z^j) \cup Z^i, (Y^o + Z^p) \cup Z^o, Y^j \cup (Y^p + Z^j), Y^p + Z^p\rangle}$$

Clauses 1, 2 and 3 hold by the induction hypothesis. For clause 4, we have $(Y^o + Z^p) \cup Z^o \subseteq Y^i \cup (Y^o + Z^j) \cup Z^i \subseteq \mathcal{R}$ since $Z^p \subseteq Z^j$, $Z^o \subseteq Z^i$ by C well-typed and $Y^o + Z^j \subseteq \mathcal{R}$ by side condition, $Y^i \subseteq \mathcal{R}, Z^i \subseteq \mathcal{R}$ by the induction hypothesis. Similarly, $Y^p + Z^p \subseteq Y^j \cup (Y^p + Z^j) \subseteq \mathcal{R}$ holds since $Z^p \subseteq Z^j$ and $Y^p + Z^j \subseteq Y^o + Z^j \subseteq \mathcal{R}$.

For clause 5, since $Y^j \subseteq Y^i$ and $Z^p \subseteq Z^o$, we get $0 \leq X^i(z) - X^j(z)$ for all z. In addition,

$$X^i(z) - X^j(z) = max \left\{ \begin{array}{l} Y^i(z) - (Y^j \cup (Y^p + Z^j))(z), \\ (Y^o + Z^j)(z) - (Y^j \cup (Y^p + Z^j))(z), \\ Z^i(z) - (Y^j \cup (Y^p + Z^j))(z) \end{array} \right\}$$

each of the three cases is less then or equals 1 so $X^i(z) - X^j(z) \leq 1$. Similarly, it is easy to see that $0 \leq X^o(z) - X^p(z) = (Y^o + Z^p) \cup Z^o)(z) - (Y^p + Z^p)(z) \leq 1$ for all z.

- Case CHOICE:

$$\text{(CHOICE)}\ \frac{\Gamma \vdash C\!:\!Z \quad \Gamma \vdash B\!:\!Y}{\Gamma \vdash (C + B)\!:\!\langle Z^i \cup Y^i, Z^o \cup Y^o, Z^j \cup Y^j, Z^p \cup Y^p\rangle}$$

Analogous to case SEQ, the first three clauses hold by the induction hypothesis.

Clause 4 follows by the induction hypothesis and the definition of multiset union. Clause 5 is also easy for the first half:

$$X^i(z) - X^j(z) = (Y^i \cup Z^i)(z) - (Y^j \cup Z^j)(z) \geq 0$$

For the second half, if $Y^i \supseteq Z^i$, then $X^i(z) - X^j(z) = Y^i(z) - (Y^j \cup Z^j)(z) \leq Y^i(z) - Y^j(z) \leq 1$. The other way around, $Z^i \supseteq Y^i$, is the same. The second part, $0 \leq X^o(z) - X^p(z) \leq 1$, is analogous.

- Case SCOPE:

$$\text{(SCOPE)}\ \frac{\Gamma \vdash B\!:\!Y}{\Gamma \vdash \{B\}\!:\!\langle Y^i, [], Y^j, []\rangle}$$

All clauses hold by the induction hypothesis or are trivial.

□

Lemma 5.3.8 (Associativity). *If $\Gamma \vdash_{\mathcal{R}} A_i : X_i$, for $i \in \{1, 2, 3\}$, then the typing judgments for $(A_1A_2)A_3$ and $A_1(A_2A_3)$, if typable, are the same.*

Proof.

$$\text{(SEQ)}\ \frac{\Gamma \vdash A_1\!:\!X_1 \quad \Gamma \vdash A_2\!:\!X_2 X_1^o + X_2^j \subseteq \mathcal{R} \quad A_1, A_2 \neq \epsilon}{\Gamma \vdash A_1A_2\!:\!\langle X_1^i \cup (X_1^o + X_2^j) \cup X_2^i, (X_1^o + X_2^p) \cup X_2^o, X_1^j \cup (X_1^p + X_2^j), X_1^p + X_2^p\rangle}$$

By the rule SEQ, we have $\Gamma \vdash A_1A_2 : Y$ and

$$Y = \langle X_1^i \cup (X_1^o + X_2^j) \cup X_2^i, (X_1^o + X_2^p) \cup X_2^o, X_1^j \cup (X_1^p + X_2^j), X_1^p + X_2^p\rangle$$

Similarly, we have $\Gamma \vdash A_2A_3 : Z$ and

$$Z = \langle X_2^i \cup (X_2^o + X_3^j) \cup X_3^i, (X_2^o + X_3^p) \cup X_3^o, X_2^j \cup (X_2^p + X_3^j), X_2^p + X_3^p \rangle$$

Continue to apply the rule Seq, we get the typing judgments for $(A_1A_2)A_3$ and $A_1(A_2A_3)$. Then to prove that the two judgments are the same, we need to prove the following equations:

$$\begin{aligned} Y^i \cup (Y^o + X_3^j) \cup X_3^i &= X_1^i \cup (X_1^o + Z^j) \cup Z^i \\ (Y^o + X_3^p) \cup X_3^o &= (X_1^o + Z^p) \cup Z^o \\ Y^j \cup (Y^p + X_3^j) &= X_1^j \cup (X_1^p + Z^j) \\ Y^p + X_3^p &= X_1^p + Z^p \end{aligned}$$

For the first one, we have:

$$\begin{aligned} & Y^i \cup (Y^o + X_3^j) \cup X_3^i \\ &= (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i) \cup (((X_1^o + X_2^p) \cup X_2^o) + X_3^j) \cup X_3^i \\ &= X_1^i \cup X_2^i \cup X_3^i \cup (X_1^o + X_2^j) \cup (((X_1^o + X_2^p) \cup X_2^o) + X_3^j) \\ &= X_1^i \cup X_2^i \cup X_3^i \cup (X_1^o + X_2^j) \cup (X_1^o + X_2^p + X_3^j) \cup (X_2^o + X_3^j) \\ &= X_1^i \cup X_2^i \cup X_3^i \cup (X_1^o + (X_2^j \cup (X_2^p + X_3^j))) \cup (X_2^o + X_3^j) \\ &= X_1^i \cup (X_1^o + (X_2^j \cup (X_2^p + X_3^j))) \cup (X_2^i \cup (X_2^o + X_3^j) \cup X_3^i) \\ &= X_1^i \cup (X_1^o + Z^j) \cup Z^i \end{aligned}$$

For the second one, we have:

$$\begin{aligned} (Y^o + X_3^p) \cup X_3^o &= (((X_1^o + X_2^p) \cup X_2^o) + X_3^p) \cup X_3^o \\ &= (X_1^o + X_2^p + X_3^p) \cup (X_2^o + X_3^p) \cup X_3^o \\ &= (X_1^o + (X_2^p + X_3^p)) \cup ((X_2^o + X_3^p) \cup X_3^o) \\ &= (X_1^o + Z^p) \cup Z^o \end{aligned}$$

For the third one, we have:

$$\begin{aligned} Y^j \cup (Y^p + X_3^j) &= (X_1^j \cup (X_1^p + X_2^j)) \cup ((X_1^p + X_2^p) + X_3^j) \\ &= X_1^j \cup (X_1^p + X_2^j) \cup (X_1^p + X_2^p + X_3^j) \\ &= X_1^j \cup (X_1^p + (X_2^j \cup (X_2^p + X_3^j))) \\ &= X_1^j \cup (X_1^p + Z^j) \end{aligned}$$

The last equation follows easily:

$$\begin{aligned} Y^p + X_3^p &= (X_1^p + X_2^p) + X_3^p \\ &= X_1^p + (X_2^p + X_3^p) \\ &= X_1^p + Z^p \end{aligned}$$

□

Lemma 5.3.9 (Generation).

1. *If* $\Gamma \vdash \texttt{new}\, x : X$, *then* $x \in X^p$ *and there exist* Δ, Δ', A *and* Y *such that* $\Gamma = \Delta, x \prec A, \Delta'$ *and* $\Delta \vdash A : Y$ *and* $X = \langle Y^i + x, Y^o + x, Y^j + x, Y^p + x \rangle$.

2. *If* $\Gamma \vdash \texttt{reu}\, x : X$, *then* $x \in X^o$ *and there exist* Δ, Δ', A *and* Y *such that* $\Gamma = \Delta, x \prec A, \Delta'$ *and* $\Delta \vdash A : Y$ *and* $X = \langle Y^i + x, Y^o + x, Y^j, Y^p \rangle$.

3. *If* $\Gamma \vdash AB : Z$ *with* $A, B \neq \epsilon$*, then there exist* X *and* Y *such that* $\Gamma \vdash A : X$ *and* $\Gamma \vdash B{:}Y$ *and* $Z = \langle X^i \cup (X^o + Y^j) \cup Y^i, (X^o + Y^p) \cup Y^o, X^j \cup (X^p + Y^j), X^p + Y^p\rangle$.
4. *If* $\Gamma \vdash (A + B){:}Z$*, then there exist* X *and* Y *such that* $\Gamma \vdash A{:}X$ *and* $\Gamma \vdash B{:}Y$ *and* $Z = \langle X^i \cup Y^i, X^o \cup Y^o, X^j \cup Y^j, X^p \cup Y^p\rangle$.
5. *If* $\Gamma \vdash \{A\}{:}Z$*, then there exists* X *such that* $\Gamma \vdash A{:}X$ *and* $Z = \langle X^i, [], X^j, []\rangle$.

Proof. The proof is analogous to the proof of Lemma 2.3.11. □

Lemma 5.3.10 (Weakening).

1. *If* $\Gamma = \Delta, x{\prec}E, \Delta'$ *is legal, then* $\Delta \vdash E{:}X$ *for some* X.
2. *If* $\Gamma \vdash_{\mathcal{R}} E{:}X$ *and* Γ *is an initial segment of a legal basis* Γ'*, then* $\Gamma' \vdash_{\mathcal{R}} E{:}X$.

Proof.

1. The only way to extend Δ to $\Delta, x{\prec}E$ in a derivation is by applying the rule New, Reu or WeakenB.

$$\text{(New)}\quad \frac{\Delta \vdash E{:}X \quad x \notin \mathsf{dom}(\Delta)}{\Delta, x{\prec}E \vdash \mathtt{new}\, x{:}\langle X^i + x, X^o + x, X^j + x, X^p + x\rangle}$$

$$\text{(Reu)}\quad \frac{\Delta \vdash E{:}X \quad x \notin \mathsf{dom}(\Delta)}{\Delta, x{\prec}E \vdash \mathtt{reu}\, x{:}\langle X^i + x, X^o + x, X^j, X^p\rangle}$$

$$\text{(WeakenB)}\quad \frac{\Delta \vdash E{:}X \quad \Delta \vdash B{:}Y \quad x \notin \mathsf{dom}(\Delta)}{\Delta, x{\prec}E \vdash B{:}Y}$$

   Each of the rules has $\Delta \vdash E{:}X$ as a premise.

2. The proof is the same as the proof of Lemma 2.3.12.

□

Lemma 5.3.11 (Strengthening). *If* $\Gamma, x{\prec}A \vdash B{:}Y$ *and* $x \notin \mathsf{var}(B)$*, then* $\Gamma \vdash B{:}Y$ *and* $x \notin Y^i$.

Proof. The proof is analogous to the proof of Lemma 2.3.13. □

Proposition 5.3.12 (Uniqueness of types). *If* $\Gamma \vdash_{\mathcal{R}} A : X$ *and* $\Gamma \vdash_{\mathcal{R}} A : Y$*, then* $X^i = Y^i$*,* $X^o = Y^o$*,* $X^j = Y^j$ *and* $X^p = Y^p$.

Proof. The proof is analogous to the proof of Proposition 2.3.14. □

5.3.3 Soundness Proofs

Proof of Lemma 5.3.4 (Preservation). *If* $\Gamma \models_{\mathcal{R}} \mathbf{S}_1$ *and* $\mathbf{S}_1 \longrightarrow \mathbf{S}_2$*, then* $\Gamma \models_{\mathcal{R}} \mathbf{S}_2$.

Proof. By case analysis on the transition relation $\longrightarrow$. Since in most cases only the top of the stack is affected, we restrict our attention to the parts of the stack that change.

- Case osNew:

$$\text{(osNew)}\quad x{\prec}A \in \mathit{Decls}$$
$$\mathbf{S}_1 = \mathbf{S} \circ (M, \mathtt{new}\, xE) \longrightarrow \mathbf{S} \circ (M + x, AE) = \mathbf{S}_2$$

Since $\mathbf{S}_1$ is well-typed, there exists X such that $\Gamma \vdash \texttt{new}\, xE : X$ and $[\mathbf{S}]+M+X^j \subseteq \mathcal{R}$. We only need to prove that $\Gamma \vdash AE : Z$ and $[\mathbf{S}] + (M + x) + Z^j \subseteq \mathcal{R}$ since the stack only changes at the top.

We prove AE well-typed as follows. By Generation Lemma 5.3.9, clause 3 applied to $\Gamma \vdash \texttt{new}\, xE : X$, we get $\Gamma \vdash \texttt{new}\, x : X_1$ and $\Gamma \vdash E : X_2$ with $X = \langle X_1^i \cup (X_1^o + X_2^j) \cup X_2^i, (X_1^o + X_2^p) \cup X_2^o, X_1^j \cup (X_1^p + X_2^j), X_1^p + X_2^p \rangle$.

Also by Generation Lemma 5.3.9, clause 1 applied to $\Gamma \vdash \texttt{new}\, x : X_1$ and Lemma 5.3.10, we get $\Gamma \vdash A : Y$ with $X_1 = \langle Y^i + x, Y^o + x, Y^j + x, Y^p + x \rangle$. So we can derive $\Gamma \vdash AE : \langle Y^i \cup (Y^o + X_2^j) \cup X_2^i, (Y^o + X_2^p) \cup X_2^o, Y^j \cup (Y^p + X_2^j), Y^p + X_2^p \rangle$ by applying the rule SEQ as the side condition $Y^o + X_2^j \subseteq \mathcal{R}$ holds by $Y^o \subset Y^o + x = X_1^o$ and $X_1^o + X_2^j \subseteq \mathcal{R}$.

Next we need to prove that $[\mathbf{S}] + (M + x) + (Y^j \cup (Y^p + X_2^j)) \subseteq \mathcal{R}$. We have:

$$\begin{aligned} LHS &= [\mathbf{S}] + (M + x) + (Y^j \cup (Y^p + X_2^j)) \\ &= [\mathbf{S}] + M + ((Y^j + x) \cup ((Y^p + x) + X_2^j)) \\ &= [\mathbf{S}] + M + (X_1^j \cup (X_1^p + X_2^j)) && (X_1^* = Y^* + x) \\ &= [\mathbf{S}] + M + X^j \\ &\subseteq \mathcal{R} && (\mathbf{S}_1 \text{ well-typed}) \end{aligned}$$

- Case osReu1:

$$\begin{array}{l} (\mathsf{osReu1}) \quad x \prec A \in \mathit{Decls} \quad x \notin [\mathbf{S}] + M \\ \mathbf{S}_1 = \mathbf{S} \circ (M, \texttt{reu}\, xE) \longrightarrow \mathbf{S} \circ (M + x, AE) = \mathbf{S}_2 \end{array}$$

Since $\mathbf{S}_1$ is well-typed, there exists X such that $\Gamma \vdash \texttt{reu}\, xE : X$ and $[\mathbf{S}]+M+X^j \subseteq \mathcal{R}$. As in the case osNew we only need to prove that $\Gamma \vdash AE : Z$ and $[\mathbf{S}]+(M+x)+Z^j \subseteq \mathcal{R}$.

First we prove that AE is well-typed. By Generation Lemma 5.3.9, clause 3 applied to $\Gamma \vdash \texttt{reu}\, xE : X$, we get $\Gamma \vdash \texttt{reu}\, x : X_1$ and $\Gamma \vdash E : X_2$ with $X = \langle X_1^i \cup (X_1^o + X_2^j) \cup X_2^i, (X_1^o + X_2^p) \cup X_2^o, X_1^j \cup (X_1^p + X_2^j), X_1^p + X_2^p \rangle$.

Also by Generation Lemma 5.3.9, clause 2 applied to $\Gamma \vdash \texttt{reu}\, x : X_1$ and Lemma 5.3.10, we get $\Gamma \vdash A : Y$ with $X_1 = \langle Y^i + x, Y^o + x, Y^j, Y^p \rangle$. So we can derive $\Gamma \vdash AE : \langle Y^i \cup (Y^o + X_2^j) \cup X_2^i, (Y^o + X_2^p) \cup X_2^o, Y^j \cup (Y^p + X_2^j), Y^p + X_2^p \rangle$ as the side condition $Y^o + X_2^j \subseteq \mathcal{R}$ holds by $Y^o \subset X_1^o$ and $X_1^o + X_2^j \subseteq \mathcal{R}$.

Next we prove that $[\mathbf{S}] + (M + x) + Z^j \subseteq \mathcal{R}$. Note that $Z^j = Y^j \cup (Y^p + X_2^j) = X_1^j \cup (X_1^p + X_2^j) = X^j$, so for all $z \neq x$ we have $([\mathbf{S}] + (M + x) + Z^j)(z) \leq \mathcal{R}(z)$ holds by assumption. For $z = x$, as $x \notin [\mathbf{S}] + M$ in the side condition, we have:

$$\begin{aligned} &([\mathbf{S}] + (M + x) + Z^j)(x) \\ &= (x + Z^j)(x) && (x \notin [\mathbf{S}] + M) \\ &= (x + (Y^j \cup (Y^p + X_2^j)))(x) && (Z^j = Y^j \cup (Y^p + X_2^j)) \\ &\leq ((x + Y^i) \cup ((x + Y^o) + X_2^j))(x) && (Y^j \subseteq Y^i,\ Y^p \subseteq Y^o, \text{Lemma 5.3.7, clause 5}) \\ &= (X_1^i \cup (X_1^o + X_2^j))(x) && (X_1^i = Y^i + x, X_1^o = Y^o + x) \\ &\leq (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i)(x) \\ &= X^i(x) && (X^i = X_1^i \cup (X_1^o + X_2^j) \cup X_2^i) \\ &\leq \mathcal{R}(x) = \mathcal{R}(z) && (\text{Lemma 5.3.7, clause 4}) \end{aligned}$$

- Case osReu2:

$$\begin{array}{l}(\mathsf{osReu2}) \quad x \prec A \in \mathit{Decls} \quad x \in [\mathbf{S}] + M \\ \mathbf{S}_1 = \mathbf{S} \circ (M, \mathtt{reu}\, xE) \longrightarrow \mathbf{S} \circ (M, AE) = \mathbf{S}_2\end{array}$$

Since $\mathbf{S}_1$ is well-typed, there exists X such that $\Gamma \vdash \mathtt{reu}\, xE : X$ and $[\mathbf{S}]+M+X^j \subseteq \mathcal{R}$. As in the case osNew we only need to prove that $\Gamma \vdash AE : Z$ and $M + [\mathbf{S}] + Z^j \subseteq \mathcal{R}$.

The proof of $\Gamma \vdash AE : Z$ is the same as in the previous case, with again $Z^j = X^j$, which trivializes the second proof obligation.

- Case osChoice:

$$\begin{array}{l}(\mathsf{osChoice}) \quad i \in \{1, 2\} \\ \mathbf{S}_1 = \mathbf{S} \circ (M, (A_1 + A_2)E) \longrightarrow \mathbf{S} \circ (M, A_i E) = \mathbf{S}_2\end{array}$$

We treat the case $i = 1$. The case $i = 2$ is symmetric.

Since $\mathbf{S}_1$ is well-typed, there exists X such that $\Gamma \vdash (A_1+A_2)E : X$ and $[\mathbf{S}]+M+X^j \subseteq \mathcal{R}$. We need to prove that $\Gamma \vdash A_1 E : Z$ and $[\mathbf{S}] + M + Z^j \subseteq \mathcal{R}$.

