UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

PROGRAMMING MULTICORES SAFELY: HANDLING BARRIER DEADLOCKS

Tiago Soares Cogumbreiro Garcia

DOUTORAMENTO EM INFORMÁTICA
Especialidade em Ciência da Computação

2015

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

PROGRAMMING MULTICORES SAFELY: HANDLING BARRIER DEADLOCKS

Tiago Soares Cogumbreiro Garcia

Tese orientada pelo
Prof. Doutor Francisco Cipriano da Cunha Martins
especialmente elaborada para a obtenção do grau de doutor no ramo de Informática, especialidade de Ciência da Computação

2015

Resumo

Actualmente, a generalidade dos dispositivos de computação inclui um processador *multicore*. As aplicações que correm em processadores *multicore* só aumentam o seu desempenho se computarem em paralelo, aproveitando assim o poder computacional dos núcleos disponíveis. Para este efeito, as linguagens de programação mais populares, tal como Java e C♯, adoptaram, nos últimos anos, várias técnicas de programação paralela. Esta tese lida com uma classe de falhas que origina da utilização de uma técnica de programação paralela, chamada barreira, cuja funcionalidade é a de sincronizar grupos de tarefas. Uma barreira coordena a ordem de execução de um grupo de tarefas, disponibilizando um ponto de execução em que as várias tarefas dum grupo podem esperar umas pelas outras. As tarefas que usam barreiras são vulneráveis ao problema de *impasse*, em que pelo menos duas tarefas estão (indirectamente) à espera uma da outra em barreiras diferentes sem que qualquer uma das tarefas possa avançar. Os impasses constituem uma classe de falhas, da área de concorrência, com grande impacto em programas paralelos. O nosso objectivo é aumentar a produtividade da programação paralela tratando do problema de impasses em barreiras. Nesta tese propomos duas técnicas complementares para lidar com o problema de impasses: uma ferramenta de verificação especializada em impasses sobre barreiras que é distribuída, tolerante a falhas e verifica aplicações X10 e Java; um modelo de programação isento de impasses.

Palavras-chave: impasse, barreira, sincronização, verificação, programação paralela, programação distribuída, Java, X10.

Resumo estendido

Actualmente, a generalidade dos dispositivos de computação inclui um processador *multicore*, constituído por vários elementos de processamento (chamados núcleos). As aplicações feitas ignorando múltiplos núcleos, só aumentam o desempenho com o aumento da velocidade de cada núcleo. No entanto, por restrições físicas, os fabricantes de processadores *multicores* deixaram de aumentar a velocidade dos núcleos e, ao invés disso, aumentam o número núcleos disponíveis em cada processador. As aplicações que correm em processadores *multicore* só aumentam o seu desempenho se computarem em paralelo, aproveitando assim o poder computacional dos núcleos disponíveis. Para este efeito, as linguagens de programação mais populares, tal como Java e C♯, adoptaram, nos últimos anos, várias técnicas de programação paralela.

Esta tese lida com uma classe de falhas que origina da utilização de uma técnica de programação paralela, chamada barreira, cuja funcionalidade é a de sincronizar grupos de tarefas. Uma barreira coordena a ordem de execução de um grupo de tarefas, disponibilizando um ponto de execução em que as várias tarefas dum grupo podem esperar umas pelas outras. Isto é, a tarefa bloqueia ao executar a instrução barreira até que todos os membros do grupo executem a mesma barreira ("cheguem" à barreira). As tarefas que usam barreiras são vulneráveis ao problema de *impasse*, em que pelo menos duas tarefas estão (indirectamente) à espera uma da outra em barreiras diferentes sem que qualquer uma das tarefas possa avançar. Os impasses constituem uma classe de falhas, da área de concorrência, com grande impacto em programas paralelos.

O nosso objectivo é aumentar a produtividade da programação paralela tratando do problema de impasses em barreiras. Com o intuito de tratar rigorosamente este problema, surge a necessidade de caracterizar matematicamente (*i.e.,* formalizar) o mecanismo de sincronização existente numa barreira. Nesta tese, fazemos um levantamento de primitivas de sincronização baseadas em barreiras. Com esta pesquisa concluímos que o construtor *phaser*, que origina da linguagem Habanero Java, consegue ser adaptado para representar os vários padrões de sincronização por nós documentado. A nossa primeira contribuição é a linguagem de programação paralela Brenner que define as operações essenciais para representar os vários padrões de sincronização capturados por *phasers*.

Propomos duas técnicas complementares para lidar com o problema de impasses. A primeira técnica, considerada *dinâmica*, consiste numa análise contínua da execução do programa, com o objetivo de identificar situações de impasse. A segunda técnica, considerada *estática*, propõe um modelo de programação isento de impasses. Técnicas dinâmicas são mais gerais que as técnicas estáticas, podendo ser aplicadas a mais programas existentes, mas incorrem numa degradação da velocidade de execução. Técnicas estáticas, embora menos gerais, garantem propriedades—neste caso a ausências de impasses—sem influenciar o desempenho do programa. Qualquer programa que respeite o nosso modelo de programação não sofrerá de impasses causados por barreiras.

Verificação dinâmica Propomos a ferramenta Armus que é capaz de verificar aplicações Java e X10, e que incorpora a nossa técnica de verificação dinâmica. Utilizamos a linguagem Brenner e as operações sobre *phasers* como base da nossa técnica de verificação. Com esta base conseguimos identificar mais padrões de sincronização que as técnicas disponíveis em trabalho relacionado. Adicionalmente, Armus é a primeira ferramenta de verificação que identificas impasses sobre barreiras nas linguagens Java e X10.

O problema da verificação dinâmica de impasses pode ser visto como um sistema de restrições em que quando não existe solução, estamos na situação indesejada de impasse. As restrições correspondem a dependências de concorrência existentes entre tarefas e mecanismos de sincronização. Por exemplo, podemos representar uma dependência entre a tarefa bloqueada numa barreira e as tarefas participantes nessa barreira que ainda não a executaram. No contexto da verificação dinâmica de impasses, a teoria de grafos é a mais utilizada para modelar dependências de concorrência. No nosso caso, resolver as dependências de concorrência equivale a encontrar um ciclo num grafo. O Armus escolhe entre duas representações de dependências para gerar grafos mais pequenos, que demoram menos tempo a serem analisados. Para isso mostramos formalmente que a existência de um ciclo num grafo da primeira representação implica a existência de um ciclo no grafo da segunda representação, e vice-versa. Uma das representações favorece situações em que há mais tarefas do que barreiras, a outra representação favorece a situação oposta. Mostramos mais dois resultados cruciais para garantir a correcção da nossa técnica de verificação: (i) a verificação é *fidedigna*, visto que qualquer situação de impasse corresponde a uma situação de impasse no programa; (ii) a verificação é *completa*, visto que qualquer caso em que o programa esteja num impasse é identificado pela análise.

A linguagem de programação X10 permite o desenvolvimento de programas distribuídos. Um programa distribuído corre simultaneamente em vários computadores, utilizando os seus recursos. A nossa técnica de análise melhora o

estado da arte na verificação dinâmica de programas distribuídos. O trabalho relacionado só inclui informação sobre as tarefas bloqueadas, o que é insuficiente para verificar padrões de sincronização em que os participantes de uma barreira não são conhecidos *a priori*. Para suplantar esta limitação, a nossa técnica regista adicionalmente as barreiras a que cada tarefa bloqueada ainda não chegou. Uma implicação desta novidade é que com esta informação extra, a análise consegue ser efectuada em qualquer computador da rede sem sincronização adicional. Em contraste, o trabalho relacionado necessita de introduzir sincronização adicional entre as várias máquinas que estão a ser analisadas.

Avaliamos o Armus usando três conjuntos de *benchmarks*, em cenários locais e distribuídos numa máquina com 64 núcleos. No primeiro conjunto de *benchmarks* avaliamos o impacto que a verificação tem no tempo de execução de programas Java paralelos, num contexto local. O impacto de verificação é na maioria nulo e no pior dos casos a aplicação demora mais 15% to tempo a terminar. No segundo conjunto de *benchmarks* avaliamos o impacto que a verificação tem no tempo de execução de programas X10 distribuídos. O impacto de verificação é nulo. No terceiro conjunto de *benchmarks* avaliamos o impacto que a escolha de um modelo de grafos tem na análise, num contexto local. Os resultados da avaliação mostram que a nossa técnica de selecção automática de modelos de grafos pode aumentar encurtar o tempo da análise até 7 vezes, *versus* a técnica efectuada por trabalho relacionado que só utiliza um modelo de grafos.

Verificação estática Este trabalho visa criar um modelo de programação isento de impasses adaptando a linguagem Brenner. Usamos como ponto de partida o modelo de programação isento de impasses existente na linguagem X10, que é mais limitada em termos de padrões de sincronização do que Brenner. Os padrões de sincronização extra na linguagem Brenner são o produtor-consumidor com e sem limite, necessário para representar o paralelismo *streaming* e o paralelismo *pipeline*. O nosso trabalho também pode ser visto como uma adaptação da linguagem Habanero-Java para um modelo de programação isento de impasses, que adicionalmente unifica as várias funcionalidades propostas para os *phasers*. O modelo de programação é formalizado na linguagem SBrenner, que é composto por uma definição de uma sintaxe, de uma semântica operacional e dum sistema de tipos. A semântica operacional caracteriza matematicamente o estado da computação e o efeito que cada instrução tem neste estado. Um sistema de tipos consiste num conjunto de regras que especificam o comportamento válido da linguagem, identificando estados válidos e inválidos de computação.

Os dois resultados principais que mostramos são: a preservação de tipos e o progresso. A preservação de tipos garante que: dado um estado válido

identificado pelo sistema de tipos, se alguma tarefa executar uma instrução, então o resultado do estado de computação também é considerado válido pelo sistema de tipos. Este resultado mostra que, partindo dum estado válido, a computação não "chega" a um estado inválido.

O resultado de progresso garante que, para qualquer estado válido, existe um passo de computação ou então que a computação termina. O progresso garante a ausência de impasses, pois nesse caso a computação não termina mas é impossível dar um passo de computação. Para conseguir provar que há sempre uma tarefa que está pronta a executar, e visto estarmos na presença do padrão de sincronização de produtor-consumidor limitado, introduzimos uma invariante em sistemas que sincronizam com *phasers*. Uma tarefa a dista n fases de uma tarefa b se e só se para qualquer *phaser* em que ambas tarefas participem a diferença da fase local entre a tarefa a e b é de n. Um *phaser* pode ser visto como uma série de barreiras, a fase local n representa a n-ésima barreira desta série. A invariante do sistema de *phasers* é que, embora a diferença de fases entre tarefas possa alterar à medida que o programa executa, existe sempre uma diferença de fases entre quaisquer duas tarefas. Desta invariante conseguimos estabelecer uma ordem total sobre as tarefas que executam e mostramos que a menor destas tarefas não está bloqueada em qualquer *phaser*.

Abstract

Nowadays, most produced computing devices include multicore processors. Applications that run on these devices only scale if they can compute in parallel. To this end, mainstream programming languages, like Java and C♯, adopted various parallel programming techniques.

This thesis focuses on a parallel technique, called barrier, used for synchronisation. A barrier coordinates the execution order of parallel activities, by letting them wait for each other. Tasks using barriers are susceptible to the problem of deadlocks, where at least two activities are (indirectly) in a stalemate because of a conflicting ordering of some barriers. Deadlocks are a class of concurrency failures with a big impact in parallel programs.