First we prove that $A_1 E$ is well-typed. By Generation Lemma 5.3.9, clause 3 applied to $\Gamma \vdash (A_1 + A_2)E : X$, we get $\Gamma \vdash (A_1 + A_2) : X_1$ and $\Gamma \vdash E : X_2$. Also by Generation Lemma 5.3.9, clause 4 applied to $\Gamma \vdash (A_1 + A_2) : X_1$, we have $\Gamma \vdash A_1 : Y$ with $Y^* \subseteq X_1^*$. So we can derive $\Gamma \vdash A_1 E : \langle Y^i \cup (Y^o + X_2^j) \cup X_2^i, (Y^o + X_2^p) \cup X_2^o, Y^j \cup (Y^p + X_2^j), Y^p + X_2^p \rangle$ as the side condition obviously holds.

Next we prove the second clause $[\mathbf{S}] + M + Z^j \subseteq \mathcal{R}$. We have:

$$\begin{aligned} LHS &= [\mathbf{S}] + M + (Y^j \cup (Y^p + X_2^j)) && (Z = Y^j \cup (Y^p + X_2^j)) \\ &\subseteq [\mathbf{S}] + M + (X_1^j \cup (X_1^p + X_2^j)) && (Y^* \subseteq X_1^*) \\ &= [\mathbf{S}] + M + X^j \\ &\subseteq \mathcal{R} && (\mathbf{S}_1 \text{ well-typed}) \end{aligned}$$

- Case osPush:

$$\begin{array}{l}(\mathsf{osPush}) \\ \mathbf{S}_1 = \mathbf{S} \circ (M, \{A\}E) \longrightarrow \mathbf{S} \circ (M, E) \circ ([], A) = \mathbf{S}_2\end{array}$$

Since $\mathbf{S}_1$ is well-typed, there exists X such that $\Gamma \vdash \{A\}E : X$ and $[\mathbf{S}] + M + X^j \subseteq \mathcal{R}$. Because we have pushed the stack, we have two things to check. For the well-typedness of the new configuration, we need to prove that (i) $\Gamma \vdash A : Y$ and $[\mathbf{S}] + M + [] + Y^j \subseteq \mathcal{R}$, and (ii) $\Gamma \vdash E : X_2$ and $[\mathbf{S}] + M + X_2^j \subseteq \mathcal{R}$.

We prove (i) as follows. By Generation Lemma 5.3.9, clause 3 applied to $\Gamma \vdash \{A\}E : X$, we get $\Gamma \vdash \{A\} : X_1$ and $\Gamma \vdash E : X_2$ with $X^j = X_1^j \cup (X_1^p + X_2^j)$. Also by Generation Lemma 5.3.9, clause 5 applied to $\Gamma \vdash \{A\} : X_1$, we get the first subclause $\Gamma \vdash A : Y$ with $X_1 = \langle Y^i, [], Y^j, [] \rangle$. The second subclause follows by $Y^j = X_1^j \subseteq X^j$ and $[\mathbf{S}] + M + X^j \subseteq \mathcal{R}$ by the well-typedness of $\mathbf{S}_1$.

For (ii) the first subclause is proved in (i). The second subclause follows by $X_2^j \subseteq X_1^p + X_2^j \subseteq X^j$ and $[\mathbf{S}] + M + X^j \subseteq \mathcal{R}$.

- Case osPop:

$$\begin{array}{l}(\mathsf{osPop}) \\ \mathbf{S} \circ (M, E) \circ (M', \epsilon) \longrightarrow \mathbf{S} \circ (M, E)\end{array}$$

We get immediately $\mathbf{S} \circ (M, E)$ well-typed by $\mathbf{S} \circ (M, E) \circ (M', \epsilon)$ well-typed.

$\square$

Proof of Lemma 5.3.5 (Progress). *If* $\Gamma \models_{\mathcal{R}} \mathbf{S}$*, then either* $\mathbf{S}$ *is terminal or there exists configuration* $\mathbf{S}'$ *such that* $\mathbf{S} \longrightarrow \mathbf{S}'$*.*

Proof. The proof is analogous to the proof of Lemma 2.3.5. □

Proof of Theorem 5.3.6 (Soundness). *If program Prog* $=$ *Decls*$;E$ *is well-typed with respect to a requirement* $\mathcal{R}$*, then for any* $\mathbf{S}$ *such that* $([], E) \longrightarrow^* \mathbf{S}$ *we have* $\mathbf{S}$ *is not stuck and* $[\mathbf{S}] \subseteq \mathcal{R}$*.*

Proof. Since program *Prog* is well-typed, by Definition 5.2.3 there exist a reordering Γ of declarations in *Decls* and a type X such that $\Gamma \vdash_{\mathcal{R}} E : X$. Hence $([], E)$ is a well-typed configuration.

Both conclusions follow by Lemma 5.3.4, Lemma 5.3.5 and the transitivity of $\subseteq$. □

Termination. As in Chapter 2, any well-typed programs terminates after a finite number of transition steps. The termination theorem and its proof are analogous to Theorem 2.3.15. We only need to extend the function mts for the new form of expressions: $\mathtt{reu}\, x$. The definition of mts for $\mathtt{reu}\, x$ is the same as for $\mathtt{new}\, x$.

$$\mathsf{mts}(\mathtt{reu}\, x) = 1 + \mathsf{mts}(A), \qquad \text{if } x \prec A \in \mathit{Decls}$$

Chapter 6

Reuse Instantiation and Parallel Composition

In this chapter, we extend the language of Chapter 5 with parallel composition. Similar to adding the parallel composition in Chapter 4, the syntax of the language simply has a new form of expressions and we use binary trees for describing the operational semantics. The sophisticated parts of this chapter are caused by the notion of reusable instances and an additional set in the definition of types. The type inference is analogous as in Chapter 2, so we leave it out for brevity.

6.1 Language

6.1.1 Syntax

Table 6.1 defines component programs, declarations and expressions of the language. Compared to the language of Chapter 5, the syntax has only a new form of expressions: parallel composition, which allows many configurations in Chapter 5 running concurrently.

Table 6.1: Syntax of the language with reu and parallel composition

$Prog$	$::=$	$Decls; E$	Program
$Decls$	$::=$	$\overline{x {\prec} E}$	Declarations
$A, .., E$	$::=$		Expressions
		ϵ	Empty expression
	$\mid$	$\mathtt{new}\, x$	New instantiation
	$\mid$	$\mathtt{reu}\, x$	Reuse instantiation
	$\mid$	$E\ E$	Sequencing
	$\mid$	$(E + E)$	Choice
	$\mid$	$(E \parallel E)$	Parallel
	$\mid$	$\{E\}$	Scope

The following example program will be used to illustrate the operational semantics and typing derivations in the subsequent sections.

$$\begin{array}{l} d{\prec}\epsilon \quad e{\prec}\epsilon \quad a{\prec}(\mathtt{new}\, d \parallel \{\mathtt{reu}\, d\}\, \mathtt{reu}\, e) \\ b{\prec}(\mathtt{reu}\, d\{\mathtt{new}\, a\} + \mathtt{new}\, e\, \mathtt{new}\, a)\, \mathtt{reu}\, d; \\ \mathtt{reu}\, b \end{array}$$

In this example, d and e are primitive components. Component a is the parallel composition of `new` d and $\{$ `reu` $d\}$ `reu` e. Component b has a choice expression before a reuse of an instance of d. The main expression of the program is `reu` b.

6.1.2 Operational Semantics

As in Chapter 4, we use binary trees of stacks to model the machine states. However, because the semantics of the reuse primitive depends on the whole machine state, we modeled the operational semantics in this chapter by two relations: a reduction relation and a structural congruence relation. The reduction relation is a set of small-step reduction rules between configurations. These rules define the behaviour of each primitive of component programs. The structural congruence relation, essentially commutativity of $+$ and $\|$, allows us to rearrange the structure of configurations so that the reduction rules may be applied.

Before going into the details of congruence and reduction rules, we define the notion of configuration and its relevant notions. As in Chapter 4, a *configuration* is a binary tree $\mathbb{T}$ of threads. A thread is a stack $\mathbf{S}$ of pairs (M, E) of a local store and an expression, where M is a multiset over component names $\mathbb{C}$, and E is an expression as defined in Table 6.1. A thread is *active* if it is a leaf thread. Reduction always occurs at one of the leaf/active threads. A configuration is *terminal* if it has only one thread of the form (M, ϵ). The notation of stacks and configurations are the same as before:

$\mathbf{S}$	$::=$	$(M_1, E_1) \circ \ldots \circ (M_n, E_n)$	Stack
$\mathbb{T}, \mathbb{R}$	$::=$		Configurations
		$\mathsf{Lf}(\mathbf{S})$	Leaf
	$\mid$	$\mathsf{Nd}(\mathbf{S}, \mathbb{T})$	Node with one branch
	$\mid$	$\mathsf{Nd}(\mathbf{S}, \mathbb{T}, \mathbb{T})$	Node with two branches

The above stack $\mathbf{S}$ has n elements where (M_1, E_1) is the bottom, (M_n, E_n) is the top of the stack, and '$\circ$' is the stack separator. We denote by $\mathsf{hi}(\mathbf{S})$ the height of the stack and $\mathbf{S}|_k$ is the stack of from bottom to the kth element: $\mathbf{S}|_k = (M_1, E_1) \circ \ldots \circ (M_k, E_k)$. By $[\mathbf{S}]$ we denote the multiset of active instances in $\mathbf{S}$, that is, $[\mathbf{S}] = M_1 + \ldots + M_n$.

We assign to each node in a tree a *location*, ranged over by α and β, as in Section 4.1.2. We denote by $\mathsf{leaves}(\mathbb{T})$ the set of locations of all the leaves of $\mathbb{T}$. We denote by $\mathbb{T}(\alpha)$ the stack at location α in $\mathbb{T}$. As in Section 4.3, we call $\alpha.k$ the *position* of the kth element (from the bottom) of the stack $\mathbb{T}(\alpha)$. Recall, the set of all positions $\alpha.k$ of a tree is a partially ordered set.

The next notion of *reusable instances* is important, because semantics of the primitive `reu` depends on the state of the configuration. For the moment we only need the concept of reusable instances for an expression at the top of a leaf node. Later, when formalizing soundness property, we will need to extend the notion of reusable instances to other positions of a configuration. However, the definition of reusable instances for the top of a leaf is the same as for other positions of a leaf, so we will define the notion for all positions of a leaf.

The multiset of reusable instances at level k of the leaf stack α is the collection of all existing instances in all the predecessor nodes $\beta \prec \alpha$ and all the existing instances from the bottom of stack $\mathbb{T}(\alpha)$ up to k (inclusive).

$$\mathsf{reuLf}_{\mathbb{T}}(\alpha.k) = \biguplus_{\beta \prec \alpha} [\mathbb{T}(\beta)] \quad + \quad [\mathbb{T}(\alpha)|_k]$$

The reduction relation is defined in terms of a rewriting system [40]. By $\mathbb{T}[\![\,]\!]_\alpha$ we denote a tree with a hole at the leaf location α. Filling this hole with a (sub)tree $\mathbb{T}'$ will be denoted by $\mathbb{T}[\![\mathbb{T}']\!]_\alpha$. $\mathbb{T}'$ now becomes a branch of $\mathbb{T}$.

Table 6.2 defines the reduction rules. Each reduction rule has two lines. The first

Table 6.2: Reduction rules of the language with `reu` and parallel composition

(osNew) $x \prec A \in \mathit{Decls}$
$\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{new}\, xE))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\alpha$

(osReu1) $x \prec A \in \mathit{Decls}$ $x \notin \mathsf{reuLf}_{\mathbb{T}}(\alpha.\mathsf{hi}(\mathbb{T}(\alpha)))$
$\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{reu}\, xE))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\alpha$

(osReu2) $x \prec A \in \mathit{Decls}$ $x \in \mathsf{reuLf}_{\mathbb{T}}(\alpha.\mathsf{hi}(\mathbb{T}(\alpha)))$
$\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{reu}\, xE))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\alpha$

(osChoice)
$\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, (A + B)E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\alpha$

(osPush)
$\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \{A\}E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ ([], A))]\!]_\alpha$

(osPop)
$\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ (M', \epsilon))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E))]\!]_\alpha$

(osParIntr)
$\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, (A \parallel B)E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B)))]\!]_\alpha$

(osParElim1)
$\mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathbb{R}, \mathsf{Lf}((M', \epsilon)))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})]\!]_\alpha$

(osParElim2)
$\mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + M', E))]\!]_\alpha$

(osCong) $\mathbb{R} \equiv \mathbb{R}'$
$\mathbb{T}[\![\mathbb{R}]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathbb{R}']\!]_\alpha$

Table 6.3: Structural congruence: basic axioms

(conChoice)
$\mathsf{Lf}(\mathbf{S} \circ (M, (A + B)E)) \equiv \mathsf{Lf}(\mathbf{S} \circ (M, (B + A))E)$
(conBranch)
$\mathsf{Nd}(\mathbf{S}, \mathsf{Lf}(\mathbf{S}), \mathbb{T}) \equiv \mathsf{Nd}(\mathbf{S}, \mathbb{T}, \mathsf{Lf}(\mathbf{S}))$

line contains a rule name followed by a list of conditions. The second line has the form $\mathbb{T} \longrightarrow \mathbb{T}'$, which states that if the configuration has the form $\mathbb{T}$ and all the conditions in the first line hold, then $\mathbb{T}$ can move to $\mathbb{T}'$. As usual, $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

One-step reduction is defined first by choosing an arbitrary active thread. Then depending on the pattern of the expression at the top of the chosen thread and the state of the configuration, the appropriate rewrite rule is selected. If necessary the configuration is rearranged by the congruence rules. By the rules osNew, osReu1, osReu2, and osChoice we only rewrite the element at the top of the stack. The rule osPush adds an element to the top of the leaf stack. The rule osPop only removes the element at the top of the stack when the stack has at least two elements. This means that no stack in any configuration is empty. By the rule osParIntr, a leaf is replaced by a branch of a node and two leaves. In contrast, by the rules osParElim1, osParElim2, a leaf is removed from the tree and its parent node may be promoted to be a leaf if it is the case (osParElim2). The rule osCong allows the configuration to be rearranged so that reduction rule can be applied.

The structural congruence relation $\equiv$ is defined in Table 6.3. By the congruence rules, we can replace the left hand side of $\equiv$ by the right hand side in the reduction rule osCong.

The example at the end of Section 6.1.1 is used to illustrate the operational semantics. There are many possible runs of the program due to the choice composition and when a configuration has more than one leaf thread, the number of possible runs can be exponential as active threads have the same priority. Here we only show one of the possible runs. To make it easier to follow, we represent the trees graphically instead of using the formal syntax; '$\leftharpoonup$' and '$\langle$' denote branches with one and two child nodes, respectively. At the starting point, the configuration has one leaf $\mathsf{Lf}([], \texttt{reu}\, b)$. After the first step, there are two possibilities because we can apply the congruence rule $\mathsf{conChoice}$ before the rule $\mathsf{osChoice}$.

$$
\begin{array}{rll}
(\mathsf{Start}) & ([], \texttt{reu}\, b) & \\
(\mathsf{osReu}) \longrightarrow & ([b], (\,\texttt{reu}\, d\{\,\texttt{new}\, a\} + \texttt{new}\, e\, \texttt{new}\, a)\, \texttt{reu}\, d) & \\
(\mathsf{osChoice}) \longrightarrow & ([b], \texttt{reu}\, d\{\,\texttt{new}\, a\}\, \texttt{reu}\, d) & (\text{or } ([b], \texttt{new}\, e\, \texttt{new}\, a\, \texttt{reu}\, d))
\end{array}
$$

Now we continue with the first possibility. When there are two or more leaves, we draw a box around the leaf which is to be executed in the next step.

$$
\begin{array}{rl}
 & ([b], \texttt{reu}\, d\{\,\texttt{new}\, a\}\, \texttt{reu}\, d) \\
(\mathsf{osReu1}) \longrightarrow & ([b, d], \{\,\texttt{new}\, a\}\, \texttt{reu}\, d) \\
(\mathsf{osPush}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([], \texttt{new}\, a) \\
(\mathsf{osNew}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a], (\,\texttt{new}\, d \parallel \{\,\texttt{reu}\, d\}\, \texttt{reu}\, e)) \\
(\mathsf{osParIntr}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a], \epsilon) \ \langle \begin{array}{l} ([], \texttt{new}\, d) \\ \boxed{([], \{\,\texttt{reu}\, d\}\, \texttt{reu}\, e)} \end{array} \\
(\mathsf{osPush}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a], \epsilon) \ \langle \begin{array}{l} \boxed{([], \texttt{new}\, d)} \\ ([], \texttt{reu}\, e) \circ ([], \texttt{reu}\, d) \end{array} \\
(\mathsf{osNew}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a], \epsilon) \ \langle \begin{array}{l} ([d], \epsilon) \\ \boxed{([], \texttt{reu}\, e) \circ ([], \texttt{reu}\, d)} \end{array} \\
(\mathsf{osReu1}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a], \epsilon) \ \langle \begin{array}{l} \boxed{([d], \epsilon)} \\ ([], \texttt{reu}\, e) \circ ([], \epsilon) \end{array} \\
(\mathsf{osParElim1}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a, d], \epsilon) \leftharpoonup ([], \texttt{reu}\, e) \circ ([], \epsilon) \\
(\mathsf{osPop}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a, d], \epsilon) \leftharpoonup ([], \texttt{reu}\, e) \\
(\mathsf{osReu}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a, d], \epsilon) \leftharpoonup ([e], \epsilon) \\
(\mathsf{osParElim2}) \longrightarrow & ([b, d], \texttt{reu}\, d) \circ ([a, d, e], \epsilon) \\
(\mathsf{osPop}) \longrightarrow & ([b, d], \texttt{reu}\, d) \\
(\mathsf{osReu2}) \longrightarrow & ([b, d], \epsilon) \qquad\qquad (\text{terminal})
\end{array}
$$

Last, we should note that we could model the operational semantics slightly simpler by using only *complete binary trees*. A complete binary tree is a binary tree with the additional property that every node must have exactly two children if an internal node, and zero children if a leaf node. Then we have only one rule for truncating the tree:

$$
\begin{array}{l}
(\mathsf{osParElim}) \\
\mathbb{T}[\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)), \mathsf{Lf}((M'', \epsilon)))]_\alpha \longrightarrow \mathbb{T}[\mathsf{Lf}(\mathbf{S} \circ (M + M' + M'', E))]_\alpha
\end{array}
$$

However, doing in this way reduces the reuse capability because two sibling threads cannot reuse instances of each other, after one has terminated before the other. In this model this is possible as a leaf can return its instances to its parent and the other sibling branch can reuse the instances from its parent.

To maximize reuse, we can even allow sibling threads to reuse each other instances in

the store at bottom of their stacks, since we know that these instances will be returned to their parent thread. However this is naturally unsafe, especially in the presence of the explicit deallocation of Chapters 3, 4.

6.2 Type System

We start this section by describing the types informally. Then we will define and explain the typing rules in more details.

Definition 6.2.1 (Types). *Types of component expressions are tuples*

$$X = \langle X^i, X^o, X^j, X^p, X^l \rangle$$

where X^i, X^o, X^j, X^p and X^l are finite multisets over $\mathbb{C}$.

Let us first explain informally why multisets, which multisets and why five. In short, the first four multisets have the same purpose as in Chapter 5. The last multiset X^l is just a set of instances that survive the expression in any possible execution path. The reasons we need X^l are as follows.

Recall that without the parallel composition, the difference between $X^i(x)$ and $X^j(x)$ as well as between $X^o(x)$ and $X^p(x)$ is at most one for every x (see Lemma 5.3.7). With the addition of the parallel composition, these differences may be greater than one. For example, $(\mathtt{reu}\,x \parallel \mathtt{reu}\,x)$ generates no x's if there is a reusable x for the expression, and maximum two x's otherwise. Moreover, consider expression AB, due to the non-determinism of the choice composition, the surviving instances after executing A are also non-deterministic. For example, $\mathtt{new}\,x + \mathtt{new}\,y$ may or may not leave an active x. In order to obtain a sharp bound for x during the execution of AB, we need to know whether B can always reuse x after executing A or not. Because if it is the case, the maximum number of additional instances of x generated by B is only $Y^j(x)$, where Y is the type of B. Therefore, we need the last multiset X^l in the type expression. X^l is the set of instances which are guaranteed to survive at the end of the execution of A, in any run. Although X^l could be a set, we let X^l be a multiset so that the multiset operations in the later sections can be applied without any conversion. We will have more explanation in the typing relation below, but before that we need some preliminary notions. Many of these notions are the same as in previous chapters.

As in Section 5.2, a requirement $\mathcal{R}$ is a total function from $\mathbb{C}$ to $\mathbb{N} \cup \{\infty\}$. $\mathcal{R}(x) \in \mathbb{N}$ is the maximum allowed number of simultaneously active instances of x; $\mathcal{R}(x) = \infty$ expresses that x can have any number of instances. By definition $n < \infty$ for all $n \in \mathbb{N}$. For a multiset M, we denote $M \subseteq \mathcal{R}$ when $M(x) \leq \mathcal{R}(x)$ for all $x \in M$.

The notions of bases and typing judgments are also the same as in Chapter 5. A typing judgment is a tuple of the form:

$$\Gamma \vdash_{\mathcal{R}} A : X$$

and it asserts that expression A has type X in the environment Γ, with respect to requirement $\mathcal{R}$. We leave out the subscript $\mathcal{R}$ when it is clear from context.

Definition 6.2.2 (Valid typing judgments). *Let $\mathcal{R}$ be a requirement. Valid typing judgments $\Gamma \vdash_{\mathcal{R}} A : X$ are derived by applying the typing rules in Table 6.4 in the usual inductive way.*

In the rule Seq in Table 6.4, the operation $M!_N$, where M, N are multisets, is defined as follows:

$$(M!_N)(x) = \begin{cases} 0, & \text{if } x \in N \\ M(x), & \text{otherwise} \end{cases}$$

We let this operator have higher order of priority than other multiset operations.

Table 6.4: Typing rules of the language with `reu` and parallel composition

(Axiom)
$$\frac{}{\varnothing \vdash \epsilon : \langle [], [], [], [], [] \rangle}$$

(WeakenB)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec B \vdash A : X}$$

(New)
$$\frac{\Gamma \vdash A : X \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{new}\, x : \langle X^i + x, X^o + x, X^j + x, X^p + x, X^l + x \rangle}$$

(Reu)
$$\frac{\Gamma \vdash A : X \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{reu}\, x : \langle X^i + x, X^o + x, X^j, X^p, X^l + x \rangle}$$

(Seq)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad X^o + Y^j \subseteq \mathcal{R} \quad A, B \neq \epsilon}{\Gamma \vdash AB : \langle X^i \cup (X^o + Y^j) \cup Y^i!_{X^l}, (X^o + Y^p) \cup Y^o!_{X^l}, X^j \cup (X^p + Y^j), X^p + Y^p, X^l \cup Y^l \rangle}$$

(Choice)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y}{\Gamma \vdash (A + B) : \langle X^i \cup Y^i, X^o \cup Y^o, X^j \cup Y^j, X^p \cup Y^p, X^l \cap Y^l \rangle}$$

(Parallel)
$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad X^i + Y^i \subseteq \mathcal{R}}{\Gamma \vdash (A \parallel B) : \langle X^i + Y^i, X^o + Y^o, X^j + Y^j, X^p + Y^p, X^l \cup Y^l \rangle}$$

(Scope)
$$\frac{\Gamma \vdash A : X}{\Gamma \vdash \{A\} : \langle X^i, [], X^j, [], [] \rangle}$$

In addition to the intuition given in the beginning of this section, some further explanation of these typing rules is in order. The most critical rule is Seq because sequencing two expressions can lead to increase in instances of the composed expression. Let us start with the first multiset of the type expression of AB. After the expression A is executed, there are at most $X^o(x)$ instances of component x. If x is not in the system state after the execution of A, then at most $Y^i(x)$ instances of x can be created when executing B. Otherwise, at most $Y^j(x)$ additional instances of x can be created. If we take the maximum of $(X^o+Y^j)(x)$ and $Y^i(x)$ to be the maximum number of x that can be created after the execution of A and during the execution of B, then we do not obtain the sharp upper bound. For example, let $A = \mathtt{reu}\, x$ and $B = (\mathtt{reu}\, x \parallel \mathtt{reu}\, x)$. Executing B alone can create two instances of x. However, executing AB creates only one instance of x.

To remedy the situation we need to know whether an instance of x is always in the system state after the execution of A or not. If it is, then we know that at most $Y^j(x)$ additional instances can be created; otherwise, $Y^i(x)$ additional instances can be created when executing B. Therefore the maximum number of x after execution of A and during execution of B is either $(X^o+Y^j)(x)$, or $(X^l+Y^j)(x)$ if $X^l(x) \geq 1$, or $Y^i(x)$ if $X^l(x) = 0$. Since $X^o \supseteq X^l$, the number becomes $((X^o + Y^j) \cup Y^i!_{X^l})(x)$.

Moreover, because executing A can create at most $X^i(x)$ instances, the first component of type of AB is the maximum of $X^i(x)$ and $((X^o + Y^j) \cup Y^i!_{X^l})(x)$. For the safety the maximum must not excess $\mathcal{R}$, however, since X^i and Y^i already satisfy the requirement $\mathcal{R}$, we only require $X^o + Y^j \subseteq \mathcal{R}$ in the side condition.

Analogously, after executing AB, the maximum number of surviving instances of x is

either $X^o(x)+Y^p(x)$, or $Y^o(x)$ if there is a run of A which ends with no surviving instance of x. Hence the surviving instances of AB are $(X^o + Y^p) \cup Y^o!_{X^l}$.

By a similar reasoning, when we start with a stack containing at least one instance of every component, we can calculate the second and the last components in the type expression for AB and the whole type expression of AB is $\langle X^i \cup (X^o{+}Y^j) \cup Y^i!_{X^l}, (X^o + Y^p) \cup Y^o!_{X^l}, X^j \cup (X^p{+}Y^j), X^p + Y^p, X^l \cup Y^l \rangle$.

Other typing rules are easy referring to the semantics of each multiset of a type. The rule Axiom requires no premise and is used for startup. The rules New and Reu allow us to type expressions $\texttt{new}\, x$ and $\texttt{reu}\, x$, respectively. The rule WeakenB is used to expand bases so that we can combine typings in the other rules. The side condition $x \notin \mathsf{dom}(\Gamma)$ in the rules WeakenB, New and Reu prevents ambiguity and circularity. The rules Choice and Scope are easy to understand recalling the semantics of the corresponding reduction rules osChoice, osPush and osPop. In the rule Parallel, since we have no specific schedule for two parallel threads, both can generate their maximum numbers of instances for any component. To be on the safe side, we have to prepare for the worst case and therefore the type of two parallel expressions is additive union of their types but the last multiset. Recall that the semantics of the last multiset is just a set, it is enough to take union for the last multiset. The side condition follows naturally.

The definition of *well-typed programs* is analogous to the one in the previous chapter.

Definition 6.2.3 (Well-typed programs). *Let $\mathcal{R}$ be a requirement. Program Prog = Decls; E is well-typed with respect to $\mathcal{R}$ if there exist a reordering Γ of declarations in Decls and a type X such that $\Gamma \vdash_{\mathcal{R}} E : X$.*

Using the example in Section 6.1.1 with assumption that $\mathcal{R} = \{b \mapsto 1, e \mapsto 2, a, d \mapsto 4\}$, we derive type for $\texttt{reu}\, b$. Note that we omitted some side conditions as they can be checked easily and we shortened the rule names to the first two characters. The rule Axiom is also simplified.

$$\textsc{We}\,\dfrac{\textsc{Sc}\,\dfrac{\textsc{Re}\,\dfrac{\varnothing \vdash \epsilon : \langle [], [], [], [], [] \rangle}{d \mathrel{-\!\!\prec} \epsilon \vdash \texttt{reu}\, d : \langle [d], [d], [], [], [d] \rangle}}{d \mathrel{-\!\!\prec} \epsilon \vdash \{\, \texttt{reu}\, d\} : \langle [d], [], [], [], [] \rangle} \quad \textsc{We}\,\dfrac{\varnothing \vdash \epsilon : \langle [], [], [], [], [] \rangle}{d \mathrel{-\!\!\prec} \epsilon \vdash \epsilon : \langle [], [], [], [], [] \rangle}}{d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon \vdash \{\, \texttt{reu}\, d\} : \langle [d], [], [], [], [] \rangle} \tag{6.1}$$

$$\textsc{Se}\,\dfrac{(6.1) \quad \textsc{Re}\,\dfrac{\textsc{We}\,\dfrac{\varnothing \vdash \epsilon : \langle [], [], [], [], [] \rangle \quad \varnothing \vdash \epsilon : \langle [], [], [], [], [] \rangle}{d \mathrel{-\!\!\prec} \epsilon \vdash \epsilon : \langle [], [], [], [], [] \rangle}}{d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon \vdash \texttt{reu}\, e : \langle [e], [e], [], [], [e] \rangle}}{d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon \vdash \{\, \texttt{reu}\, d\}\, \texttt{reu}\, e : \langle [d, e], [e], [], [], [e] \rangle} \tag{6.2}$$

$$\textsc{Ne}\,\dfrac{\textsc{Pa}\,\dfrac{\textsc{We}\,\dfrac{\textsc{Ne}\,\dfrac{\varnothing \vdash \epsilon : \langle [], [], [], [], [] \rangle}{d \mathrel{-\!\!\prec} \epsilon \vdash \texttt{new}\, d : \langle [d], [d], [d], [d], [d] \rangle}}{d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon \vdash \texttt{new}\, d : \langle [d], [d], [d], [d], [d] \rangle} \quad (6.2)}{d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon \vdash (\, \texttt{new}\, d \parallel \{\, \texttt{reu}\, d\}\, \texttt{reu}\, e) : \langle [d, d, e], [d, e], [d], [d], [d, e] \rangle}}{d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} (\, \texttt{new}\, d \parallel \{\, \texttt{reu}\, d\}\, \texttt{reu}\, e) \vdash \texttt{new}\, a : \langle [a, d, d, e], [a, d, e], [a, d], [a, d], [a, d, e] \rangle}$$

Similarly, we can derive $\Gamma \vdash \texttt{reu}\, b : \langle [b, a, d, d, e], [b, a, d, e], [a, d, e], [a, d, e], [a, b, d, e] \rangle$ where $\Gamma = d \mathrel{-\!\!\prec} \epsilon, e \mathrel{-\!\!\prec} \epsilon, a \mathrel{-\!\!\prec} (\, \texttt{new}\, d \parallel \{\, \texttt{reu}\, d\}\, \texttt{reu}\, d), b \mathrel{-\!\!\prec} (\, \texttt{reu}\, d\{\, \texttt{new}\, a\} + \texttt{new}\, e\, \texttt{new}\, a)\, \texttt{reu}\, d$.

In this example, $\texttt{reu}\, b$ is typable. If $\mathcal{R}(d) = 1$, the expression would not be typable as the side condition when paralleling $\texttt{new}\, d$ and $\{\, \texttt{reu}\, d\}\, \texttt{reu}\, e$ would not be satisfied. Also, note that the above typing derivation is not the only one but, as we will see later, the type of any expression is unique.

6.3 Properties

6.3.1 Type Soundness

As in the previous chapters, the proof of the type soundness is based on the approach of Wright and Felleisen [47]. We will define the notion of *well-typed configuration* and

prove two main lemmas: Preservation and Progress. We begin with some preliminary definitions.

First, the notion of *subtree* is the same as Section 4.3. Given a tree $\mathbb{T}$ and a valid set $\mathcal{L} = \{\alpha_i.k_i \in \mathbb{T} \mid i = 1..m\}$ of positions such that any two positions are not ordered: $\alpha_i.k_i \not\preccurlyeq \alpha_j.k_j$ for all $i \neq j$. Tree $\mathbb{T}'$ obtained from $\mathbb{T}$ by removing all elements at positions $\alpha.k \succcurlyeq \alpha_i.k_i$ for all $1 \leq i \leq m$ is a subtree of $\mathbb{T}$, notation $\mathbb{T}' \sqsubseteq_{\mathcal{L}} \mathbb{T}$ or $\mathbb{T}' = \mathbb{T}|_{\mathcal{L}}$.

Next, we compute the collection of instances that an expression E at an arbitrary position $\alpha.k$ of a tree $\mathbb{T}$ can reuse when the expression is executed. Recall that we have defined the collection of reusable instances for an expression in a leaf node in Section 6.1.2. Now we extend the notion to define function $\mathsf{reuI}_{\mathbb{T}}(\alpha.k)$, which returns the multiset of elements that can always be reused when the expression at $\alpha.k$ is executed. Due to the semantics of `reu`, we do not need the multiplicity of the reusable instances. However, we will define the function $\mathsf{reuI}_{\mathbb{T}}(\alpha.k)$ which returns a multiset to avoid casting from multisets to sets and vice versa in further computation.

The elements of $\mathsf{reuI}_{\mathbb{T}}(\alpha.k)$ are not only those in $\mathsf{reuLf}_{\mathbb{T}}(\alpha.k)$ but also the ones returned from its child nodes: $\mathsf{retI}_{\mathbb{T}}(\alpha.k)$—see the rules osParElim1 and osParElim2 in Table 6.2.

$$\mathsf{reuI}_{\mathbb{T}}(\alpha.k) = \mathsf{reuLf}_{\mathbb{T}}(\alpha.k) \cup \mathsf{retI}_{\mathbb{T}}(\alpha.k)$$

The instances that will be returned to $\alpha.k$, denoted by function $\mathsf{retI}_{\mathbb{T}}(\alpha.k)$, is none if $\alpha.k$ is not at the top of stack $\mathbb{T}(\alpha)$ (see the rule osPop) or α has no child nodes. Otherwise, they are the instances in the stores at the bottom of its child nodes and additional instances that the expressions at the bottom of the child nodes *will* create. Since the child nodes may have more children, we need to call the function recursively.

$$\mathsf{retI}_{\mathbb{T}}(\alpha.k) = \begin{cases} [], & \text{if } k < \mathsf{hi}(\mathbb{T}(\alpha)) \text{ or } \alpha \in \mathsf{leaves}(\mathbb{T}) \\ \bigcup_{\beta \in \{\alpha l, \alpha r\}} ([\mathbb{T}(\beta.1)] \cup X^l \cup \mathsf{retI}_{\mathbb{T}}(\beta.1)), & \text{otherwise} \end{cases}$$

where X is the type of the expression at position $\beta.1$ and $[\mathbb{T}(\beta.1)]$ is the multiset at position $\beta.1$. In fact, $\mathsf{retI}_{\mathbb{T}}(\alpha.k)$ contains instances that will be created and returned to the store at $\alpha.k$. Since we only care that there is at least one or none, so the unions in the definition suffice.

Analogously, we calculate the upper bound of instances that can be created and returned to the store at position $\alpha.k$. We denote the function by $\mathsf{retop}_{\mathbb{T}}(\alpha.k)$. The function returns an empty multiset if k is not at the top of the stack at α or α has no child nodes. Otherwise, first, it contains the existing instances in the multisets at the bottom of its child nodes and the maximal number of instances which can survive the expressions there. We denote the latter by function $\mathsf{op}_{\mathbb{T}}(\alpha.k)$:

$$\mathsf{op}_{\mathbb{T}}(\alpha.k) = X^p \cup X^o!_{\mathsf{reuI}_{\mathbb{T}}(\alpha.k)}$$

where X is the type of the expression at position $\alpha.k$. Moreover, the child nodes of $\alpha.k$ may received instances from its child nodes and so on, so we need to call the function recursively.

$$\mathsf{retop}_{\mathbb{T}}(\alpha.k) = \begin{cases} [], \quad \text{if } k < \mathsf{hi}(\mathbb{T}(\alpha)) \text{ or } \alpha \in \mathsf{leaves}(\mathbb{T}) \\ \biguplus_{\beta \in \{\alpha l, \alpha r\}} ([\mathbb{T}(\beta.1)] + \mathsf{op}_{\mathbb{T}}(\beta.1) + \mathsf{retop}_{\mathbb{T}}(\beta.1)), & \text{otherwise} \end{cases}$$

We are going to define the central notion of well-typed configuration. Its main statement is that the total number of active instances in the current configuration respects the requirement $\mathcal{R}$. Since the leaves of the configuration tree may generate more instances in the future, we need to include these instances in the total number. Furthermore, because the tree can shrink and when it shrinks, some nodes eventually become leaves we need to count on for these future states also. The function $\mathsf{ij}_{\mathbb{T}}(\alpha.k)$ below returns the maximal number of instances which can be generated by the expression at the position $\alpha.k$. As in

the sequencing typing rule SEQ, this number is bounded by the maximal number returned from its child nodes ($\mathsf{retop}_{\mathbb{T}}(\alpha.k)$) and the additional instances (X^j) for components that indeed are reused, where X is the type of the expression at position $\alpha.k$. For runs after which instance of a component x may not be in the set of reusable instances, an additional bound $X^i(x)$ should be taken into account. This explains the definition of the function ij.

$$\mathsf{ij}_{\mathbb{T}}(\alpha.k) = [\mathbb{T}(\alpha.k)] + ((\mathsf{retop}_{\mathbb{T}}(\alpha.k) + X^j) \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\alpha.k)})$$

Now we are ready to define the notion of a *well-typed configuration*. The first clause requires that all expressions in the configuration are well-typed. The second one contains the safety behaviour of the configuration. It requires that the total number of existing instances in the configuration together with the ones which may be generated by expressions in the future still respect the requirement $\mathcal{R}$.

Definition 6.3.1 (Well-typed configurations). *Configuration $\mathbb{T}$ is well-typed with respect to a basis Γ and a requirement $\mathcal{R}$, notation $\Gamma \models_{\mathcal{R}} \mathbb{T}$, if*

1. *for every E occurring in $\mathbb{T}$ there exists X such that $\Gamma \vdash_{\mathcal{R}} E\!:\!X$, and*
2. *for all valid sets $\mathcal{L}$ of positions in $\mathbb{T}$:*

$$\biguplus_{\alpha.k \prec \mathcal{L}'} [\mathbb{T}(\alpha.k)] + \biguplus_{\alpha.k \in \mathcal{L}'} \mathsf{ij}_{\mathbb{T}}(\alpha.k) \subseteq \mathcal{R}$$

where $\mathcal{L}'$ is the set of all positions at the top of leaves of subtree $\mathbb{T}|_{\mathcal{L}}$, that is, $\mathcal{L}' = \{\alpha.\mathsf{hi}(\mathbb{T}|_{\mathcal{L}}(\alpha)) \mid \alpha \in \mathsf{leaves}(\mathbb{T}|_{\mathcal{L}})\}$.

The formal definitions of terminal configurations and stuck states are the same as in previous chapters.

Definition 6.3.2 (Terminal configurations). *A configuration $\mathbb{T}$ is terminal if it has the form (M, ϵ).*

Definition 6.3.3 (Stuck states). *A configuration $\mathbb{T}$ is stuck if no transition rule applies and $\mathbb{T}$ is not terminal.*

The two main lemmas Preservation and Progress are stated as follows.