To help make parallel programming more productive, we propose two complementary techniques that handle deadlocks caused by barriers: a runtime verification tool, and a deadlock-free programming model. We present Armus, a runtime verification tool specialised in barrier deadlocks that is distributed, fault-tolerant, and verifies X10 and Java programs. Our technique verifies more barrier synchronisation patterns than existing state-of-the-art techniques. We improve deadlock verification based on graph analysis: our technique selects from two alternative graph representations of concurrency dependencies to hasten deadlock checking. Armus is evaluated with three benchmark suites in local and distributed scenarios.

To handle barrier deadlocks at design time we propose a language called SBrenner that extends and formalises a programming model that originates from the Habanero-Java and the X10 languages. The outcome is a deadlock-free programming model that leverages pipeline parallelism. We present an operational semantics and a type system for SBrenner. Our type system enjoys the properties of progress and subject reduction.

Keywords: deadlock, barrier, synchronisation, verification, parallel programming, distributed programming, Java, X10.

Acknowledgements

I am extremely grateful to my advisor, Francisco Martins, for his support and enthusiasm. His passion for education and science are a constant source of inspiration. I am grateful for Francisco's attention to detail, rigour, and wisdom.

I am in debt to Vivek Sarkar for welcoming me at Rice University and for our thoughtful discussions. I also want to thank Vivek Sarkar and Jun Shirako for the discussions regarding the semantics of phasers and Habanero Java.

During my doctorate studies I had the privilege to work at Universidade de Lisboa, Rice University, and the Imperial College London. Thank you to Vasco Vasconcelos, Nobuko Yoshida, and Raymond Hu for our collaborations, which greatly improved the outcome of this disseration. I am grateful to the members of LASIGE (Universidade de Lisboa), the Habanero group (Rice University), and the Mobility Reading Group (Imperial College London) for the friendly and insightful discussions.

Contents

Contents **x**

List of Figures **xi**

List of Tables **xii**

1 Introduction **1**
1.1 Problem statement . . . 1
1.2 Objectives . . . 4
1.3 Thesis outline . . . 5

2 Barriers and its applications **7**
2.1 Historical background . . . 7
2.2 Related work . . . 13

3 Brenner **15**
3.1 Syntax . . . 15
3.2 Operational Semantics . . . 20

4 Runtime deadlock verification **27**
4.1 Resource dependencies . . . 27
4.2 Basic graph theory . . . 29
4.3 Graph-based deadlock identification . . . 30
4.4 Results . . . 32
4.5 Armus: a tool for runtime deadlock verification . . . 38
4.6 Evaluation . . . 42

5 Deadlock prevention **47**
5.1 Language restrictions . . . 47
5.2 Syntax . . . 53
5.3 Operational Semantics . . . 56
5.4 Type System . . . 67

6 Type system properties **73**
6.1 Typing phaser maps . . . 73
6.2 Typing states . . . 77
6.3 Inversion . . . 79
6.4 The domain of typing contexts . . . 86
6.5 Strengthening . . . 89

7 Subject reduction **93**
7.1 Async . . . 93
7.2 Phaser creation . . . 104
7.3 Deregistration . . . 113
7.4 Advance phase . . . 115
7.5 Change bound . . . 118
7.6 Await . . . 119
7.7 Next . . . 119
7.8 Finish . . . 128
7.9 Join . . . 129
7.10 Control flow . . . 129
7.11 Main result . . . 131

8 Progress **139**

9 Conclusion **149**
9.1 Contributions . . . 149
9.2 Summary of personal publications . . . 151
9.3 Future work . . . 152

Bibliography **155**

List of Figures

3.1 Top-level syntax. . . . 19
3.2 Syntax of the abstract machine. . . . 20
3.3 The small-step semantics of Brenner (tasks). . . . 21
3.4 The small-step semantics of Brenner (phasers). . . . 22
3.5 Small step semantics for control flow instructions $\boxed{c; b \to b}$. . . . 23

4.1 Two different graphs representing a deadlocked system. 32
4.2 Comparative execution time for non-distributed benchmarks (lower means faster). 44
4.3 Comparative execution time for distributed deadlock detection (lower means faster). 45
4.4 Comparative execution time for different graph model choices (lower means faster), using deadlock avoidance. 45
4.5 Comparative execution time for different graph model choices (lower means faster), using deadlock detection. 45

5.1 SBrenner syntax. 54
5.2 Syntax of the abstract machine. 61
5.3 A dependency tree of tasks. 62
5.4 Small step semantics for states (phaser related) $\boxed{S \to S}$. 63
5.5 Small step semantics for states (finish, control flow) $\boxed{S \to S}$. 67
5.6 Small step semantics for control flow instructions $\boxed{c; b \to b}$. 67
5.7 Typing rules for arguments $\boxed{\Gamma \vdash s \colon \Gamma}$. 68
5.8 Typing rules for instructions $\boxed{\Gamma \vdash i \colon \Gamma}$. 70
5.9 Typing rules for programs $\boxed{\Gamma \vdash b \colon \Gamma}$. 70

6.1 Typing rules for phasers and phaser maps. 76
6.2 Typing rules for permissions, bounds, tasks, and task maps. 78
6.3 Typing rules for states. 79

List of Tables

4.1 Relative execution overhead in detection mode. 43
4.2 Relative execution overhead in avoidance mode. 44
4.3 Edge count and verification overhead per benchmark per graph mode. 46

Chapter One

Introduction

We want to improve the productivity of multicore programmers, by reducing program failures. Recently, mainstream languages incorporated parallel programming techniques that introduced new forms of failures. Our work studies a technique called barrier synchronisation and a form of failure called a deadlock. We propose two different strategies to solve barrier deadlocks and implement one of these strategies as a tool.

1.1 Problem statement

This thesis centres on the impact of the "multicore revolution" on programming languages. Physical restrictions made chip manufacturers develop multicore processors composed of multiple processing units, called cores. The execution model of these processors is *parallel*: all cores work at the same time and communicate through a main memory. Each core executes *sequentially*, running a sequence of instructions, one after the other. The trend of processor design for the past decade has been to increase the number of cores linearly and to stabilise the execution speed [98].

Applications stopped scaling with multicores because they were not designed to compute in parallel [83]. The reaction of mainstream programming languages, like Java, and C♯, was to incorporate parallel programming techniques. Surveys [20, 82] analyse the usage parallel programming techniques in Java and C♯. At the foundation of parallel programming is task parallelism [27], which is both a *programming model* that defines a an interface for programming a system, and also an *execution model* that defines how actions are executed by the system. Tasks correspond to a logical unit of work that is composed of a series of actions that are executed sequentially, while sharing and manipulating memory. The system executes multiple tasks *concurrently*, *i.e.,* at the same time.

With parallelism comes the need to synchronise the concurrent execution of multiple tasks. A barrier [55] coordinates the execution of a group of tasks: it can ensure all members wait for each other before advancing to their next action. Barrier synchronisation was developed in 1978 and rapidly became a cornerstone

of parallel programming. The most popular choices for parallel programming are MPI [40] and OpenMP [36], through C or Fortran. The performance of barrier synchronisation is of such high importance that active research exists on implementing this mechanism in hardware [2, 54, 93].

Mainstream programming languages provide a multitude of abstractions that perform barrier synchronisation [20, 82]. Java 5.0–7.0 (2004–2011) and C♯ (2010) introduced four abstractions that use barrier synchronisation: latches, cyclic barriers, futures [48], and the fork/join programming model. Futures are also a novelty of C++ version 2011. A latch, or countdown event, is a counter that can only be decreased until it reaches zero. Besides decreasing the value of the latch, tasks can also wait (once) for the latch to reach zero. The wait on the latch represents a barrier synchronisation. A cyclic barrier can be used for synchronisation repeatedly. A future, or promise, is a placeholder for a value that is computed asynchronously (concurrently). Any task holding the future can wait for its value to be computed in what can be seen as a barrier synchronisation. Stream programming [100] also includes a form of reusable barrier synchronisation and is included in Java, C♯, and there is a proposal for OpenMP [87].

High-performance computing (HPC) champions the development of parallel programming to solve complex scientific problems. There is a recent interest in two well-studied aspects of mainstream languages [45, 71, 74]: language usability and application robustness. To address language usability, new task parallel languages were proposed: Chapel [26], Habanero-Java (HJ) [24], Titanium [110], UPC [37], and X10 [28]. All five languages include abstractions to perform barrier synchronisation. There many proposals to tackling the problem of concurrency-related failures: structured abstractions [28, 94], source code analysis [106, 76], and program monitoring [34, 51, 52].

Parallel programming techniques introduce concurrency-related bugs, notably difficult to track and reproduce. Deadlocks [114] are a class of nefarious concurrency failures that have a wide expression in task parallel programs [39, 72]. Barrier deadlocks arise from a cyclic-dependency among tasks that participate on multiple barriers. An example of a deadlock is when two tasks block on distinct barriers and are (indirectly) waiting for each other. Techniques that handle deadlocks for mainstream languages cannot cope with *ad hoc* synchronisation mechanisms [57]. In particular, these techniques cannot handle barrier deadlocks.

Literature considers four strategies to handle deadlocks [58]: ignoring, preventing, avoiding, and detecting. The plainest strategy is to just *ignore* the error, a useful strategy when the deadlock only shows up rarely, and the effort of handling it is steep. Barrier deadlocks are usually deterministic, because the arrival order does not disturb barrier synchronisation. It is often the case that if

the program can deadlock, it does deadlock, which renders impractical, ignoring these class of failures.

Prevention reduces the expressiveness of the abstraction in such a way that any program that runs is known beforehand to be free of deadlocks. To achieve prevention there are syntactic and semantic approaches. The fork/join programming model of OpenMP and of X10 uses syntactic scoping to ensure that there are no deadlocks while tasks join their execution, *i.e.,* there is no syntax to write a program that deadlocks just by joining the execution of tasks. The fork/join programming model is also available in many languages as a library, but in such cases deadlocks are not prevented syntactically. The X10 language [28] provides a limited programming model that prevents all barrier deadlocks, although any use of synchronisation codes outside of this programming model voids the deadlock freedom guarantee. An alternative approach is to perform source code analysis to predict deadlocks. Promising work from Le *et al.* [69] annotates C code to verify some safety properties of barriers. Prevention is too limiting to be applied to the whole system, so language designers use this strategy to eliminate some deadlock patterns.

Avoidance and detection happen at run-time. With *avoidance*, the system proactively compensates calls that lead to a deadlock, *e.g.*, by delaying a call, or aborting execution. Some systems, such as MPI [40], deadlock if a barrier participant forgets to synchronise and terminates. The synchronisation algorithm of X10 disregards terminated participants to avoid this class of deadlocks. The HJ language dynamically avoids deadlocks that arise from the interference between cyclic barriers and the fork/join programming model at a cost of expressiveness. Avoiding deadlocks caused by misaligned barriers is usually too expensive as every call that uses barrier synchronisation must be monitored, so language designers opt for deadlock detection instead.

The strategy of *detection* requires a system that is capable of introspecting its state to identify deadlocked states. This monitoring does not interfere with the program execution, so the system must break—or just inform the user of—any deadlock it identifies. Works on barrier deadlock detection are largely concerned with the idiosyncrasies of the system and with the performance of the tool at hand. For example, works on deadlock detection for MPI, *e.g.*, [52], are not applicable to UPC, nor *vice versa* [34], because neither has a barrier construct capable of encoding the other. In the context of parallel programming, there is a need for run-time error detection [74] and a need for formalisation [45]. The programming language and formal methods community payed little attention to runtime strategies that handle deadlocks, *i.e.,* to detection and to avoidance. A notable exception is the work from Boudol [21] that presents a language equipped with locks along with a formal semantics that avoids deadlocks.