Lemma 6.3.4 (Preservation). *If $\Gamma \models_{\mathcal{R}} \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then $\Gamma \models_{\mathcal{R}} \mathbb{T}'$.*

Lemma 6.3.5 (Progress). *If $\Gamma \models_{\mathcal{R}} \mathbb{T}$, then either $\mathbb{T}$ is terminal or there exists configuration $\mathbb{T}'$ such that $\mathbb{T} \longrightarrow \mathbb{T}'$.*

Finally, the type soundness property allows us to safely execute well-typed component programs. That is, during the execution of the programs the number of active instances of any component never exceeds the allowed number. We denote by $[\mathbb{T}]$ the multiset of all active instances in $\mathbb{T}$:

$$[\mathbb{T}] = \biguplus_{\alpha \in \mathbb{T}} [\mathbb{T}(\alpha)]$$

Theorem 6.3.6 (Soundness). *If program $Prog = Decls; E$ is well-typed with respect to a requirement $\mathcal{R}$, then for any $\mathbb{T}$ such that $\mathsf{Lf}([], E) \longrightarrow^* \mathbb{T}$ we have $\mathbb{T}$ is not stuck and $[\mathbb{T}] \subseteq \mathcal{R}$.*

To prove the above lemmas, we will prove the following additional invariants. On one hand, these invariants give us a better understanding about the behaviour of the operational semantics. On the other hand, they simplify the proof of Lemma 6.3.4.

Lemma 6.3.7 (Well-typedness of expressions under reduction). *If $\Gamma \models_{\mathcal{R}} \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then for every E occurring in $\mathbb{T}'$, there exists X such that $\Gamma \vdash_{\mathcal{R}} E\!:\!X$.*

Lemma 6.3.8 (Invariant of reul**).** *If* $\Gamma \models_{\mathcal{R}} \mathbb{T}$ *and* $\mathbb{T} \longrightarrow \mathbb{T}'$*, then for all positions* $\alpha.k$ *occurring in both* $\mathbb{T}$ *and* $\mathbb{T}'$*:*

$$z \in \mathsf{reul}_{\mathbb{T}}(\alpha.k) \textit{ implies } z \in \mathsf{reul}_{\mathbb{T}'}(\alpha.k)$$

Lemma 6.3.9 (Invariant of retop**).** *If* $\Gamma \models_{\mathcal{R}} \mathbb{T}$ *and* $\mathbb{T} \longrightarrow \mathbb{T}'$*, then for all positions* $\alpha.k$ *occurring in both* $\mathbb{T}$ *and* $\mathbb{T}'$*:*

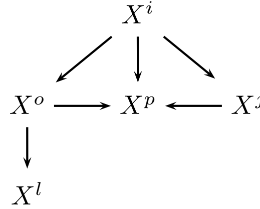
$$[\mathbb{T}(\alpha.k)] + \mathsf{op}_{\mathbb{T}}(\alpha.k) + \mathsf{retop}_{\mathbb{T}}(\alpha.k) \supseteq [\mathbb{T}'(\alpha.k)] + \mathsf{op}_{\mathbb{T}'}(\alpha.k) + \mathsf{retop}_{\mathbb{T}'}(\alpha.k)$$

6.3.2 Typing Properties

This section lists some fundamental properties of the type system. Most of these properties are analogous to those in Chapter 5. We start by giving some definitions. We use X^* for any of X^i, X^o, X^j, X^p and X^l. The function var is updated from the definition in Chapter 5.3.2 as follows.

$$\mathsf{var}((A \parallel B)) = \mathsf{var}(A) \cup \mathsf{var}(B)$$

The following lemma collects a number of simple properties of a typing judgment. It also shows some relations among multisets of types and any legal basis always has distinct declarations. The inclusion relations between multisets of a type expression are depicted by the following picture. in the picture The arrows point to smaller multisets.



Lemma 6.3.10 (Valid typing judgment). *If* $\Gamma \vdash A : X$*, then*

1. *elements of* $\mathsf{var}(A)$*,* X^* *are in* $\mathsf{dom}(\Gamma)$*,*
2. $\Gamma \vdash \epsilon : \langle [], [], [], [], [] \rangle$*,*
3. *every variable in* $\mathsf{dom}(\Gamma)$ *is declared only once in* Γ*,*
4. $X^o \subseteq X^i \subseteq \mathcal{R}$ *and* $X^p \subseteq X^j \subseteq \mathcal{R}$*,*
5. $X^l(z) = 1$ *for all* z*,*
6. $X^j \subseteq X^i$*,* $X^p \subseteq X^o$*, and* $X^l \subseteq X^o$*.*

Proof. By simultaneous induction on derivation.

- Base case Axiom: Since $\mathsf{var}(\epsilon) = \mathsf{dom}([]) = \mathsf{dom}(\varnothing) = \{\}$, all the clauses hold easily.
- Case WeakenB:

$$\frac{\Gamma' \vdash A : X \quad \Gamma' \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x \prec B \vdash A : X}\ (\text{WeakenB})$$

  Clause 1 holds by the induction hypothesis. Clause 2 holds by applying WeakenB to the induction hypothesis $\Gamma' \vdash \epsilon : \langle [], [], [], [], [] \rangle$. Clause 3 follows by the side condition and the induction hypothesis. Clauses 4, 5 and 6 hold by the induction hypothesis.

- Case New:

$$\text{(New)}\quad \frac{\Gamma' \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma', x \prec B \vdash \mathtt{new}\, x : \langle Y^i + x, Y^o + x, Y^j + x, Y^p + x, Y^l + x\rangle}$$

with $\Gamma = \Gamma', x \prec B$ and $X^* = Y^* + x$. Assume the lemma is correct for the premise of this rule, so elements of $\mathsf{var}(B)$, Y^* are in $\mathsf{dom}(\Gamma')$. Clause 1 holds easily as the new element x in $\mathsf{var}(\mathtt{new}\, x)$ and X^* is in $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma', x \prec B) = \mathsf{dom}(\Gamma') \cup \{x\}$. Clause 2 follows by applying WeakenB. Clause 3 follows by the side condition and the induction hypothesis. Clauses 4, 5 and 6 hold by the induction hypothesis for all $z \neq x$. For $z = x$, by the induction hypothesis elements of Y^* are in $\mathsf{dom}(\Gamma')$ and so they are different from x. Hence $X^*(x) = (Y^* + x)(x) = 1 \leq \mathcal{R}(x)$.

- Case Reu:

$$\text{(Reu)}\quad \frac{\Gamma' \vdash B : Y \quad x \notin \mathsf{dom}(\Gamma')}{\Gamma, x \prec B \vdash \mathtt{reu}\, x : \langle Y^i + x, Y^o + x, Y^j, Y^p, Y^l + x\rangle}$$

with $\Gamma = \Gamma', x \prec B$, $X = \langle Y^i + x, Y^o + x, Y^j, Y^p, Y^l + x\rangle$. All clauses can be proved analogously as in the case New.

- Case Seq:

$$\text{(Seq)}\quad \frac{\Gamma \vdash B : Y \quad \Gamma \vdash C : Z \quad Y^o + Z^j \subseteq \mathcal{R} \quad B, C \neq \epsilon}{\Gamma \vdash BC : \langle Y^i \cup (Y^o{+}Z^j) \cup Z^i!_{Y^l}, (Y^o + Z^p) \cup Z^o!_{Y^l}, Y^j \cup (Y^p{+}Z^j), Y^p + Z^p, Y^l \cup Z^l\rangle}$$

Clauses 1, 2 and 3 hold by the induction hypothesis.

For clause 4, first we prove $(Y^o + Z^p) \cup Z^o!_{Y^l} \subseteq Y^i \cup (Y^o{+}Z^j) \cup Z^i!_{Y^l}$ as follows.

$$\begin{aligned}
(Y^o + Z^p) \cup Z^o!_{Y^l} &\subseteq (Y^o + Z^p) \cup Z^i!_{Y^l} && (Z^o \subseteq Z^i \text{ by the induction hypothesis})\\
&\subseteq (Y^o + Z^j) \cup Z^i!_{Y^l} && (Z^p \subseteq Z^j \text{ by the induction hypothesis})\\
&\subseteq Y^i \cup (Y^o + Z^j) \cup Z^i!_{Y^l} && (\text{definition of } \cup)
\end{aligned}$$

Second, we need to prove that $Y^i \cup (Y^o{+}Z^j) \cup Z^i!_{Y^l} \subseteq \mathcal{R}$. We have $Y^i \subseteq \mathcal{R}$ and $Z^i \subseteq \mathcal{R}$ by the induction hypothesis; $Y^o + Z^j \subseteq \mathcal{R}$ by the side condition.

The second part of clause 4, $Y^p + Z^p \subseteq Y^j \cup (Y^p{+}Z^j) \subseteq \mathcal{R}$, is proved as follows.

$$\begin{aligned}
Y^p + Z^p &\subseteq Y^p + Z^j && (Z^p \subseteq Z^j \text{ by the induction hypothesis})\\
&\subseteq Y^j \cup (Y^p + Z^j) && (\text{definition of } \cup)
\end{aligned}$$

So the first inequality holds. The second inequality holds since $Y^j \subseteq Y^i \subseteq \mathcal{R}$ by the induction hypothesis and $Y^p + Z^j \subseteq Y^o + Z^j \subseteq \mathcal{R}$ by the side condition. Clause 5 follows by the induction hypothesis. For clause 6, we need to prove that

$$\begin{aligned}
Y^j \cup (Y^p{+}Z^j) &\subseteq Y^i \cup (Y^o{+}Z^j) \cup Z^i!_{Y^l}\\
Y^p + Z^p &\subseteq (Y^o + Z^p) \cup Z^o!_{Y^l}\\
Y^l \cup Z^l &\subseteq (Y^o + Z^p) \cup Z^o!_{Y^l}
\end{aligned}$$

We prove the first one as follows.

$$\begin{aligned}
Y^j \cup (Y^p{+}Z^j) &\subseteq Y^i \cup (Y^p + Z^j) && (Y^j \subseteq Y^i \text{ by the induction hypothesis})\\
&\subseteq Y^i \cup (Y^o + Z^j) && (Y^j \subseteq Y^i \text{ by the induction hypothesis})\\
&\subseteq Y^i \cup (Y^o{+}Z^j) \cup Z^i!_{Y^l} && (\text{definition of } \cup)
\end{aligned}$$

The second one follows analogously by the induction hypothesis $Y^p \subseteq Y^o$.

For the last one, by clause 5, we only need to show that $z \in Y^l \cup Z^l$ implies $z \in (Y^o + Z^p) \cup Z^o!_{Y^l}$. If $x \in Y^l$, then $x \in Y^o$ since $Y^l \subseteq Y^o$ by the induction hypothesis, thus the clause follows. Otherwise $x \notin Y^l$ and $x \in Z^l$, then $x \in Z^o!_{Y^l}$, thus the clause also follows.

- Case Choice:

$$\text{(Choice)}\quad \frac{\Gamma \vdash C : Z \quad \Gamma \vdash B : Y}{\Gamma \vdash (C + B) : \langle Z^i \cup Y^i, Z^o \cup Y^o, Z^j \cup Y^j, Z^p \cup Y^p, Z^l \cap Y^l \rangle}$$

Clauses 1, 2 and 3 hold by the induction hypothesis. Clause 4 holds since $Y^o \subseteq Y^i \subseteq \mathcal{R}$ and $Z^o \subseteq Z^i \subseteq \mathcal{R}$ by the induction hypothesis imply $Y^o \cup Z^o \subseteq Y^i \cup Z^i \subseteq \mathcal{R}$. The rest goes similarly.

- Case Parallel:

$$\text{(Parallel)}\quad \frac{\Gamma \vdash C : Z \quad \Gamma \vdash B : Y \quad Z^i + Y^i \subseteq \mathcal{R}}{\Gamma \vdash (C \parallel B) : \langle Z^i + Y^i, Z^o + Y^o, Z^j + Y^j, Z^p + Y^p, Z^l \cup Y^l \rangle}$$

Clauses 1, 2 and 3 hold by the induction hypothesis. The first part of clause 4 holds since $Y^o \subseteq Y^i$ and $Z^o \subseteq Z^i$ by the induction hypothesis imply $Y^o + Z^o \subseteq Y^i + Z^i$ and $Y^i + Z^i \subseteq \mathcal{R}$ by the side condition. The second part of this clause is proved analogously. Clauses 5 and 6 follow easily by the induction hypothesis.

- Case Scope:

$$\text{(Scope)}\quad \frac{\Gamma \vdash B : Y}{\Gamma \vdash \{B\} : \langle Y^i, [], Y^j, [], [] \rangle}$$

Clauses 1, 2 and 3 hold by the induction hypothesis. Clauses 4 and 5 are trivial. The first inequality of clause 6 follows by the induction hypothesis. The second one is trivial.

□

Lemma 6.3.11 (Associativity). *If $\Gamma \vdash A_i : X_i$, for $i \in \{1, 2, 3\}$, then the types of $(A_1 A_2) A_3$ and $A_1(A_2 A_3)$, if typable, are the same.*

Proof. As in proof of Lemma 2.3.10, we assume that $A_i \neq \epsilon$ for all i. By the sequencing rule Seq, we have $\Gamma \vdash A_1 A_2 : X$ with

$$X = \langle X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l}, (X_1^o + X_2^p) \cup X_2^o!_{X_1^l}, X_1^j \cup (X_1^p + X_2^j), X_1^p + X_2^p, X_1^l \cup X_2^l \rangle$$

and $\Gamma \vdash A_2 A_3 : Z$ with

$$Z = \langle X_2^i \cup (X_2^o + X_3^j) \cup X_3^i!_{X_2^l}, (X_2^o + X_3^p) \cup X_3^o!_{X_2^l}, X_2^j \cup (X_2^p + X_3^j), X_2^p + X_3^p, X_2^l \cup X_3^l \rangle$$

Continue applying the rule Seq, we get the types of $(A_1 A_2) A_3$ and $A_1(A_2 A_3)$ and to prove that the two types are the same, we need to prove five equations:

$$\begin{aligned} X^i \cup (X^o + X_3^j) \cup X_3^i!_{X^l} &= X_1^i \cup (X_1^o + Z^j) \cup Z^i!_{X_1^l} \\ (X^o + X_3^p) \cup X_3^o!_{X^l} &= (X_1^o + Z^p) \cup Z^o!_{X_1^l} \\ X^j \cup (X^p + X_3^j) &= X_1^j \cup (X_1^p + Z^j) \\ X^p + X_3^p &= X_1^p + Z^p \\ X^l \cup X_3^l &= X_1^l \cup Z^l \end{aligned}$$

First, we prove the following equation, which will be used in proofs of the first two above equations.

$$(M+N)\cup(X_2^o+N)!_{X_1^l}=(M+N)\cup((X_2^o!_{X_1^l})+N) \tag{6.3}$$

To prove this result, we divide into two cases. First, if $x\in X_1^l$, then $LHS(x)=(M+N)(x)=((M+N)\cup N)(x)=RHS(x)$. Otherwise, $x\notin X_1^l$, then $(X_2^o+N)!_{X_1^l}(x)=(X_2^o+N)(x)=X_2^o!_{X_1^l}(x)+N(x)$. Hence, Equation 6.3 follows.

We prove the first equation as follows. Note that $(N_1\cup N_2)!_M=N_1!_M\cup N_2!_M$ and $(N!_{M_2})!_{M_1}=N!_{(M_1\cup M_2)}$.

$$\begin{aligned}
&X^i\cup(X^o{+}X_3^j)\cup X_3^i!_{X^l}\\
&=X_1^i\cup(X_1^o{+}X_2^j)\cup X_2^i!_{X_1^l}\cup(((X_1^o+X_2^p)\cup X_2^o!_{X_1^l})+X_3^j)\cup X_3^i!_{(X_1^l\cup X_2^l)}\\
&=X_1^i\cup(X_1^o{+}X_2^j)\cup X_2^i!_{X_1^l}\cup(X_1^o+X_2^p+X_3^j)\cup(X_2^o!_{X_1^l}+X_3^j)\cup X_3^i!_{(X_1^l\cup X_2^l)}\\
&=X_1^i\cup(X_1^o+X_2^j)\cup(X_1^o+X_2^p+X_3^j)\cup X_2^i!_{X_1^l}\cup(X_2^o+X_3^j)!_{X_1^l}\cup X_3^i!_{(X_1^l\cup X_2^l)}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(by Equation (6.3))}\\
&=X_1^i\cup(X_1^o+X_2^j)\cup(X_1^o+X_2^p+X_3^j)\cup(X_2^i\cup(X_2^o{+}X_3^j)\cup X_3^i!_{X_2^l})!_{X_1^l}\\
&=X_1^i\cup(X_1^o+(X_2^j\cup(X_2^p{+}X_3^j)))\cup(X_2^i\cup(X_2^o{+}X_3^j)\cup X_3^i!_{X_2^l})!_{X_1^l}\\
&=X_1^i\cup(X_1^o{+}Z^j)\cup Z^i!_{X_1^l}
\end{aligned}$$

The second equation is proved analogously.

$$\begin{aligned}
&(X^o+X_3^p)\cup X_3^o!_{X^l}\\
&=(((X_1^o+X_2^p)\cup X_2^o!_{X_1^l})+X_3^p)\cup X_3^o!_{(X_1^l\cup X_2^l)}\\
&=(X_1^o+X_2^p+X_3^p)\cup(X_2^o!_{X_1^l}+X_3^p)\cup X_3^o!_{(X_1^l\cup X_2^l)}\\
&=(X_1^o+X_2^p+X_3^p)\cup(X_2^o+X_3^p)!_{X_1^l}\cup X_3^o!_{(X_1^l\cup X_2^l)} &&\text{(by Equation (6.3))}\\
&=(X_1^o+(X_2^p+X_3^p))\cup((X_2^o+X_3^p)\cup X_3^o!_{X_2^l})!_{X_1^l}\\
&=(X_1^o+Z^p)\cup Z^o!_{X_1^l}
\end{aligned}$$

The third and the fourth equations are proved as in the proof of Lemma 5.3.8.

The last equation holds easily: $X^l\cup X_3^l=X_1^l\cup X_2^l\cup X_3^l=X_1^l\cup Z^l$. □

Lemma 6.3.12 (Generation).

1. *If* $\Gamma\vdash \texttt{new}\,x{:}X$, *then* $x\in X^p$ *and there exist* Δ, Δ', A *and* Y *such that* $\Gamma=\Delta,x{\prec}A,\Delta'$ *and* $\Delta\vdash A{:}Y$ *and* $X=\langle Y^i+x,Y^o+x,Y^j+x,Y^p+x,Y^l+x\rangle$.
2. *If* $\Gamma\vdash \texttt{reu}\,x{:}X$, *then* $x\in X^o$ *and there exist* Δ, Δ', A *and* Y *such that* $\Gamma=\Delta,x{\prec}A,\Delta'$, *and* $\Delta\vdash A{:}Y$ *with* $X=\langle Y^i+x,Y^o+x,Y^j,Y^p,Y^l+x\rangle$.
3. *If* $\Gamma\vdash AB{:}Z$ *with* $A,B\neq\epsilon$, *then there exist* X, Y *such that* $\Gamma\vdash A{:}X$, $\Gamma\vdash B{:}Y$ *and* $Z=\langle X^i\cup(X^o{+}Y^j)\cup Y^i!_{X^l},(X^o+Y^p)\cup Y^o!_{X^l},X^j\cup(X^p{+}Y^j),X^p+Y^p,X^l\cup Y^l\rangle$.
4. *If* $\Gamma\vdash(A+B):Z$, *then there exist* X, Y *such that* $\Gamma\vdash A:X$, $\Gamma\vdash B:Y$ *and* $Z=\langle X^i\cup Y^i,X^o\cup Y^o,X^j\cup Y^j,X^p\cup Y^p,X^l\cap Y^l\rangle$.
5. *If* $\Gamma\vdash(A\parallel B):Z$, *then there exist* X, Y *such that* $\Gamma\vdash A:X$, $\Gamma\vdash B:Y$, *and* $Z=\langle X^i+Y^i,X^o+Y^o,X^j+Y^j,X^p+Y^p,X^l\cup Y^l\rangle$.
6. *If* $\Gamma\vdash\{A\}{:}Z$, *then there exists* X *such that* $\Gamma\vdash A{:}X$ *and* $Z=\langle X^i,[],X^j,[],[]\rangle$.