There is a lack of a precise, mathematical description of what barrier synchro-

nisation actually is. Throughout its 30 years of existence, there are many different ways of employing barrier synchronisation but no surveys on its fundamental semantics. Formal methods can help assess the correctness of the design early on and acts as a crucial guide to a more accurate implementation [19]. In particular, runtime techniques cannot verify existing barrier-based abstractions and prevention techniques are too limited to be useful.

1.2 Objectives

We want improve the productivity of parallel programming, by reducing the number of faults caused by concurrency. This thesis revisits the classical problem of deadlocks in the point of view of programming languages, in particular we focus on a *comprehensive* approach to handle deadlocks caused by barriers. Our objectives can be summarised in five topics:

1. Survey the usual barrier properties. We study the origins of parallel programming and associated programming models to identify the abstractions that use this synchronisation mechanism. We catalogue the properties found and illustrate them with programming examples.

2. Propose a general theoretical framework to reason about barriers. The idea is to distil the semantics of the surveyed properties into a single, unifying abstraction that can then be used as the cornerstone of our techniques to handle deadlocks.

3. Introduce techniques that handle deadlocks at runtime. We show how to detect (or avoid) deadlocks on our general barrier framework. Dynamic techniques are the only ones that can cope with the full expressiveness of barrier synchronisation.

4. Present deadlock-free techniques. We restrict our initial model to include the prevention techniques used by X10 and HJ. Our proposal pushes the limits of expressiveness set forth by X10 and HJ while maintaining the deadlock-free guarantee, that we prove to be hold.

5. Develop tools that can help programmers avoid barrier deadlocks. We implement the synchronisation mechanism behind our theoretical framework and build two tools that detect or avoid barrier deadlocks for Java and X10.

1.3 Thesis outline

Chapter 2 surveys the properties of abstractions that use barrier synchronisation. We give a brief historical context of the evolution and the different uses of this synchronisation mechanism. Each barrier idiom is accompanied with code listings and an informal description of its semantics. We conclude the chapter discussing related work on the formalisation of barrier synchronisation and on handling barrier deadlocks.

Chapter 3 presents a minimal task parallel language that contains a general barrier abstraction. The language is defined by a syntax and a formal semantics. We describes the semantics operationally: so we characterise the state of a program, and specify how each primitive alters this state.

Chapter 4 explores the detection and avoidance of barrier deadlocks. The basic idea is to abstract the state of a program as a graph, and then reduce the problem of deadlock detection and avoidance to finding a cycle in a graph. We introduce some basic notions of graph theory, then show how to obtain a graph from a program state, and finally establish the soundness and completeness of our detection algorithm. The realisation of the theory are two runtime verification tools: one for Java, and another for one X10. The tools can perform deadlock avoidance, and fault-tolerant and distributed deadlock detection.

Chapter 5 explores the prevention of barrier deadlocks. We restrict the language introduced in Chapter 3 in such a way that programs are deadlock-free by construction, *i.e.,* there is no syntax to write programs that deadlock. The syntactic and semantic limitations we impose to achieve deadlock freedom are taken from the languages X10 and HJ. Our contributions of this chapter are: pushing the limits of expressiveness under a deadlock-free setting, and establishing the properties of subject reduction and of progress.

Chapter 9 summarises the thesis, outlines our technical contributions and key findings, and presents future directions of our work.

Chapter Two

Barriers and its applications

Parallel programming includes several abstractions that perform barrier synchronisation. We examine common properties of this synchronisation mechanism with the objective of identifying a single unifying abstraction to reason about barrier deadlocks.

Section 2.1 highlights the different uses of barrier synchronisation in programming languages through the history of computing. The outcome is a survey on different barrier properties. In Section 2.2, we examine the state-of-the-art on handling barrier deadlocks to identify our research opportunities.

2.1 Historical background

The importance of coordinating the execution of independent processing units (tasks, processors, or even computers) can be traced back to the first computers ever designed. The 1960's brings into play the simplest form of barrier synchronisation, the fork/join programming model. The Gamma 60 computer [18] is announced in 1958, a machine that includes multiple processing units that can synchronise upon the completion of an instruction. Any processing unit can run an instruction on another unit with a "fork" instruction and then wait for that instruction to complete with a "join" instruction. In 1963, Melvin E. Conway proposes a multiprocessor design [33] based the fork/join programming model, where the join instruction can wait for multiple instructions to conclude, instead of just one. John A. Gosden presents a historical survey of this subject in [46].

In 1965, Ascher Opler elevates the fork/join programming model to a lan-

Listing 2.1: Matrix multiplication in Fortran.

```
1   do LOOP I=1,21
2   do LOOP J=1,21
3   do LOOP K=1,21
4   LOOP: C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

Listing 2.2: Matrix multiplication programmed with do-together.

```
1  do together BLOCK1, BLOCK2, (END)
2  BLOCK1: do LOOP1 I1=1,21,2
3           do LOOP1 J1=1,21
4           do LOOP1 K1=1,21
5  LOOP1:   C(I1,J1) = C(I1,J1) + A(I1,K1) * B(K1,J1)
6  BLOCK2: do LOOP2 I2 = 2,20,2
7           do LOOP2 J2 = 1,21
8           do LOOP2 K2 = 1,21
9  LOOP2:   C(12,J2) = C(I2,J2) + A(I2,K2) * B(K2,J2)
10 END: hold
```

guage abstraction [84] called do-together. Listing 2.1 is a sequential program that multiplies matrices `A` and `B` and places the result in matrix `C`. The matrices are 21 rows by 21 columns. Listing 2.2 is the parallel version of the sequential algorithm. Instruction DO TOGETHER receives the instruction sequences to be executed in parallel and an instruction label between parenthesis that marks the end of the block—the language used in the example lacks the concept of structured code. Here, there are two instruction sequences: `BLOCK1`, that ranges from lines 2–5, and `BLOCK2`, that ranges from lines 6–9. The instruction sequence `BLOCK1` multiplies cells with odd rows, and instruction sequence `BLOCK2` multiplies cells with even rows. Instruction `hold` is a join barrier that waits for the completion of both instruction sequences.

Listing 2.3: Matrix multiplication using Lamport's concurrent-do.

```
1  do LOOP conc I=1,21
2  do LOOP J=1,21
3  do LOOP K=1,21
4  LOOP: C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

The late 1960's bring the ILLIAC IV [17], a computer with multiple processors that execute a single instruction stream. Leslie Lamport proposes two language constructs to coordinate the execution of parallel loops [66]. Any DO-loop that includes the keyword CONC schedules each iteration to a different processor. Listing 2.3 revisits the matrix multiplication, but assigns the computation of each row to a different processor. Similarly to the DO TOGETHER, there is an implicit barrier at the end of the outermost cycle, where all processors synchronise. A DO-loop with the keyword `sim` also schedules each iteration to a different

Listing 2.4: An iterative averaging algorithm.

```
1 for (i=1;i<=N;i++) do par
2   for (k=1;k<=M;k++) do seq
3     P[i] = (P[(i+1) % N] + P[(i-1) % N])/2;
```

processor, but differently than CONC, it makes every processor executing the parallel cycle to synchronises at each instruction. This is the first time the notion of a *reusable* barrier synchronisation appears in a programming language, although in this case it is an implicit notion. A reusable barrier can also be seen as a stream of barriers. We call *phase* to each barrier of a stream of barriers.

Harry F. Jordan coins the term "barrier synchronisation" in 1978 [55], in the context of the design of a parallel machine that performs finite element analysis. The author proposes a primitive to perform reusable barrier synchronisation. The intent of this synchronisation mechanism is to separate two phases of the element analysis algorithm. The processors must wait for each other at the barrier before advancing to the second phase of the algorithm.

The 1980's are marked, at the software level, by the exploration of parallelising compilers that take a sequential program and make it parallel [108]. Parallelising compiler introduce barriers in the generated code to enforce data dependence. Programmers can provide source code annotations to improve the work of the compiler. The programming languages Force [56]—initiated by Harry F. Jordan, among others—and PISCES [89] included a reusable barrier primitive.

Listing 2.4 shows a typical smoothing algorithm picked from [47], a pattern seen, for example, in computing a partial differential equation. The array `P` holds `N` numbers. Each value in the array is calculated by using its neighbours from the previous iteration. The outer loop iterates over the contents of the array and executes its steps in parallel (hence the keyword `par`). The inner loop performs the smoothing and executes sequentially.

A parallelising compiler must notice the data dependency between iteration i and iteration $i + 1$, or otherwise there is a race condition. In Listing 2.5, the compiler adds two (reusable) barriers. Every task waits for the others to read the neighbouring values into `tmp`, and then all tasks wait for each other after updating their own cell.

Rajiv Gupta introduces fuzzy barriers [47], in 1989, as an optimisation technique to overlap synchronisation with computation. A similar technique, called *split-phase communication*, is used to hide communication latency [25, 29, 113], so nowadays fuzzy barriers are also known as split-phase barriers. A split-phase barrier consists of two primitives: `initBarrier` initiates the synchronisation

Listing 2.5: Iterative averaging programmed with barrier synchronisation.

```
1 for (i=1;i<=N;i++) do par
2   for (k=1;k<=M;k++) do seq {
3     tmp = (P[(i+1) % N] + P[(i-1) % N])/2;
4     barrier;
5     P[i] = tmp;
6     barrier;}
```

mechanism concurrently, and primitive `waitBarrier` waits for the synchronisation to happen. A task blocks on `waitBarrier` until all other participants execute `initBarrier`. Listing 2.6 rewrites the smoothing algorithm with a split-phase barrier. With split-phase barriers Line 5 can be run concurrently with Line 7.

Listing 2.6: Iterative averaging programmed with barrier synchronisation.

```
1 for (i=1;i<=N;i++) do par
2   for (k=1;k<=M;k++) do seq {
3     l = P[(i-1) % N]; r = P[(i+1) % N];
4     initBarrier;
5     tmp = (l + r)/2;
6     waitBarrier;
7     P[i] = tmp;
8     barrier;}
```

At the time, research is mostly geared towards the performance of the synchronisation algorithm [9, 22, 49]. Rajiv Gupta also introduces the notion of barrier synchronisation in a subset of tasks in the system, in contrast with a global barrier that affects all tasks.

In the 1980's, there are also some advances related to the fork/join programming model. The parallel functional languages Multilisp [48] and Id [11] include abstractions that mix communication with a barrier synchronisation. Multilisp introduces *futures*, or promises, that can be seen as a placeholder for the outcome of a function that is being computed concurrently, possibly in parallel. An arbitrary number of consumer tasks can be awaiting a result to be produced on the placeholder (the barrier). Once the function evaluates, the waiting tasks can resume their work and have access to the outcome of the function. The language Id proposes *I-structures* as a simplification of futures. An I-structure can also be seen as a placeholder for the outcome of a computation, yet, unlike futures, this

Listing 2.7: A deadlock using synchrons.

```
1  (let ((a (synchron)) (b (synchron)))
2    (par
3      (begin (wait a) (wait b))
4      (begin (wait b) (wait a))))
```

mechanism does not spawn any tasks. Tasks can observe and consume values written in an I-structure. Writing to the I-structure is synchronised with all the pending reads. Writing is a non-blocking operation, so the writer task does not wait for the readers tasks.