Proof. The proof is analogous to the proof of Lemma 2.3.11. □

Lemma 6.3.13 (Weakening).

1. *If* $\Gamma = \Delta, x \prec E, \Delta'$ *is legal, then* $\Delta \vdash E : X$ *for some* X.
2. *If* $\Gamma \vdash E : X$ *and* Γ *is an initial segment of a legal basis* Γ'*, then* $\Gamma' \vdash E : X$.

Proof. The proof is the same as the proof of Lemma 2.3.12. □

Lemma 6.3.14 (Strengthening). *If* $\Gamma, x \prec A \vdash B : Y$ *and* $x \notin \mathsf{var}(B)$*, then* $\Gamma \vdash B : Y$ *and* $x \notin Y^i$.

Proof. The proof is same as the proof of Lemma 2.3.13. □

Proposition 6.3.15 (Uniqueness of types). *If* $\Gamma \vdash A : X$ *and* $\Gamma \vdash A : Y$*, then* $X^i = Y^i$, $X^o = Y^o$, $X^j = Y^j$, $X^p = Y^p$*, and* $X^l = Y^l$.

Proof. The proof is analogous to the proof of Proposition 2.3.14. □

6.4 Soundness Proofs

Proof of Lemma 6.3.7 (Well-typedness of expressions under reduction). *If* $\Gamma \models_{\mathcal{R}} \mathbb{T}$ *and* $\mathbb{T} \longrightarrow \mathbb{T}'$*, then every* E *occurring in* $\mathbb{T}'$ *is well-typed with respect to* Γ, $\mathcal{R}$.

Proof. The proof proceeds by case analysis on the reduction relation: $\longrightarrow$. We will consistently use location β for the node where the reduction occurs and let $n = \mathsf{hi}(\mathbb{T}(\beta))$.

- Case osNew:

$$\begin{array}{l}(\mathsf{osNew}) \quad x \prec A \in \mathit{Decls} \\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{new}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\beta\end{array}$$

  Since the only new expression in $\mathbb{T}'$ is AE at $\beta.n$, we only need to prove that AE is well-typed. By the hypothesis, $\mathtt{new}\, xE \in \mathbb{T}$ is well-typed, thus there exists X such that $\Gamma \vdash \mathtt{new}\, xE : X$. By Generation Lemma 6.3.12, clause 3 applied to $\Gamma \vdash \mathtt{new}\, xE : X$, we have $\Gamma \vdash \mathtt{new}\, x : X_1$ and $\Gamma \vdash E : X_2$ with

$$X = \langle X_1^i \cup (X_1^o + X_2^j) \cup X_2^i !_{X_1^l}, (X_1^o + X_2^p) \cup X_2^o !_{X_1^l}, X_1^j \cup (X_1^p + X_2^j), X_1^p + X_2^p, X_1^l \cup X_2^l \rangle$$

  Also by Generation Lemma 6.3.12, clause 1 applied to $\Gamma \vdash \mathtt{new}\, x : X_1$, we have $\Gamma \vdash A : Y$ with $X_1^* = Y^* + x$. Now we can derive $\Gamma \vdash AE : Z$ by the rule Seq since the side condition $Y^o + X_2^j \subseteq \mathcal{R}$ holds by $Y^o \subset X_1^o$ and $X_1^o + X_2^j \subseteq \mathcal{R}$, and we get:

$$Z = \langle Y^i \cup (Y^o + X_2^j) \cup X_2^i !_{Y^l}, (Y^o + X_2^p) \cup X_2^o !_{Y^l}, Y^j \cup (Y^p + X_2^j), Y^p + X_2^p, Y^l \cup X_2^l \rangle$$

- Case osReu1:

$$\begin{array}{l}(\mathsf{osReu1}) \quad x \prec A \in \mathit{Decls} \quad x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.\mathsf{hi}(\mathbb{T}(\beta))) \\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{reu}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\beta\end{array}$$

  Analogous to the case osNew, the proof of AE well-typed is as follows. Since $\mathtt{reu}\, xE \in \mathbb{T}$ is well-typed, there exists X such that $\Gamma \vdash \mathtt{reu}\, xE : X$. By Generation Lemma 6.3.12, clause 3, we get $\Gamma \vdash \mathtt{reu}\, x : X_1$ and $\Gamma \vdash E : X_2$ with $X = \langle X_1^i \cup (X_1^o + X_2^j) \cup X_2^i !_{X_1^l}, (X_1^o + X_2^p) \cup X_2^o !_{X_1^l}, X_1^j \cup (X_1^p + X_2^j), X_1^p +$

$X_2^p, X_1^l \cup X_2^l\rangle$. Also by Generation Lemma 6.3.12, clause 2, we have $\Gamma \vdash A : Y$ with $X_1 = \langle Y^i + x, Y^o + x, Y^j, Y^p, Y^l + x\rangle$ and we can derive $\Gamma \vdash AE : Z$ with $Z = \langle Y^i \cup (Y^o{+}X_2^j) \cup X_2^i!_{Y^l}, (Y^o + X_2^p) \cup X_2^o!_{Y^l}, Y^j \cup (Y^p{+}X_2^j), Y^p + X_2^p, Y^l \cup X_2^l\rangle$ since the side condition $Y^o{+}X_2^j \subseteq \mathcal{R}$ holds by $Y^o \subset Y^o{+}x = X_1^o$ and $X_1^o{+}X_2^j \subseteq \mathcal{R}$.

- Case osReu2:

$$\frac{(\mathsf{osReu2}) \quad x{\prec}A \in \mathit{Decls} \quad x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.\mathsf{hi}(\mathbb{T}(\beta)))}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{reu}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\beta}$$

The proof is the same as in the case osReu1.

- Case osChoice:

$$\frac{(\mathsf{osChoice})}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, (A+B)E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\beta}$$

Since the only new expression in $\mathbb{T}'$ is AE at $\beta.n$, we only need to prove that AE is well-typed. Since $(A+B)E \in \mathbb{T}$ is well-typed by the hypothesis, there exists X such that $\Gamma \vdash (A+B)E : X$. By Generation Lemma 6.3.12, clause 3 applied to $\Gamma \vdash (A+B)E : X$, we get $\Gamma \vdash (A+B) : X_1$ and $\Gamma \vdash E : X_2$ with $X = \langle X_1^i \cup (X_1^o{+}X_2^j) \cup X_2^i!_{X_1^l}, (X_1^o + X_2^p) \cup X_2^o!_{X_1^l}, X_1^j \cup (X_1^p{+}X_2^j), X_1^p + X_2^p, X_1^l \cup X_2^l\rangle$. Also by Generation Lemma 6.3.12, clause 4, we have $\Gamma \vdash A : Y_1$, $\Gamma \vdash B : Y_2$ with $X_1 = \langle Y_1^i \cup Y_2^i, Y_1^o \cup Y_2^o, Y_1^j \cup Y_2^j, Y_1^p \cup Y_2^p, Y_1^l \cap Y_2^l\rangle$. Then we can derive $\Gamma \vdash AE : Z$ with $Z = \langle Y_1^i \cup (Y_1^o{+}X_2^j) \cup X_2^i!_{Y_1^l}, (Y_1^o + X_2^p) \cup X_2^o!_{Y_1^l}, Y_1^j \cup (Y_1^p{+}X_2^j), Y_1^p + X_2^p, Y_1^l \cup X_2^l\rangle$ since the side condition holds by $Y_1^o + X_2^p \subseteq (Y_1^o \cup Y_2^o) + X_2^p = X_1^o + X_2^p \subseteq \mathcal{R}$.

- Case osPush:

$$\frac{(\mathsf{osPush})}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \{A\}E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ ([], A))]\!]_\beta}$$

Since the only new expressions in $\mathbb{T}'$ are E at $\beta.n$ and A at $\beta.(n+1)$, we only need to prove that A and E are well-typed. Since $\{A\}E$ is well-typed by hypothesis, there exists X such that $\Gamma \vdash \{A\}E : X$. By Generation Lemma 6.3.12, clause 3 and 6, we get $\Gamma \vdash \{A\} : X_1$ and $\Gamma \vdash E : X_2$ with $X = \langle X_1^i \cup (X_1^o{+}X_2^j) \cup X_2^i!_{X_1^l}, (X_1^o + X_2^p) \cup X_2^o!_{X_1^l}, X_1^j \cup (X_1^p{+}X_2^j), X_1^p + X_2^p, X_1^l \cup X_2^l\rangle$ and $X_1^o = X_1^p = X_1^l = []$. Therefore, $X^o = X_2^o$, $X^p = X_2^p$, and $X^l = X_2^l$. Also by Generation Lemma 6.3.12, clause 6 applied to $\Gamma \vdash \{A\} : X_1$, we get $\Gamma \vdash A : Y$ with $Y^i = X_1^i$ and $Y^j = X_1^j$.

- Case osPop:

$$\frac{(\mathsf{osPop})}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ (M', \epsilon))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E))]\!]_\beta}$$

The clause holds by the hypothesis since there are no new expressions in $\mathbb{T}'$.

- Case osParIntr:

$$\frac{(\mathsf{osParIntr})}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, (A \parallel B)E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B)))]\!]_\beta}$$

Analogous to the previous cases, we only need to prove that A, B, and E are well-typed. The proof is analogous to the case osChoice. Since $(A \parallel B)E \in \mathbb{T}$ is well-typed by the hypothesis, there exists X such that $\Gamma \vdash (A \parallel B)E : X$. By Generation Lemma 6.3.12, clause 3 applied to $\Gamma \vdash (A \parallel B)E : X$, we get $\Gamma \vdash (A \parallel B) : X_1$

and $\Gamma \vdash E : X_2$ with $X = \langle X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l}, (X_1^o + X_2^p) \cup X_2^o!_{X_1^l}, X_1^j \cup (X_1^p + X_2^j), X_1^p + X_2^p, X_1^l \cup X_2^l\rangle$. Also by Generation Lemma 6.3.12, clause 5, we have $\Gamma \vdash A : Y_1$, $\Gamma \vdash B : Y_2$ with $X_1 = \langle Y_1^i + Y_2^i, Y_1^o + Y_2^o, Y_1^j + Y_2^j, Y_1^p + Y_2^p, Y_1^l \cup Y_2^l\rangle$.

- Case osParElim1:

$$\begin{array}{l}(\mathsf{osParElim1})\\ \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathbb{R}, \mathsf{Lf}((M', \epsilon)))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})]\!]_\beta\end{array}$$

  The clause holds by the hypothesis.

- Case osParElim2:

$$\begin{array}{l}(\mathsf{osParElim2})\\ \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + M', E))]\!]_\beta\end{array}$$

  The clause holds by the hypothesis.

- Case osCong:

$$\begin{array}{l}(\mathsf{osCong}) \quad \mathbb{R} \equiv \mathbb{R}'\\ \mathbb{T}[\![\mathbb{R}]\!]_\beta \longrightarrow \mathbb{T}[\![\mathbb{R}']\!]_\beta\end{array}$$

  The clause holds by the hypothesis.

□

Proof of Lemma 6.3.8 (Invariant of reul). *If $\Gamma \models_{\mathcal{R}} \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then for all positions $\alpha.k$ occurring in both $\mathbb{T}$ and $\mathbb{T}'$:*

$$z \in \mathsf{reul}_{\mathbb{T}}(\alpha.k) \ \textit{implies}\ z \in \mathsf{reul}_{\mathbb{T}'}(\alpha.k)$$

Proof. The proof proceeds by case analysis on the reduction relation $\longrightarrow$.

In the Lemma 6.3.7, we have proved that all the expressions in $\mathbb{T}'$ are well-typed. In this proof, we assume that the expressions in $\mathbb{T}$ and $\mathbb{T}'$ have the types as in the proof of Lemma 6.3.7. We will consistently use location β for the node where the reduction occurs and let $n = \mathsf{hi}(\mathbb{T}(\beta))$.

Since the definition of $\mathsf{reul}_{\mathbb{T}}(\alpha.k)$ only involves the nodes α' on the path from the root to α: $\alpha' \preccurlyeq \beta$, and the nodes in the branch starting from α: $\alpha' \succ \beta$, we are only interested in these positions.

Case osNew:

$$\begin{array}{l}(\mathsf{osNew}) \quad x \prec A \in \mathit{Decls}\\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{new}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\beta\end{array}$$

We divide into the following cases.

- Case $\alpha.k = \beta.n$, we have

$$\begin{aligned}\mathsf{reul}_{\mathbb{T}}(\beta.n) &= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup \mathsf{retl}_{\mathbb{T}}(\beta.n) && \text{(definition of reul)}\\ &= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup [] && (\beta l, \beta r \notin \mathbb{T})\\ &\subset \mathsf{reuLf}_{\mathbb{T}'}(\beta.n) && (\mathsf{reuLf}_{\mathbb{T}'}(\beta.n) = \mathsf{reuLf}_{\mathbb{T}}(\beta.n) + x)\\ &= \mathsf{reul}_{\mathbb{T}'}(\beta.n) && (\beta l, \beta r \notin \mathbb{T}')\end{aligned}$$

- Case $\alpha.k = \beta.k$ and $k < n$, we have

$$\begin{aligned} \mathsf{reul}_{\mathbb{T}}(\beta.k) &= \mathsf{reuLf}_{\mathbb{T}}(\beta.k) \cup \mathsf{retl}_{\mathbb{T}}(\beta.k) \\ &= \mathsf{reuLf}_{\mathbb{T}}(\beta.k) \cup [] && (k < n,\ \mathsf{retl}_{\mathbb{T}}(\beta.k) = []) \\ &= \mathsf{reuLf}_{\mathbb{T}'}(\beta.k) && (\text{definition of } \mathsf{reuLf}) \\ &= \mathsf{reul}_{\mathbb{T}'}(\beta.k) && (k < n) \end{aligned}$$

  So the clause holds.

- Case $\alpha \prec \beta$, if $k < \mathsf{hi}(\mathbb{T}(\alpha))$ or there exists $\alpha' \succ \alpha$ such that $\mathsf{hi}(\mathbb{T}(\alpha')) > 1$, then $\beta.n$ is not involved in the computation of both $\mathsf{retl}_{\mathbb{T}}(\alpha.k)$ and $\mathsf{retl}_{\mathbb{T}'}(\alpha.k)$. Hence $\mathsf{reul}_{\mathbb{T}}(\alpha.k) = \mathsf{reul}_{\mathbb{T}'}(\alpha.k)$ since the two trees $\mathbb{T}$ and $\mathbb{T}'$ are only different at $\beta.n$.

  Otherwise, $k = \mathsf{hi}(\mathbb{T}(\alpha))$ and for all α' such that $\alpha \prec \alpha' \preccurlyeq \beta$, we have $\mathsf{hi}(\mathbb{T}(\alpha')) = 1$, then since the two trees are only different at $\beta.n$ ($n = 1$), we only need to show that

$$z \in [\mathbb{T}(\beta.1)] \cup X^l \cup \mathsf{retl}_{\mathbb{T}}(\beta.1) \text{ implies } z \in [\mathbb{T}'(\beta.1)] \cup Z^l \cup \mathsf{retl}_{\mathbb{T}'}(\beta.1)$$

  If $z \neq x$ the clause holds easily since the modification involves only x. So we only need to prove for $z = x$. Since $x \in X^l$ and $x \in [\mathbb{T}'(\beta.1)]$ the clause holds.

Case osReu1:

$$\begin{array}{l} (\mathsf{osReu1}) \quad x{\prec}A \in \mathit{Decls} \ \ x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.\mathsf{hi}(\mathbb{T}(\beta))) \\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M,\, \mathtt{reu}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\beta \end{array}$$

The same as the case osNew.

Case osReu2:

$$\begin{array}{l} (\mathsf{osReu2}) \quad x{\prec}A \in \mathit{Decls} \ \ x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.\mathsf{hi}(\mathbb{T}(\beta))) \\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M,\, \mathtt{reu}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\beta \end{array}$$

We divide into the following cases.

- Case $\alpha.k = \beta.n$, the clause holds since

$$\begin{aligned} \mathsf{reul}_{\mathbb{T}}(\beta.n) &= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup \mathsf{retl}_{\mathbb{T}}(\beta.n) && (\text{definition of } \mathsf{reul}) \\ &= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup [] && (\beta l, \beta r \notin \mathbb{T}) \\ &= \mathsf{reuLf}_{\mathbb{T}'}(\beta.n) \\ &= \mathsf{reul}_{\mathbb{T}'}(\beta.n) && (\beta l, \beta r \notin \mathbb{T}') \end{aligned}$$

- Case $\alpha.k = \beta.k$ and $k < n$, the proof is the same as in the case osNew.

- Case $\alpha \prec \beta$, as in the case osNew, we only need to prove for the cases $k = \mathsf{hi}(\mathbb{T}(\alpha))$ and $\mathsf{hi}(\mathbb{T}(\alpha')) = 1$ for all α' such that $\alpha \prec \alpha' \preccurlyeq \beta$ that

$$z \in \mathsf{reul}_{\mathbb{T}}(\alpha.k) \text{ implies } z \in \mathsf{reul}_{\mathbb{T}'}(\alpha.k)$$

  If $z \neq x$ the clause holds easily since the modification involves only x. So we only prove for $z = x$, that is, we only need to show that $x \in \mathsf{reul}_{\mathbb{T}'}(\alpha.k)$.

  By the side condition we have $x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.n)$, so $x \in \mathsf{reuLf}_{\mathbb{T}}(\alpha.k)$ or $x \in [\mathbb{T}(\alpha'.1)]$ where $\alpha \prec \alpha' \preccurlyeq \beta$. That implies $x \in \mathsf{reuLf}_{\mathbb{T}'}(\alpha.k)$ or $x \in [\mathbb{T}'(\alpha'.1)]$. Hence $x \in \mathsf{reul}_{\mathbb{T}'}(\alpha.k)$ and the clause holds.

Case osChoice:

$$\begin{array}{l} (\mathsf{osChoice}) \\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, (A + B)E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\beta \end{array}$$

We divide into the following cases.

- Case $\alpha.k = \beta.n$, the proof is the same as in the case osReu2.
- Case $\alpha.k = \beta.k$ and $k < n$, the proof is the same as in the case osNew.
- Case $\alpha \prec \beta$, as in the case osNew, we only need to prove for the cases $k = \mathsf{hi}(\mathbb{T}(\alpha))$ and $\mathsf{hi}(\mathbb{T}(\alpha')) = 1$ for all α' such that $\alpha \prec \alpha' \preccurlyeq \beta$.

  Since the two trees $\mathbb{T}$ and $\mathbb{T}'$ are only different at $\beta.n$, we only need to show that

$$[\mathbb{T}(\beta.n)] \cup X^l \subseteq [\mathbb{T}'(\beta.n)] \cup Z^l$$

  Since $[\mathbb{T}(\beta.n)] = [\mathbb{T}'(\beta.n)] = M$, we have

$$\begin{aligned}
LHS &= M \cup X^l \\
&= M \cup X_1^l \cup X_2^l && (X^l = X_1^l \cup X_2^l) \\
&= M \cup (Y_1^l \cap Y_2^l) \cup X_2^l && (X_1^l = Y_1^l \cap Y_2^l) \\
&\subseteq M \cup Y_1^l \cup X_2^l && (Y_1^l \cap Y_2^l \subseteq Y_1^l) \\
&= M \cup Z^l && (Z^l = Y_1^l \cup X_2^l) \\
&= RHS
\end{aligned}$$

Case osPush:

$$\begin{array}{l}
(\mathsf{osPush}) \\
\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \{A\}E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ ([], A))]\!]_\beta
\end{array}$$

Since $\beta.(n+1) \notin \mathbb{T}$, we have the following cases.