In the 1990's there is an ongoing exploration of (explicit) task parallelism, a continued work on the fork/join programming model, and barriers appear as first-class values and varying participation is introduced. Two notable languages based on the fork/join programming are announced: Cilk [41] (as a C extension) and OpenMP [36] (as a Fortran extension). As barriers make their appearance in more programming languages, their semantics become richer. This decade introduces barrier synchronisation where the group of participants varies over time. In 1990, varying participation appears first in hardware barrier synchronisation [81]. In 1996, Franklyn Turbak proposes synchrons [102]: the barrier abstraction is a first-class value that can be stored in any data structure. Furthermore, synchrons also allow for varying participation, the first time such property appears in software-based barriers. In Listing 2.7 two tasks wait for two synchrons in a alternative order, rendering them in a deadly embrace. A main task creates two synchrons, in line 1, and then uses primitive `par` to spawn two new tasks. One of the spawned tasks, in line 3, waits first on synchron `a` and then on synchron `b`. The other spawned task, in line 4, waits on `b` first and on synchron `a` second.

MPI [40], an extension of C or of Fortran, is announced in 1992. There is support for *collective operations* and the possibility to group tasks. Collective operations must be executed by every member of a group of tasks, introducing an implicit barrier at each operation. For example, if a task executes an `MPI_Broadcast` while another task executes a `MPI_Scatter`, then we have a deadlock caused by misaligned barriers. Additionally, any task that shares (transitively) a group with any of the deadlocked tasks also becomes deadlocked.

The 2000's give rise to a new family of parallel programming languages called PGAS (Partitioned Global Address Space) for task parallel languages with access to a hierarchic shared memory. Some languages that are part of this family include Chapel, Titanium, UPC, and X10. In 2001, Jung *et al.* promotes the split-phase barrier synchronisation to a first-class synchronisation mech-

anism [59], in contrast with Gupta's view of split-phase barrier as a compiler optimisation. MPI, UPC, and X10 offer split-phase barrier synchronisation. X10 includes a fork/join programming model and a barrier abstraction called *clock*, that is a first-class value and supports group synchronisation with a varying number of participants.

Only in the 2000's do mainstream languages start incorporating barrier synchronisation in their libraries. Java and the language family behind the .NET framework did not change their syntax to accommodate barrier synchronisation; all abstractions that perform barrier synchronisation are first-class values. Java 5.0, in 2005, include three abstractions that perform barrier synchronisation: latches, cyclic barriers, and futures. A latch performs one-shot barrier synchronisation for a fixed number of participants. The cyclic barrier performs reusable barrier synchronisation, also for a fixed number of participants. The .NET framework 4.0, in 2010, includes latches, a cyclic barrier that supports varying participation, futures, and a fork/join programming model.

HJ is a derivation of X10, so they share the programming model and most language constructs. A novelty of HJ is the proposal of phasers [94] to replace clocks. The semantic novelty in this abstraction is the way tasks can influence barrier synchronisation, which resembles latches and I-structures. A task can observe a phaser and just await participants, without others waiting for it. A task can cross the barrier (*i.e.,* arrive and proceed without waiting), yet others still need to wait for it to arrive at the barrier. A task can still use a phaser for regular reusable barrier synchronisation, by arriving and waiting. Phasers can be used to perform producer/consumer synchronisation, usually done with condition variables [53], whose deadlocks are known to be very difficult to handle [4, 57]. Later, there is a phaser extension to support bounded producer/consumer synchronisation patterns, called phaser beams [96]. Finally, in 2011, Java 7.0 adds an abstraction inspired by phasers, but that does not support observers; a Java phaser is essentially a clock but, confusingly, it is also called a phaser.

To summarise, the barrier properties we consider are:

Group synchronisation: A subset of the available tasks can synchronise together as a group. Examples: clocks, cyclic barriers, join barriers, latches, MPI collective operations, phasers, and synchrons.

Reuse: Participants may use the same abstraction to perform more than one barrier synchronisation. Examples: clocks, cyclic barriers, MPI/UPC collective operations, phasers, and synchrons.

Split-phase synchronisation: The synchronisation mechanism must be able to be commenced asynchronously. Examples: clocks, latches, MPI/UPC collective operations, and phasers.

Varying participation: A task can join and leave a group that is synchronised with a barrier. Examples: clocks, MPI collective operations, phasers, and synchrons.

Phase observing: A task can observe the barrier without influencing it. In the case of a reusable barrier, the task must be able to observe a specific phase. A task can arrive without needing to await any participant. Examples: futures, I-structures, join barriers, latches, and phasers.

2.2 Related work

The seminal work from Peter J. Landin [68] along with the programmer's manual of LISP 1.5 [79] pioneered the idea of reasoning in terms of families of programming languages, called *calculi*. The goal of a calculus is to unify multiple programming languages by abstracting mere syntactic variations. Examples of calculi include the λ-calculus [30] for functional programming languages, process algebras (*e.g.*, the π-calculus [80]) for concurrent languages, and the object calculus [1] for object-oriented languages.

Calculi that include barrier synchronisation are usually limited to a specific barrier idiom. SPMD languages usually have global collective operations, so a calculus that targets this family of languages only concerns with global barrier synchronisation. Similarly, a calculus that deals with futures, or with join barriers, only concerns with one-shot barrier synchronisation. Yet HJ, Java, the .NET framework, OpenMP, and X10 are just some examples of languages that comprise varied barrier idioms.

Deadlock prevention. The literature around source code analysis to prevent global barrier deadlocks is vast: MPI [76, 85, 97, 111], OpenMP [112], OpenSHMEM [88], and Split-C [7] (a predecessor of UPC). It is worth noting that MPI supports group barrier synchronisation, but works on deadlock prevention can only cope with global synchronisation.

The fork/join programming model is easily restricted syntactically to prevent deadlocks from happening. The λ_S-calculus by Arvind *et al.* [10] and the calculus by Aditya *et al.* [3] study the fork/join programming model in the context of functional programming languages. Lee and Palsberg presented a calculus for a fork/join programming model [70], suited for inter-procedural analysis through type inference, and establishes the deadlock freedom property. The work by Lee and Palsberg also includes a type system that is used to identify may-happen-parallelism, further explored by Agarwal *et al.* in [5].