- Case $\alpha.k = \beta.n$, we have

$$\begin{aligned}
\mathsf{reul}_{\mathbb{T}}(\beta.n) &= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup \mathsf{retl}_{\mathbb{T}}(\beta.n) && \text{(definition of } \mathsf{reul}) \\
&= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup [] && (\beta l, \beta r \notin \mathbb{T}) \\
&= \mathsf{reuLf}_{\mathbb{T}'}(\beta.n) \\
&= \mathsf{reul}_{\mathbb{T}'}(\beta.n) && (n < \mathsf{hi}(\mathbb{T}'(\beta)))
\end{aligned}$$

- Case $\alpha.k = \beta.k$ and $k < n$, the proof is the same as in the case osNew.
- Case $\alpha \prec \beta$, as in the case osNew, we only need to prove for the cases $k = \mathsf{hi}(\mathbb{T}(\alpha))$ and $\mathsf{hi}(\mathbb{T}(\alpha')) = 1$ for all α' such that $\alpha \prec \alpha' \preccurlyeq \beta$.

  Since the two trees are different at $\beta.n$ $(n = 1)$ and $\beta.(n+1)$, we only need to show that

$$[\mathbb{T}(\beta.n)] \cup X^l \subseteq [\mathbb{T}'(\beta.n)] \cup X_2^l$$

  This inequality holds by $X^l = X_2^l$ in the proof of Lemma 6.3.7 and $[\mathbb{T}(\beta.n)] = [\mathbb{T}'(\beta.n)] = M$.

Case osPop:

$$\begin{array}{l}
(\mathsf{osPop}) \\
\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ (M', \epsilon))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E))]\!]_\beta
\end{array}$$

Since $\beta.n \notin \mathbb{T}'$, we have the following cases.

- Case $\alpha.k = \beta.(n-1)$, we have

$$\begin{aligned}
\mathsf{reul}_{\mathbb{T}}(\beta.(n-1)) &= \mathsf{reuLf}_{\mathbb{T}}(\beta.(n-1)) \cup \mathsf{retl}_{\mathbb{T}}(\beta.(n-1)) && \text{(definition of } \mathsf{reul}) \\
&= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup [] && (n-1 < \mathsf{hi}(\mathbb{T}(\beta))) \\
&= \mathsf{reuLf}_{\mathbb{T}'}(\beta.n) \\
&= \mathsf{reul}_{\mathbb{T}'}(\beta.n) && (\beta l, \beta r \notin \mathbb{T}')
\end{aligned}$$

- Case $\alpha.k = \beta.k$ and $k < n$, the proof is the same as in the case osNew.
- Case $\alpha \prec \beta$, the clause holds since $\beta.n$ is not involved in the calculation of the function $\mathsf{reul}_{\mathbb{T}}(\alpha.k)$ and the two trees are only different at $\beta.n$ $(n > 1)$.

Case $\mathsf{osParIntr}$:

$$\begin{array}{l}(\mathsf{osParIntr})\\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, (A \parallel B)E))]\!]_{\beta} \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B)))]\!]_{\beta}\end{array}$$

Since $\beta l.1$ and $\beta r.1$ are not in $\mathbb{T}$, we have the following cases:

- Case $\alpha.k = \beta.n$, we have

$$\begin{aligned}\mathsf{reul}_{\mathbb{T}}(\beta.n) &= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup \mathsf{retl}_{\mathbb{T}}(\beta.n) && \text{(definition of } \mathsf{reul})\\ &= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup [] && (\beta l, \beta r \notin \mathbb{T})\\ &\subseteq \mathsf{reuLf}_{\mathbb{T}'}(\beta.n) \cup \mathsf{retl}_{\mathbb{T}'}(\beta.n)\\ &= \mathsf{reul}_{\mathbb{T}'}(\beta.n)\end{aligned}$$

- Case $\alpha.k = \beta.k$ and $k < n$, the proof is the same as in the case osNew.
- Case $\alpha \prec \beta$, as in the case osNew, we only need to prove for the cases $k = \mathsf{hi}(\mathbb{T}(\alpha))$ and $\mathsf{hi}(\mathbb{T}(\alpha')) = 1$ for all α' such that $\alpha \prec \alpha' \preccurlyeq \beta$.

  Since the two trees are only different at $\beta.n$, $\beta l.1$ and $\beta r.1$, we only need to show that
$$[\mathbb{T}(\beta.n)] \cup X^l \subseteq [\mathbb{T}'(\beta.n)] \cup X_2^l \cup \mathsf{retl}_{\mathbb{T}'}(\beta.n)$$

  We have

$$\begin{aligned}&\mathsf{retl}_{\mathbb{T}'}(\beta.n)\\ &= Y_1^l \cup \mathsf{retl}_{\mathbb{T}'}(\beta l.1) \cup Y_2^l \cup \mathsf{retl}_{\mathbb{T}'}(\beta r.1)\\ &= Y_1^l \cup Y_2^l && (\mathsf{retl}_{\mathbb{T}'}(\beta l.n) = \mathsf{retl}_{\mathbb{T}'}(\beta r.n) = [])\\ &= X_1^l\end{aligned}$$

  In addition, since $[\mathbb{T}(\beta.n)] = [\mathbb{T}'(\beta.n)]$ and $X^l = X_1^l \cup X_2^l$, the inequality follows.

Case $\mathsf{osParElim1}$:

$$\begin{array}{l}(\mathsf{osParElim1})\\ \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathbb{R}, \mathsf{Lf}((M', \epsilon)))]\!]_{\beta} \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})]\!]_{\beta}\end{array}$$

Suppose the position of $\mathsf{Lf}((M', \epsilon))$ is $\beta r.1$, then $\beta r.1 \notin \mathbb{T}'$ and we have the following cases.

- Case $\alpha.k \succ \beta.n$, that is, $\alpha.k$ is a position in $\mathbb{R}$, the clause follows since:

$$\begin{aligned}&\mathsf{reul}_{\mathbb{T}}(\alpha.k)\\ &= \mathsf{reuLf}_{\mathbb{T}}(\alpha.k) \cup \mathsf{retl}_{\mathbb{T}}(\alpha.k) && \text{(definition of } \mathsf{reul})\\ &\subseteq (\mathsf{reuLf}_{\mathbb{T}}(\alpha.k) + M') \cup \mathsf{retl}_{\mathbb{T}'}(\alpha.k) && (\mathsf{retl}_{\mathbb{T}}(\alpha.k) = \mathsf{retl}_{\mathbb{T}'}(\alpha.k))\\ &= \mathsf{reul}_{\mathbb{T}'}(\alpha.k) && \text{(definition of } \mathsf{reul})\end{aligned}$$

- Case $\alpha.k = \beta.n$, let U be the type of the expression at $\beta l.1$ in $\mathbb{T}$, the clause follows

since:

$$\begin{aligned}
&\mathsf{reul}_{\mathbb{T}}(\beta.n)\\
&= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup \mathsf{retl}_{\mathbb{T}}(\beta.n) && \text{(definition of } \mathsf{reul}\text{)}\\
&= \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup M' \cup [\mathbb{T}(\beta l.1)] \cup U^l \cup \mathsf{retl}_{\mathbb{T}}(\beta l.1)\\
&\subseteq \mathsf{reuLf}_{\mathbb{T}'}(\beta.n) \cup [\mathbb{T}(\beta l.1)] \cup U^l \cup \mathsf{retl}_{\mathbb{T}}(\beta l.1)\\
&\qquad (\mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup M' \subseteq \mathsf{reuLf}_{\mathbb{T}}(\beta.n) + M' = \mathsf{reuLf}_{\mathbb{T}'}(\beta.n))\\
&= \mathsf{reul}_{\mathbb{T}'}(\beta.n)
\end{aligned}$$

- Case $\alpha.k = \beta.k$ and $k < n$, the proof is the same as in the case osNew.
- Case $\alpha \prec \beta$, as in the case osNew, we only need to prove for the cases $k = \mathsf{hi}(\mathbb{T}(\alpha))$ and $\mathsf{hi}(\mathbb{T}(\alpha')) = 1$ for all α' such that $\alpha \prec \alpha' \preccurlyeq \beta$.

  Since the two trees are only different at $\beta.n$ and $\beta r.1$, we only need to show that

  $$[\mathbb{T}(\beta.n)] \cup X^l \cup \mathsf{retl}_{\mathbb{T}}(\beta.n) \subseteq [\mathbb{T}'(\beta.n)] \cup X^l \cup \mathsf{retl}_{\mathbb{T}'}(\beta.n)$$

  let U be the type of the expression at $\beta l.1$ in $\mathbb{T}$. We have

  $$\begin{aligned}
  LHS &= [\mathbb{T}(\beta.n)] \cup X^l \cup M' \cup [\mathbb{T}(\beta l.1)] \cup U^l \cup \mathsf{retl}_{\mathbb{T}}(\beta l.1)\\
  &\subseteq [\mathbb{T}'(\beta.n)] \cup X^l \cup [\mathbb{T}(\beta l.1)] \cup U^l \cup \mathsf{retl}_{\mathbb{T}}(\beta l.1)\\
  &\qquad ([\mathbb{T}(\beta.n)] \cup M' \subseteq [\mathbb{T}(\beta.n)] + M' = [\mathbb{T}'(\beta.n)])\\
  &= RHS
  \end{aligned}$$

Case osParElim2: Analogous to the case osParElim1.

Case osCong:

$$\frac{(\mathsf{osCong}) \quad \mathbb{R} \equiv \mathbb{R}'}{\mathbb{T}[\![\mathbb{R}]\!]_\beta \longrightarrow \mathbb{T}[\![\mathbb{R}']\!]_\beta}$$

All the clauses hold by the hypothesis. □

Proof of Lemma 6.3.9 (Invariant of retop). *If $\Gamma \models_{\mathcal{R}} \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then for all positions $\alpha.k$ occurring in both $\mathbb{T}$ and $\mathbb{T}'$:*

$$[\mathbb{T}(\alpha.k)] + \mathsf{op}_{\mathbb{T}}(\alpha.k) + \mathsf{retop}_{\mathbb{T}}(\alpha.k) \supseteq [\mathbb{T}'(\alpha.k)] + \mathsf{op}_{\mathbb{T}'}(\alpha.k) + \mathsf{retop}_{\mathbb{T}'}(\alpha.k)$$

Proof. The proof proceeds by case analysis on the reduction relation $\longrightarrow$. In the Lemma 6.3.7, we have proved that all the expressions in $\mathbb{T}'$ are well-typed. In the followings, we assume that the expressions in $\mathbb{T}$ and $\mathbb{T}'$ have the types as in the proof of Lemma 6.3.7. We will consistently use location β for the node where the reduction occurs and let $n = \mathsf{hi}(\mathbb{T}(\beta))$.

- Case osNew:

  $$\frac{(\mathsf{osNew}) \quad x \prec A \in \mathit{Decls}}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{new}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\beta}$$

  First, we prove that $X^p = Z^p + x$ and $X^o(z) = Z^o(z)$ for all $z \neq x$ as follows. The first one holds since $X^p = X_1^p + X_2^p = (Y^p + x) + X_2^p = (Y^p + X_2^p) + x = Z^p + x$.

The second one holds since

$$\begin{aligned} X^o(z) &= ((X_1^o + X_2^p) \cup X_2^o!_{X_1^l})(z) \\ &= (((Y^o + x) + X_2^p) \cup X_2^o!_{(Y^l+x)})(z) \\ &= ((Y^o + X_2^p) \cup X_2^o!_{Y^l})(z) \\ &= Z^o(z) \end{aligned}$$

Now we prove the main clause for all $\alpha.k \in \mathbb{T}$.

Case $\alpha.k = \beta.n$, since β has no child nodes, $\mathsf{retop}_{\mathbb{T}}(\beta.n) = \mathsf{retop}_{\mathbb{T}'}(\beta.n) = []$ and the inequality that we have to prove becomes:

$$[\mathbb{T}(\beta.n)] + \mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq [\mathbb{T}'(\beta.n)] + \mathsf{op}_{\mathbb{T}'}(\beta.n)$$

We divide into two subcases. For $z \neq x$, we have

$$\begin{aligned} LHS(z) &= (M + (X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}))(z) && \text{(definition of op)} \\ &= (M + (Z^p \cup Z^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}))(z) && (X^p = Z^p + x,\ X^o(z) = Z^o(z)) \\ &= ((M + x) + (Z^p \cup Z^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}))(z) && (z \neq x) \\ &\geq ((M + x) + (Z^p \cup Z^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}))(z) && \text{(Lemma 6.3.8)} \\ &= ([\mathbb{T}'(\beta.n)] + \mathsf{op}_{\mathbb{T}'}(\beta.n))(z) \\ &= RHS(z) \end{aligned}$$

For $z = x$, we have

$$\begin{aligned} LHS(x) &= (M + (X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}))(x) && \text{(definition of op)} \\ &\geq (M + X^p)(x) && (X^p \subseteq X^o) \\ &= ((M + x) + Z^p)(x) && (X^p = Z^p + x) \\ &= ((M + x) + (Z^p \cup Z^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}))(x) && (x \in \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\ &= ([\mathbb{T}'(\beta.n)] + \mathsf{op}_{\mathbb{T}'}(\beta.n))(x) \\ &= RHS(x) \end{aligned}$$

Case $\alpha.k \neq \beta.n$, note that the two trees $\mathbb{T}$ and $\mathbb{T}'$ are different only at $\beta.n$ and the function retop calls recursively to its child nodes where the function reul is also involved. By Lemma 6.3.8 and the definition of the function op, we have

$$\mathsf{op}_{\mathbb{T}}(\alpha.k) \supseteq \mathsf{op}_{\mathbb{T}}(\alpha.k) \text{ for all } \alpha.k \neq \beta.n$$

Hence, starting from the leaves the clause holds for all $\alpha.k \in \mathsf{leaves}(\mathbb{T})$ (including the case $\alpha.k = \beta.n$ proved above). Then the clause holds for the parent nodes of the leaves, and so on. Up until the root, the clause is proved. We will use this argument for the subsequent cases.

- Case osReu1:

$$\begin{array}{ll} (\mathsf{osReu1}) & x \prec A \in \mathit{Decls} \quad x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.\mathsf{hi}(\mathbb{T}(\beta))) \\ & \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M,\, \mathtt{reu}\, x E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\beta \end{array}$$

Analogously to the case osNew, we only need to prove for $\alpha.k = \beta.n$ and $z = x$. We have $\mathsf{retop}_{\mathbb{T}}(\beta.n) = \mathsf{retop}_{\mathbb{T}'}(\beta.n) = []$, thus the inequality that we have to prove becomes:

$$[\mathbb{T}(\beta.n)] + \mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq [\mathbb{T}'(\beta.n)] + \mathsf{op}_{\mathbb{T}'}(\beta.n)$$

We have:

$$\begin{aligned}
LHS(x) &= (X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)})(x) && (x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \supseteq [\mathbb{T}(\beta.n)]) \\
&\geq X^p(x) && (X^p \subseteq X^o) \\
&= (X_1^p + X_2^p)(x) && (X^p = X_1^p + X_2^p) \\
&= ((Y^p + x) + X_2^p)(x) && (X_1^p = Y^p + x) \\
&= (Z^p + x)(x) && (Z^p = Y^p + X_2^p) \\
&= ((M + x) + Z^p)(x) && (x \notin M) \\
&= ((M + x) + (Z^p \cup Z^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}))(x) && (x \in \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\
&= RHS(x)
\end{aligned}$$

For $\alpha.k \neq \beta.n$, we use the same argument as in the case osNew.

- Case osReu2:

$$\frac{(\mathsf{osReu2}) \quad x \prec A \in \mathit{Decls} \quad x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.\mathsf{hi}(\mathbb{T}(\beta)))}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{reu}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\beta}$$

Analogously to the case osNew, we only need to prove for $\alpha.k = \beta.n$ and $z = x$. Since $\mathsf{retop}_{\mathbb{T}}(\beta.n) = \mathsf{retop}_{\mathbb{T}'}(\beta.n) = []$ and $[\mathbb{T}(\beta.n)] = [\mathbb{T}'(\beta.n)]$, the inequality that we have to prove becomes:

$$\mathsf{op}_{\mathbb{T}}(\beta.n)(x) \geq \mathsf{op}_{\mathbb{T}}(\beta.n)(x)$$

We have:

$$\begin{aligned}
LHS &= (X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)})(x) \\
&= X^p(x) && (x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \subseteq \mathsf{reul}_{\mathbb{T}}(\beta.n)) \\
&= (X_1^p + X_2^p)(x) \\
&= (Y^p + X_2^p)(x) && (Y^p = X_1^p) \\
&= (Z^p \cup Z^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)})(x) && (x \in \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\
&= RHS
\end{aligned}$$

For $\alpha.k \neq \beta.n$, we use the same argument as in the case osNew.

- Case osChoice:

$$\frac{(\mathsf{osChoice})}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, (A + B)E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\beta}$$

For $\alpha.k = \beta.n$, as in the case osNew, we only need to show that

$$[\mathbb{T}(\beta.n)] + \mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq [\mathbb{T}'(\beta.n)] + \mathsf{op}_{\mathbb{T}'}(\beta.n)$$

Since $[\mathbb{T}(\beta.1)] = [\mathbb{T}'(\beta.1)]$, we only need to prove that $\mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq \mathsf{op}_{\mathbb{T}'}(\beta.n)$. We

have:

$$\begin{aligned}
LHS &= X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&= (X_1^p + X_2^p) \cup ((X_1^o + X_2^p) \cup X_2^o!_{X_1^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&\supseteq (Y^p + X_2^p) \cup ((Y^o + X_2^p) \cup X_2^o!_{Y^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
& \qquad\qquad (X_1^p \supseteq Y^p,\ X_1^o \supseteq Y^o,\ X_1^l \subseteq Y^l) \\
&= Z^p \cup Z^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)} \qquad (\mathsf{reul}_{\mathbb{T}}(\beta.n) = \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\
&= RHS
\end{aligned}$$

For $\alpha.k \neq \beta.n$, we use the same argument as in the case osNew.

- Case osPush:

$$\text{(osPush)} \quad \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \{A\}E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ ([], A))]\!]_\beta$$

Since $\beta.(n+1) \notin \mathbb{T}$, we do not need to prove for this case. For $\alpha.k = \beta.n$, we have $\mathsf{retop}_{\mathbb{T}}(\beta.n) = []$ since $\beta l, \beta r \notin \mathbb{T}'$ and $\mathsf{retop}_{\mathbb{T}'}(\beta.n) = []$ since $n - 1 < \mathsf{hi}(\mathbb{T}(\beta))$. Thus the inequality that we need to prove becomes:

$$[\mathbb{T}(\beta.n)] + \mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq [\mathbb{T}'(\beta.n)] + \mathsf{op}_{\mathbb{T}'}(\beta.n)$$

In addition, $[\mathbb{T}(\beta.n)] = [\mathbb{T}'(\beta.n)] = M$, so we only need to prove that $\mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq \mathsf{op}_{\mathbb{T}'}(\beta.n)$. We have

$$\begin{aligned}
LHS &= X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&= (X_1^p + X_2^p) \cup ((X_1^o + X_2^p) \cup X_2^o!_{X_1^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&= X_2^p \cup (X_2^p \cup X_2^o)!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \qquad (X_1^l = X_1^p = X_1^o = []) \\
&= X_2^p \cup X_2^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \qquad (X_2^p \subseteq X_2^o) \\
&= X_2^p \cup X_2^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)} \qquad (\mathsf{reul}_{\mathbb{T}}(\beta.n) = \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\
&= RHS
\end{aligned}$$

For $\alpha.k \neq \beta.n$, we use the same argument as in the case osNew.

- Case osPop:

$$\text{(osPop)} \quad \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ (M', \epsilon))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E))]\!]_\beta$$

By Lemma 6.3.7, we have E is well-typed. Assume that X is the type of E.

Since $\beta.n \notin \mathbb{T}'$, we do not need to prove for this position. Case $\alpha.k = \beta.(n-1)$, we have $\mathsf{retop}_{\mathbb{T}}(\beta.(n-1)) = []$ since $n - 1 < \mathsf{hi}(\mathbb{T}(\beta))$, and $\mathsf{retop}_{\mathbb{T}'}(\beta.n) = []$ since $\beta l, \beta r \notin \mathbb{T}'$. Thus the inequality that we need to prove becomes:

$$[\mathbb{T}(\beta.(n-1))] + \mathsf{op}_{\mathbb{T}}(\beta.(n-1)) \supseteq [\mathbb{T}'(\beta.(n-1))] + \mathsf{op}_{\mathbb{T}'}(\beta.(n-1))$$

In addition, $[\mathbb{T}(\beta.(n-1))] = [\mathbb{T}'(\beta.(n-1))] = M$, so we only need to prove that $\mathsf{op}_{\mathbb{T}}(\beta.(n-1)) \supseteq \mathsf{op}_{\mathbb{T}'}(\beta.(n-1))$. By definition of function op the inequality becomes

$$X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.(n-1))} \supseteq X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.(n-1))}$$

This inequality follows by Lemma 6.3.8.

For $\alpha.k \neq \beta.n$, we use the same argument as in the case osNew.

- Case osParIntr:

$$\text{(osParIntr)}$$
$$\mathbb{T}[\![\mathsf{Lf}(\mathbf{S}\circ(M,(A \parallel B)E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S}\circ(M,E),\mathsf{Lf}(([],A)),\mathsf{Lf}(([],B)))]\!]_\beta$$

Since $\beta l.1$ and $\beta r.1$ are not in $\mathbb{T}$, we do not need to prove for these cases. For $\alpha.k = \beta.n$, we have $\mathsf{retop}_{\mathbb{T}}(\beta.n) = []$ since $\beta l, \beta r \notin \mathbb{T}'$. Thus the inequality we need to prove becomes:

$$[\mathbb{T}(\beta.n)] + \mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq [\mathbb{T}'(\beta.n)] + \mathsf{op}_{\mathbb{T}'}(\beta.n) + \mathsf{retop}_{\mathbb{T}'}(\beta.n)$$

In addition, since $[\mathbb{T}(\beta.n)] = [\mathbb{T}'(\beta.n)] = M$, we only need to prove that:

$$\mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq \mathsf{op}_{\mathbb{T}'}(\beta.n) + \mathsf{retop}_{\mathbb{T}'}(\beta.n)$$

We expand $\mathsf{retop}_{\mathbb{T}'}(\beta.n)$ as follows:

$$\begin{aligned}
\mathsf{retop}_{\mathbb{T}'}(\beta.n) &= \mathsf{op}_{\mathbb{T}'}(\beta l.1) + \mathsf{op}_{\mathbb{T}'}(\beta r.1) && ([\mathbb{T}(\beta l.1)] = [\mathbb{T}(\beta r.1)] = [])\\
&= (Y_1^p \cup Y_1^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta l.1)}) + (Y_2^p \cup Y_2^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta l.1)})\\
&= (Y_1^p \cup Y_1^o!_{\mathsf{reuLf}_{\mathbb{T}'}(\beta.n)}) + (Y_2^p \cup Y_2^o!_{\mathsf{reuLf}_{\mathbb{T}'}(\beta.n)})\\
& && (\mathsf{reuLf}_{\mathbb{T}'}(\beta.n) = \mathsf{reul}_{\mathbb{T}'}(\beta l.n) = \mathsf{reul}_{\mathbb{T}'}(\beta r.n))\\
&= X_1^p \cup X_1^o!_{\mathsf{reuLf}_{\mathbb{T}'}(\beta.n)}
\end{aligned}$$

So the inequality we need to prove becomes:

$$\mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq \mathsf{op}_{\mathbb{T}'}(\beta.n) + (X_1^p \cup X_1^o!_{\mathsf{reuLf}_{\mathbb{T}'}(\beta.n)})$$

We divide into two cases. If $x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.n) = \mathsf{reuLf}_{\mathbb{T}'}(\beta.n)$, then $x \in \mathsf{reul}_{\mathbb{T}'}(\beta.n)$. So we have:

$$\begin{aligned}
LHS(x) &= (X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)})(x)\\
&= X^p(x) && (x \in \mathsf{reul}_{\mathbb{T}}(\beta.n))\\
&= (X_1^p + X_2^p)(x)\\
&= ((X_1^p \cup X_1^o!_{\mathsf{reuLf}_{\mathbb{T}'}(\beta.n)}) + X_2^p)(x) && (x \in \mathsf{reuLf}_{\mathbb{T}'}(\beta.n))\\
&= ((X_1^p \cup X_1^o!_{\mathsf{reuLf}_{\mathbb{T}'}(\beta.n)}) + (X_2^p \cup X_2^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}))(x) && (x \in \mathsf{reul}_{\mathbb{T}'}(\beta.n))\\
&= RHS(x) && (\text{definition of } \mathsf{op}_{\mathbb{T}'}(\beta.n))
\end{aligned}$$

Otherwise, $x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.n)$, then $x \notin \mathsf{reul}_{\mathbb{T}}(\beta.n)$ since $\mathsf{reuLf}_{\mathbb{T}}(\beta.n) = \mathsf{reul}_{\mathbb{T}}(\beta.n)$ and we have

$$\begin{aligned}
LHS(x) &= (X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)})(x) && (\text{definition of } \mathsf{op})\\
&= X^o(x) && (x \notin \mathsf{reul}_{\mathbb{T}}(\beta.n),\ X^p \subseteq X^o)\\
&= ((X_1^o + X_2^p) \cup X_2^o!_{X_1^l})(x)
\end{aligned}$$

and

$$\begin{aligned}
RHS(x) &= ((X_2^p \cup X_2^o!_{(\mathsf{reuLf}_{\mathbb{T}}(\beta.n) \cup X_1^l)}) + (X_1^p \cup X_1^o!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)}))(x)\\
&= ((X_2^p \cup X_2^o!_{X_1^l}) + (X_1^p \cup X_1^o))(x) && (x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.n))\\
&= ((X_2^p \cup X_2^o!_{X_1^l}) + X_1^o)(x) && (X_1^p \subseteq X_1^o)
\end{aligned}$$

Now if $x \in X_1^l$, then $LHS(x) = (X_1^o + X_2^p)(x) = RHS(x)$. Otherwise, $x \notin X_1^l$, then $x \notin X_1^o$ by Lemma 6.3.10 and we have $LHS(x) = (X_2^p \cup X_2^o)(x) = RHS(x)$. The

proof for the case $\alpha.k = \beta.n$ is completed here.

For $\alpha.k \neq \beta.n$, we use the same argument as in the previous case osNew.

- Case osParElim1:

$$\text{(osParElim1)}\quad \mathbb{T}[\![\mathsf{Nd}(\mathbf{S}\circ(M,E),\mathbb{R},\mathsf{Lf}((M',\epsilon)))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S}\circ(M+M',E),\mathbb{R})]\!]_\beta$$

Assume that the position of the node $\mathsf{Lf}(M',\epsilon)$ is βr (the proof is analogous if it is βl). Since $\beta r \notin \mathbb{T}$, we do not need to prove for this case.

For $\alpha.k \succ \beta.n$, the inequality holds by Lemma 6.3.8.

For $\alpha.k = \beta.n$, the inequality we need to prove becomes:

$$\begin{aligned}&[\mathbb{T}(\beta.n)] + \mathsf{op}_{\mathbb{T}}(\beta.n) + M'\\ &+ [\mathbb{T}(\beta l.1)] + \mathsf{op}_{\mathbb{T}}(\beta l.1) + \mathsf{retop}_{\mathbb{T}}(\beta l.1)\\ &\supseteq\\ &[\mathbb{T}'(\beta.n)] + \mathsf{op}_{\mathbb{T}'}(\beta.n)\\ &+ [\mathbb{T}'(\beta l.1)] + \mathsf{op}_{\mathbb{T}'}(\beta l.1) + \mathsf{retop}_{\mathbb{T}'}(\beta l.1)\end{aligned}$$

Since we have just proved (in case $\alpha.k \succ \beta.n$) that

$$[\mathbb{T}(\beta l.1)] + \mathsf{op}_{\mathbb{T}}(\beta l.1) + \mathsf{retop}_{\mathbb{T}}(\beta l.1) \supseteq [\mathbb{T}'(\beta l.1)] + \mathsf{op}_{\mathbb{T}'}(\beta l.1) + \mathsf{retop}_{\mathbb{T}'}(\beta l.1)$$

and $[\mathbb{T}'(\beta.n)] = [\mathbb{T}(\beta.n)] + M$, the clause is simplified to $\mathsf{op}_{\mathbb{T}}(\beta.n) \supseteq \mathsf{op}_{\mathbb{T}'}(\beta.n)$, which can be expanded to:

$$X^p \cup X^o!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)} \subseteq X^p \cup X^o!_{\mathsf{reuLf}_{\mathbb{T}'}(\beta.n)}$$

where X is the type of E. Now the inequality follows by Lemma 6.3.8.

The remaining cases are proved using the same argument as in the previous case osNew.

- Case osParElim2:

$$\text{(osParElim2)}\quad \mathbb{T}[\![\mathsf{Nd}(\mathbf{S}\circ(M,E),\mathsf{Lf}((M',\epsilon)))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S}\circ(M+M',E))]\!]_\beta$$

Analogous to the previous case osParElim1.

- Case osCong:

$$\frac{\text{(osCong)}\quad \mathbb{R}\equiv\mathbb{R}'}{\mathbb{T}[\![\mathbb{R}]\!]_\beta \longrightarrow \mathbb{T}[\![\mathbb{R}']\!]_\beta}$$

The clause holds by the hypothesis.

□

Proof of Lemma 6.3.4 (Preservation). *If* $\Gamma \models_{\mathcal{R}} \mathbb{T}$ *and* $\mathbb{T} \longrightarrow \mathbb{T}'$, *then* $\Gamma \models_{\mathcal{R}} \mathbb{T}'$.

Proof. The proof proceeds by case analysis on the reduction relation: $\longrightarrow$. The first clause of Definition 6.3.1 has been proved in Lemma 6.3.7, so we only have to prove the second clause that $\mathsf{maxins}(\mathbb{T}'|_{\mathcal{L}}) \subseteq \mathcal{R}$ for all valid sets $\mathcal{L}$ of $\mathbb{T}'$ where

$$\mathsf{maxins}(\mathbb{T}'|_{\mathcal{L}}) = \biguplus_{\alpha.k\prec\mathcal{L}'} [\mathbb{T}'(\alpha.k)] + \biguplus_{\alpha.k\in\mathcal{L}'} \mathsf{ij}_{\mathbb{T}'}(\alpha.k) \subseteq \mathcal{R}$$

where $\mathcal{L}'$ is the set of all positions at the top of leaves of subtree $\mathbb{T}'|_{\mathcal{L}}$.

We will consistently use location β for the node where the reduction occurs and let $n = \mathsf{hi}(\mathbb{T}(\beta))$.

- Case osNew:

$$\begin{array}{l}(\mathsf{osNew}) \quad x \prec A \in \mathit{Decls} \\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{new}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\beta\end{array}$$

Recall that we have proved in Lemma 6.3.7 that $\Gamma \vdash \mathtt{new}\, xE : X$, $\Gamma \vdash A : Y$ and $\Gamma \vdash E : Z$. Since the two trees $\mathbb{T}$ and $\mathbb{T}'$ have the same tree structure ($\alpha.k \in \mathbb{T}$ implies $\alpha.k \in \mathbb{T}'$ and vice versa), so $\mathcal{L}$ is also a valid set of $\mathbb{T}$.

If $\beta.n \in \mathbb{T}'|_{\mathcal{L}}$, then since the two trees $\mathbb{T}$ and $\mathbb{T}'$ are only different at $\beta.n$ we have:

$$\mathsf{maxins}(\mathbb{T}'|_{\mathcal{L}}) = \mathsf{maxins}(\mathbb{T}|_{\mathcal{L}}) - \mathsf{ij}_{\mathbb{T}}(\beta.n) + \mathsf{ij}_{\mathbb{T}'}(\beta.n)$$

Since $\mathsf{maxins}(\mathbb{T}|_{\mathcal{L}}) \subseteq \mathcal{R}$ by the hypothesis, the clause holds if we can prove that $\mathsf{ij}_{\mathbb{T}}(\beta.n) \supseteq \mathsf{ij}_{\mathbb{T}'}(\beta.n)$.

First, we have $[\mathbb{T}'(\beta.n)] = [\mathbb{T}(\beta.n)] + x$. Second, since β is a leaf of both $\mathbb{T}$ and $\mathbb{T}'$, $\mathsf{retop}_{\mathbb{T}}(\beta.n) = \mathsf{retop}_{\mathbb{T}'}(\beta.n) = []$. So by the definition of the function ij, we only need to show that

$$X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \supseteq (Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}) + x$$

We divide into two cases. If $z \neq x$, then we have $X_1^*(z) = Y^*(z)$ and

$$\begin{aligned} LHS(z) &= (X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}))(z) \\ &= ((X_1^j \cup (X_1^p + X_2^j)) \cup (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}))(z) \\ &= ((Y^j \cup (Y^p + X_2^j)) \cup (Y^i \cup (Y^o + X_2^j) \cup X_2^i!_{Y^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}))(z) \\ &= (x + (Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}))(z) \qquad (\mathsf{reul}_{\mathbb{T}}(\beta.n)(z) = \mathsf{reul}_{\mathbb{T}'}(\beta.n)(z)) \\ &= RHS(z) \end{aligned}$$

Otherwise, $z = x$, we have

$$\begin{aligned} LHS(x) &= (X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}))(x) \\ &\geq X^j(x) \\ &= (X_1^j \cup (X_1^p + X_2^j))(x) \\ &= (x + (Y^j \cup (Y^p + X_2^j)))(x) \qquad (X_1^* = Y^* + x) \\ &= (x + Z^j)(x) \\ &= (x + (Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}))(x) \qquad (x \in \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\ &= RHS(x) \end{aligned}$$

Otherwise, $\beta.n \notin \mathbb{T}'|_{\mathcal{L}}$, then the two subtrees $\mathbb{T}|_{\mathcal{L}}$ and $\mathbb{T}'|_{\mathcal{L}}$ are the same and for all $\alpha.k \in \mathsf{leaves}(\mathbb{T}|_{\mathcal{L}})$ and $\alpha.k \in \mathsf{leaves}(\mathbb{T}'|_{\mathcal{L}})$, we have $\mathsf{ij}_{\mathbb{T}}(\alpha.k) \supseteq \mathsf{ij}_{\mathbb{T}'}(\alpha.k)$ by Lemma 6.3.8 and Lemma 6.3.9. Hence the clause follows.

- Case osReu1:

$$\begin{array}{l}(\mathsf{osReu1}) \quad x \prec A \in \mathit{Decls} \quad x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.\mathsf{hi}(\mathbb{T}(\beta))) \\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{reu}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + x, AE))]\!]_\beta\end{array}$$

Analogous to the case osNew, if $\beta.n \in \mathbb{T}'|_{\mathcal{L}}$ we only need to prove that:

$$(X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)})(x) \geq (x + (Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}))(x)$$

We have

$$\begin{aligned}
LHS &= (X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)})(x) && (X^j \subseteq X^i)\\
&= X^i(x) && (x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.n) = \mathsf{reul}_{\mathbb{T}}(\beta.n))\\
&= (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l})(x) && \\
&= (X_1^i \cup (X_1^o + X_2^j))(x) && (x \in X_1^l)\\
&= (Y^i + x) \cup ((Y^o + x) + X_2^j))(x) && (X_1^i = Y^i + x,\ X_1^o = Y^o + x)\\
&\geq ((Y^j + x) \cup ((Y^p + x) + X_2^j))(x) && (Y^i \supseteq Y^j,\ Y^o \supseteq Y^p)\\
&= (x + (Y^j \cup (Y^p + X_2^j)))(x) && \\
&= (x + Z^j)(x) && \\
&= (x + (Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}))(x) && (x \in \mathsf{reul}_{\mathbb{T}'}(\beta.n))\\
&= RHS
\end{aligned}$$

Case $\beta.n \notin \mathbb{T}'|_{\mathcal{L}}$ is proved as in the analogous subcase of the case osNew.

- Case osReu2:

$$\frac{(\mathsf{osReu2}) \quad x \prec A \in \mathit{Decls} \quad x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.\mathsf{hi}(\mathbb{T}(\beta)))}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \mathtt{reu}\, xE))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\beta}$$

Analogous to the case osNew, if $\beta.n \in \mathbb{T}'|_{\mathcal{L}}$ we only need to prove that:

$$(X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)})(x) \geq (Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)})(x)$$

We have

$$\begin{aligned}
LHS &= (X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)})(x) && \\
&= X^j(x) && (x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.n))\\
&= (X_1^j \cup (X_1^p + X_2^j))(x) && \\
&= (Y^j \cup (Y^p + X_2^j))(x) && (X_1^j = Y^j,\ X_1^p = Y^p)\\
&= Z^j(x) && \\
&= (Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)})(x) && (x \in \mathsf{reuLf}_{\mathbb{T}'}(\beta.n))\\
&= RHS
\end{aligned}$$

Case $\beta.n \notin \mathbb{T}'|_{\mathcal{L}}$ is proved as in the analogous subcase of the case osNew.

- Case osChoice:

$$\frac{(\mathsf{osChoice})}{\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, (A+B)E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, AE))]\!]_\beta}$$

Analogous to the case osNew, if $\beta.n \in \mathbb{T}'|_{\mathcal{L}}$, we only need to prove that:

$$X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \supseteq Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}$$

We have

$$
\begin{aligned}
LHS &= X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&= X_1^j \cup (X_1^p + X_2^j) \cup (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&\supseteq Y^j \cup (Y^p + X_2^j) \cup (Y^i \cup (Y^o + X_2^j) \cup X_2^i!_{Y^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&\qquad\qquad (X_1^i \supseteq Y^i, X_1^o \supseteq Y^o, X_1^p \supseteq Y^p,\ X_1^o \supseteq Y^o,\ X_1^l \subseteq Y^l) \\
&= Z^j \cup Z^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)} \qquad (\mathsf{reul}_{\mathbb{T}}(\beta.n) = \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\
&= RHS
\end{aligned}
$$

Case $\beta.n \notin \mathbb{T}'|_{\mathcal{L}}$ is proved as in the analogous subcase of the case osNew.

- Case osPush:

$$
\begin{aligned}
&(\mathsf{osPush}) \\
&\mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, \{A\}E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M, E) \circ ([], A))]\!]_\beta
\end{aligned}
$$

Recall that we have proved in Lemma 6.3.7 that $\Gamma \vdash \{A\} : X_1$, $\Gamma \vdash A : Y$ and $\Gamma \vdash E : X_2$.

If $\beta.(n+1) \in \mathbb{T}'|_{\mathcal{L}}$, we choose $\mathcal{L}_0 = \mathcal{L} \setminus \{\beta.(n+1)\} \cup \{\beta.n\}$. Then consider the subtree $\mathbb{T}|_{\mathcal{L}_0}$, as in the case osNew, we only need to prove that:

$$X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \supseteq X_1^j \cup X_1^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.(n+1))}$$

We have

$$
\begin{aligned}
LHS &= X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&= X_1^j \cup (X_1^p + X_2^j) \cup (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&\supseteq X_1^j \cup X_1^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&= X_1^j \cup X_1^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)} \qquad (\mathsf{reul}_{\mathbb{T}}(\beta.n) = \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\
&= X_1^j \cup X_1^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.(n+1))} \qquad (\mathsf{reul}_{\mathbb{T}'}(\beta.n) = \mathsf{reul}_{\mathbb{T}'}(\beta.(n+1))) \\
&= RHS
\end{aligned}
$$

If $\beta.n \in \mathbb{T}'|_{\mathcal{L}}$, then we consider the subtree $\mathbb{T}|_{\mathcal{L}}$. Note that $\beta.n$ is not at the top of the stack in $\mathbb{T}'$ and $\beta.n$ has no child nodes in $\mathbb{T}$ so we have $\mathsf{retop}_{\mathbb{T}}(\beta.n) = \mathsf{retop}_{\mathbb{T}'}(\beta.n) = []$, and as in the case osNew, we only need to prove that:

$$X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \supseteq X_2^j \cup X_2^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}$$

We have

$$
\begin{aligned}
LHS &= X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&= X_1^j \cup (X_1^p + X_2^j) \cup (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \\
&\supseteq X_2^j \cup X_2^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \qquad (X_1^l = []) \\
&= X_2^j \cup X_2^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.(n+1))} \qquad (\mathsf{reul}_{\mathbb{T}}(\beta.n) = \mathsf{reul}_{\mathbb{T}'}(\beta.n)) \\
&= RHS
\end{aligned}
$$

Case $\beta.n \notin \mathbb{T}'|_{\mathcal{L}}$ is proved as in the analogous subcase of the case osNew.

- Case osPop:

$$\text{(osPop)}\\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S}\circ(M,E)\circ(M',\epsilon))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S}\circ(M,E))]\!]_\beta$$

The clause follows easily by the hypothesis.

- Case osParIntr:

$$\text{(osParIntr)}\\ \mathbb{T}[\![\mathsf{Lf}(\mathbf{S}\circ(M,(A\parallel B)E))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S}\circ(M,E),\mathsf{Lf}(([],A)),\mathsf{Lf}(([],B)))]\!]_\beta$$

Assume that βl is the location of $\mathsf{Lf}(([],A))$ and βr is the location of $\mathsf{Lf}(([],B))$ (the proof is analogous if otherwise).

We divide into the following typical cases.

If $\beta l.1 \in \mathbb{T}'|_{\mathcal{L}}$ or $\beta r.1 \in \mathbb{T}'|_{\mathcal{L}}$ or both $\beta l.1, \beta r.1 \in \mathbb{T}'|_{\mathcal{L}}$, then we choose $\mathcal{L}_0 = \mathcal{L}\setminus\{\beta l.1,\beta r.1\}\cup\{\beta.n\}$ and consider subtree $\mathbb{T}|_{\mathcal{L}_0}$ of $\mathbb{T}$. As in the case osNew, we only need to prove that:

$$X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \supseteq \mathsf{ij}_{\mathbb{T}'}(\beta l.1) + \mathsf{ij}_{\mathbb{T}'}(\beta r.1)$$

We have:

$$\begin{aligned}
&= X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}\\
&= X_1^j \cup (X_1^p + X_2^j) \cup (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l})!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}\\
&\supseteq X_1^j \cup X_1^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)} && (X_1^i = Y_1^i + Y_2^i,\ X_1^j = Y_1^j + Y_2^j)\\
&= (Y_1^j + Y_2^j) \cup (Y_1^i + Y_2^i)!_{\mathsf{reul}_{\mathbb{T}'}(\beta.n)}\\
&= (Y_1^j \cup Y_1^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta l.1)}) + (Y_2^j \cup Y_2^i!_{\mathsf{reul}_{\mathbb{T}'}(\beta r.1)})\\
& && (\mathsf{reuLf}_{\mathbb{T}}(\beta.n) = \mathsf{reul}_{\mathbb{T}'}(\beta l.1) = \mathsf{reul}_{\mathbb{T}}(\beta r.1))\\
&= \mathsf{ij}_{\mathbb{T}'}(\beta l.1) + \mathsf{ij}_{\mathbb{T}'}(\beta r.1) && ([\mathbb{T}'(\beta l.1)] = [\mathbb{T}'(\beta r.1)] = [])
\end{aligned}$$

If $\beta.n \in \mathbb{T}'|_{\mathcal{L}}$, then we consider subtree $\mathbb{T}|_{\mathcal{L}}$, as in the case osNew, we only need to prove that:

$$X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \supseteq (\mathsf{retop}_{\mathbb{T}}(\beta.n) + X_2^j) \cup X_2^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)}$$

Since $\mathsf{reuLf}_{\mathbb{T}}(\beta.n) = \mathsf{reuLf}_{\mathbb{T}'}(\beta.n)$, we have

$$\begin{aligned}
&\mathsf{retop}_{\mathbb{T}}(\beta.n)\\
&= (Y_1^p \cup Y_1^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta l.n)}) + (Y_2^p \cup Y_2^o!_{\mathsf{reul}_{\mathbb{T}'}(\beta l.n)}) && ([\mathbb{T}'(\beta l.1)] = [\mathbb{T}'(\beta r.1)] = [])\\
&= (Y_1^p \cup Y_1^o!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)}) + (Y_2^p \cup Y_2^o!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)})\\
& && (\mathsf{reul}_{\mathbb{T}'}(\beta l.1) = \mathsf{reul}_{\mathbb{T}'}(\beta r.1) = \mathsf{reuLf}_{\mathbb{T}'}(\beta.n) = \mathsf{reuLf}_{\mathbb{T}}(\beta.n))\\
&= (Y_1^p + Y_2^p) \cup (Y_1^o + Y_2^o)!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)}\\
&= X_1^p \cup X_1^o!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)} && (X_1^o = Y_1^o + Y_2^o,\ X_1^p = Y_1^p + Y_2^p)
\end{aligned}$$

In addition, $\mathsf{reul}_{\mathbb{T}'}(\beta.n) = \mathsf{reuLf}_{\mathbb{T}}(\beta.n) + X_1^l$, so the inequality we have to prove becomes

$$X^j \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\beta.n)} \supseteq ((X_1^p \cup X_1^o!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)}) + X_2^j) \cup X_2^i!_{(\mathsf{reuLf}_{\mathbb{T}}(\beta.n)+X_1^l)}$$

If $x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.n) \subseteq \mathsf{reul}_{\mathbb{T}}(\beta.n)$, then we have

$$\begin{aligned}
RHS(x) &= (X_1^p + X_2^j)(x) \\
&\subseteq (X_1^j \cup (X_1^p + X_2^j))(x) \\
&= X^j(x) \\
&= (X^j \cup X^i!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)})(x) && (x \in \mathsf{reuLf}_{\mathbb{T}}(\beta.n)) \\
&= LHS(x)
\end{aligned}$$

Otherwise, $x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.n)$, then

$$\begin{aligned}
RHS(x) &= ((X_1^o + X_2^j) \cup X_2^i!_{X_1^l})(x) && (X_1^p \subseteq X_1^o) \\
&\subseteq (X_1^i \cup (X_1^o + X_2^j) \cup X_2^i!_{X_1^l})(x) \\
&= X^i(x) \\
&= (X^i!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)})(x) && (x \notin \mathsf{reuLf}_{\mathbb{T}}(\beta.n)) \\
&= (X^j \cup X^i!_{\mathsf{reuLf}_{\mathbb{T}}(\beta.n)})(x) && (X^j \subseteq X^j) \\
&= LHS(x)
\end{aligned}$$

Case $\beta.n \notin \mathbb{T}'|_{\mathcal{L}}$ is proved as in the analogous subcase of the case osNew.

- Case osParElim1:

$$\begin{array}{l}
(\mathsf{osParElim1}) \\
\mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathbb{R}, \mathsf{Lf}((M', \epsilon)))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M + M', E), \mathbb{R})]\!]_\beta
\end{array}$$

  Assume that the position of the node $\mathsf{Lf}(M', \epsilon)$ is βr (the proof is analogous if it is βl). Analogous to the case osNew, we only need to prove for cases where $\beta.n \in \mathbb{T}'|_{\mathcal{L}}$. We we choose $\mathcal{L}_0 = \mathcal{L} \cup \{\beta r.1\}$, then the clause holds easily since $\mathsf{maxins}(\mathbb{T}'|_{\mathcal{L}}) = \mathsf{maxins}(\mathbb{T}|_{\mathcal{L}_0})$ and $\mathsf{ij}_{\mathbb{T}}(\beta.n) \supseteq \mathsf{ij}_{\mathbb{T}'}(\beta.n)$ by Lemma 6.3.8 and Lemma 6.3.9.

- Case osParElim2:

$$\begin{array}{l}
(\mathsf{osParElim2}) \\
\mathbb{T}[\![\mathsf{Nd}(\mathbf{S} \circ (M, E), \mathsf{Lf}((M', \epsilon)))]\!]_\beta \longrightarrow \mathbb{T}[\![\mathsf{Lf}(\mathbf{S} \circ (M + M', E))]\!]_\beta
\end{array}$$

  This case is a special case of the case osParElim1.

- Case osCong:

$$\frac{(\mathsf{osCong}) \quad \mathbb{R} \equiv \mathbb{R}'}{\mathbb{T}[\![\mathbb{R}]\!]_\beta \longrightarrow \mathbb{T}[\![\mathbb{R}']\!]_\beta}$$

  All the clauses in Definition 6.3.1 hold by the hypothesis.

□

Proof of Lemma 6.3.5 (Progress). *If $\Gamma \models_{\mathcal{R}} \mathbb{T}$, then either $\mathbb{T}$ is terminal or there exists configuration $\mathbb{T}'$ such that $\mathbb{T} \longrightarrow \mathbb{T}'$.*

Proof. If $\mathbb{T}$ is terminal, then the lemma is trivial. Otherwise, we assume that the next leaf to be executed is α with $\mathbb{T}(\alpha) = \mathbf{S} \circ (M, E)$ and E well-typed. There are two possibilities:

- If $E = \epsilon$ then we can apply the rule osPop if $\mathsf{hi}(\mathbb{T}(\alpha)) > 1$, and osParElim1 or osParElim2 or osCong if $\mathsf{hi}(\mathbb{T}(\alpha)) = 1$ since these rules do not require any preconditions.

- If $E \neq \epsilon$ we can break down E into E_1 and E_2 such that E_1 is one of the forms $\texttt{new}\, x$, $\texttt{reu}\, x$, $\{A\}$, $(A+B)$ and $(A \parallel B)$. If E_1 is of the forms $\{A\}$, $(A+B)$ and $(A \parallel B)$, we can apply the rules osPush, osChoice and osParIntro, respectively, since these rules do not require any preconditions. Otherwise, we can apply one of the rules osNew, osReu1 and osReu2 but we have to verify that $x \prec A \in \Gamma$. This is immediate by Generation Lemma 6.3.12, clauses 1 and 2 that there exists one $x \prec A \in \Gamma$.

□

Proof of Theorem 6.3.6 (Soundness). *If program Prog = Decls; E is well-typed with respect to a requirement $\mathcal{R}$, then for any $\mathbb{T}$ such that $\mathsf{Lf}([], E) \longrightarrow^* \mathbb{T}$ we have $\mathbb{T}$ is not stuck and $[\mathbb{T}] \subseteq \mathcal{R}$.*

Proof. The same as the proof of Theorem 5.3.6. □

Termination. As in Chapter 2, all well-typed programs terminate after a finite number of reduction steps. We can prove this property by the same method of Chapters 2 and 4, with the function mts defined for $\texttt{reu}\, x$ as in Section 5.3.3 and for the parallel composition and trees as in Section 4.3.3.

Chapter 7

Conclusions and Future Research

7.1 Conclusions

Through five technical chapters 2 to 6, we have seen that type systems can be used to control resource bound of programs. The main results are in Chapters 4 and 6; the other three chapters pave the way to the two main chapters and are special cases of the two.

In Chapter 4, the type system can find the upper bound of resources for a class of programs of a component language, whose main features are two primitives, one for instantiation and one for deallocation, and four operators for compositions: sequencing, choice, scope and parallel composition. The sequential composition plays an important role since it makes the language imperative instead of functional and thus captures aspects that are more popular in practice. We believe that the model of this chapter can be used as a base for further extensions to procedural and object-oriented programming languages in practice.

In Chapter 6, the type system can statically verify if a program satisfies a given resource constraint at runtime. The language features are four operators for composition as in Chapter 4 and two instantiation primitives: `new` and `reu`. The operational semantics of `reu` in Chapters 5 and 6 is only one among various possible alternatives. For example, `reu` x could skip executing the body of the definition of x if there exists a reusable instance of x—like making a reference to an existing object. This behaviour seems simpler but further study is needed to find out if we can build an adequate type system.

In spite of the high level of abstraction, these type systems can be directly used to detect certain classes of errors, since the languages have the core features which appear in many imperative programming languages in which resources are controlled by programmers.

7.2 Future Research

The sharpness of the resource bounds in all chapters (2 to 6) has not been proved and we leave it for future work. Further extensions of the languages such as adding functions with parameters, allowing recursions and mutual recursions in declarations, and communications between threads, are interesting research topics. We take a closer look at some of these extensions.

One of the extensions to the explicit deallocation primitive is widening the operation range of the primitive `del` so that it can delete an instance outside the local store. For example, if there is no instance of x in the local store when executing `del` x, the instance of x in the store which are 'closest' to the local store on the path from the local store to the root will be deleted. The type system could relax the requirement in the typing rules

SCOPE and PARALLEL to allow non-empty stores in the environments.

$$\text{(SCOPE)}\quad \frac{\sigma, \Gamma \vdash A : X}{\sigma, \Gamma \vdash \{A\} : \langle X^i, [] \cap X^o, [] \cap X^l \rangle}$$

$$\text{(PARALLEL)}\quad \frac{\sigma_1, \Gamma \vdash A : X \quad \sigma_2, \Gamma \vdash B : Y}{\sigma_1 + \sigma_2, \Gamma \vdash (A \parallel B) : \langle X^i + Y^i, X^o + Y^o, X^l + Y^l \rangle}$$

We conjecture that the above rules are adequate for the mentioned behaviour, but this has still to be proved.

Another extension as mentioned at the end of Section 6.1.2 is to allow sibling threads to reuse each other instances, thus maximizing reusability. To this end, expression ($\texttt{reu}\, d \parallel \texttt{reu}\, d$) has a sharp upper bound of one d, instead of two in the model of Chapter 6, assuming that d is a primitive component.

A combination of all language features of Chapters 4 and 6 is another possible consideration. Recall that the semantics of X^l in Chapter 4 is the exact lower bound of instances while the semantics of X^l in Chapter 6 is the least lower bound—one or none. It seems that if we can calculate the exact lower bound X^l in Chapter 6, then the combination of the two type systems is quite feasible, by adding a store in the typing environment and using signed multisets for types.

Last, consider the following program with a recursion in the declaration of component a.

$$\begin{array}{l} d \prec \epsilon \quad e \prec \epsilon \\ a \prec (\{\texttt{new}\, d\}\, \texttt{reu}\, a + \texttt{new}\, e); \\ \texttt{new}\, a \end{array}$$

We assume that the operational semantics for the program is the one of Chapter 5. Then we can see that the main expression of the program $\texttt{new}\, a$ is bounded by $[a, d, e]$, despite that the program has an infinite execution trace. Building type systems for languages with this kind of recursion or other kinds of recursion is a very interesting research direction.

Bibliography

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 49–65, New York, NY, USA, 1997. ACM Press.

[2] John Backus. The history of FORTRAN I, II, and III. In *HOPL–1: The first ACM SIGPLAN conference on History of programming languages*, pages 165–180, New York, NY, USA, 1978. ACM Press.

[3] Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for Java. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 132–145, New York, NY, USA, 1997. ACM Press.

[4] Hendrik P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford Univertity Press, 1992.

[5] Marc Bezem and Hoang Truong. Counting instances of software component. In Didier Galmiche, Peter O'Hearn, and David J. Pym, editors, *Proceedings of the ICALP/LICS Workshop on Logics for Resources, Processes, and Programs,* July 2004. Submitted to Journal of Logic and Computation Semantics Corner.

[6] Marc Bezem and Hoang Truong. A type system for the safe instantiation of components. *Electronic Notes in Theoretical Computer Science*, 97:197–217, 2004.

[7] Bror Bjerner and S. Holmström. A composition approach to time analysis of first order lazy functional programs. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 157–165, New York, NY, USA, 1989. ACM Press.

[8] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[9] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.

[10] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher–Order and Symbolic Computation*, 14(2–3):261–300, 2001.

[11] Karl Crary. Toward a foundational typed assembly language. *ACM SIGPLAN Notices*, 38(1):198–212, January 2003.

[12] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the International Conference on Functional Programming*, pages 233–248, Paris, France, September 1999.

[13] Karl Crary and Stephnie Weirich. Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN–SIGACT symposium on Principles of programming languages*, pages 184–198, New York, NY, USA, 2000. ACM Press.

[14] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[15] Vincent Dornic, Pierre Jouvelot, and David K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Lett. Program. Lang. Syst.*, 1(1):33–45, 1992.

[16] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher–order modules. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 236–249, New York, NY, USA, 2003. ACM Press.

[17] B. Dushnik and E. W. Miller. Partially ordered sets. *American Journal of Mathematics*, pages 600–610, 1941.

[18] Cormac Flanagan and Stephen N. Freund. Type–based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object–Oriented Software*. Addison–Wesley Professional Computing Series. Addison–Wesley Publishing Company, New York, NY, 1995.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley Professional Computing Series. Addison–Wesley Publishing Company, New York, NY, 1995.

[21] Andrew D. Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 77, Washington, DC, USA, 2002. IEEE Computer Society.

[22] Object Management Group. *CORBA Components, v3.0*, June 2002.

[23] Martin Hofmann. Linear types and non size–increasing polynomial time computation. In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, page 464, Washington, DC, USA, 1999. IEEE Computer Society.

[24] Martin Hofmann. The strength of non-size increasing computation. In *POPL '02: Proceedings of the 29th ACM SIGPLAN–SIGACT symposium on Principles of programming languages*, pages 260–269, New York, NY, USA, 2002. ACM Press.

[25] M. Douglas McIlroy. Mass–produced software components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, October 1968.

[26] Erik Meijer and Clemens Szyperski. Overcoming independent extensibility challenges. *Communications of the ACM*, 45(10):41–44, 2002.

[27] John C. Mitchell. *Type systems for programming languages.* MIT Press, Cambridge, MA, USA, 1990.

[28] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 3rd edition, October 2001.

[29] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 114–136, London, UK, 1999. Springer–Verlag.

[30] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2003.

[31] Benjamin C. Pierce, editor. *Types and Programming Languages*. MIT Press, 2002.

[32] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[33] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[34] Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 65–78, New York, NY, USA, 1994. ACM Press.

[35] Johannes Sametinger. *Software engineering with reusable components*. Springer–Verlag New York, Inc., New York, NY, USA, 1997.

[36] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison–Wesley, 1995.

[37] Clemens Szyperski. *Component Software*. Addison–Wesley, 1st edition, 1998.

[38] Clemens Szyperski. *Component Software—Beyond Object–Oriented Programming*. Addison–Wesley / ACM Press, 2nd edition, 2002.

[39] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[40] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[41] Thuan L. Thai and Hoang Q. Lam. *.NET Framework Essentials*. A Nutshell Handbook. O'Reilly & Associates, Inc., 3rd edition, August 2003.

[42] Hoang Truong. Guaranteeing resource bounds for component software. In Martin Steffen and Gianluigi Zavattaro, editors, *FMOODS*, volume 3535 of *Lecture Notes in Computer Science*, pages 179–194. Springer–Verlag, 2005.

[43] Hoang Truong and Marc Bezem. Finding resource bounds in the presence of explicit deallocation. In Dang Van Hung and Martin Wirsing, editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2005.

[44] Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, April 1990. North–Holland.

[45] David Walker, Karl Crary, and J. Gregory Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):701–771, 2000.

[46] Shengchao Qin Wei-Ngan Chin, Huu Hai Nguyen and Martin Rinard. Memory usage verification for OO programs. In Chris Hankin and Igor Siveroni, editors, *The 12th International Static Analysis Symposium (SAS '05)*, London, UK, September 2005.

[47] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[48] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 169–180, New York, NY, USA, 2001. ACM Press.

[49] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, New York, NY, USA, 1998. ACM Press.

[50] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN–SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM Press.

[51] Matthias Zenger. Type–safe prototype–based component evolution. In Boris Magnusson, editor, *ECOOP 2002—Object-Oriented Programming, 16th European Conference, Málaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 441–469, Berlin, June 2002. Springer–Verlag.