There is some work surrounding the formalisation of barrier semantics with complex properties of barrier synchronisation, but do not establish deadlock-freedom. Saraswat and Jagadeesan formalise a subset of X10 that prevents deadlocks [92], comprising join barriers and clocks. Le *et al.* devise a verification for the correct use of a cyclic barrier in a fork/join programming language [69]. Vasudevan *et al.* have a similar approach on verifying the correct use of clocks [104].

The tool X10X [44] is a *model checker* for X10. Model checkers perform source code analysis and can be used to discover potential deadlocks. This class of tools suffers from the state explosion problem: the analysis grows exponentially with the possible interleaves of the program. Thus, X10X may not be able to verify complex programs.

There is a research opportunity on formal techniques that prevent general barrier deadlocks.

Deadlock avoidance and detection. To our best knowledge, techniques that avoid deadlocks in the context of barrier synchronisation are incomplete, *i.e.,* only handle a few situations of barrier deadlocks. For instance, in X10 and HJ, tasks deregister from all barriers upon termination; this mitigates deadlocks that arise from missing participants. HJ avoids deadlocks that originate from the interaction between phasers and finish blocks by limiting the use of phasers to the scope of finish blocks. Deadlock detection tools for Titanium [60] and UPC-CHECK [91] can only handle global barrier synchronisation. Literature concerning MPI deadlock detection is still not general enough for languages like Java and X10 and lacks formal specifications. DAMPI [105], Marmot [64], and MPI-CHECK [73] report a programs as deadlocked after a period of inactivity, so it can misidentify a slow program as a being deadlocked. Umpire [51] and MUST [52] (a successor of Umpire) use a graph-based deadlock detection algorithm, but omit a formal description on how the graph is actually generated from the language. Furthermore, MUST is incapable of verifying split-phase synchronisation, known in MPI as non-blocking collective operations.

There is a research opportunity on deadlock avoidance and detection for general barrier synchronisation.

Chapter Three

Brenner: a calculus for parallel programming

We present phasers and a core-language to reason about task parallelism with this abstraction. The following section revisits some examples to introduce the primitives that comprise a phaser. Section 3.1 presents the syntax of Brenner. We discuss the operational semantics in Section 3.2.

The definitions and examples in this chapter are mechanised in Coq [78] and available online[1].

3.1 Syntax

A phaser is used to count and observe events generated by a group of tasks, similarly to a collective event counter [90]. The primitives we introduce distil the semantics *cf.* [94, 96]. Each participant is registered with an event counter, called a *local phase*, that is a non-decreasing, non-negative integer. Instruction `adv` increments the local phase of the issuing task. Instruction `await`(p, n) blocks until all members of phaser p reach phase n, *i.e.,* their local phase is at least n. Instruction `newPhaser` creates a phaser.

Tasks are referred by task names. Instruction `newTid` creates a task name. To dynamically create and launch a named task there is instruction `fork`. The members of a phaser are controlled with `reg` to add (register) a participant to a phaser, and `dereg` to remove (deregister) a participant from a phaser. Data transfers and data-related computation are abstracted and in their place we use instruction `skip`. Similarly, we represented structured control flow instructions, like for-loops and conditionals, with instruction `loop` that unfolds its body an arbitrary number of times.

Join barriers Listing 3.1 describes an one-shot barrier synchronisation as seen in the fork/join programming model. Our example revisits Listing 2.3, matrix

[1] `https://bitbucket.org/cogumbreiro/brenner-coq/`

Listing 3.1: Matrix multiplication programmed with a phaser.

```
1  p = newPhaser();
2  loop( // for (i = 0; i < 21; i++) {
3    t = newTid();
4    reg(t, p);
5    fork(t,
6      loop( // for (j = 0; j < 21; i++)
7        loop( // for (k = 0; k < 21; k++)
8          skip; // C[i][j] += A[i][k] * B[k][j];
9        end); // inner loop
10     end); // outer loop
11     adv(p); // signal termination
12   end); // fork
13 end); // loop
14 adv(p);
15 await(p, 1); // join
16 end
```

multiplication programmed with a task processing each row of the matrix. A driver task executes the code in Listing 3.1; it is responsible for forking the worker tasks processing the rows, and for joining their execution with a phaser p. In detail, the driver tasks creates phaser p, in Line 1, with instruction `newPhaser`, automatically registering the driver at phase 0. The driver uses `reg` to register `t` with p (Line 4); the registered task will inherit the phase of their registrant, in this case it is phase 0. The workers (Lines 6 to 12) advance their phase, in Line 11, to notify the driver that awaits their terminus, in Line 15. Since the driver is also registered with p, it advances its local phase before awaiting phase 1, in Line 14, otherwise it deadlocks all tasks.

Cyclic barriers Listing 3.2 revisits the split-phase synchronisation example seen in Listing 2.6. To encode a cyclic barrier, every participant advances its phase and then awaits at its local phase in Line 13, so that all members await each other. There are two variants of instruction await, when a task omits the phase number, `await`(p), then this task awaits at its local phase. Split-phase synchronisation commences with a phase advance in Line 8, and terminates with an await in Line 10.

Pipeline parallelism Phasers enable distinct synchronisation patterns when compared to other barrier-based abstractions. A case in point is the producer-

Listing 3.2: Split-phase synchronisation with a phaser.

```
1  p = newPhaser(); // c = new Clock();
2  loop( // for (i = 0; i < N; i++)
3    t = newTid();
4    reg(t, p);
5    fork(t, // async clocked(p)
6      loop(   // for (k=1; k <= M; k++)
7        skip;        // l=P[(i-1) % N];r=P[(i+1) % N];
8        adv(p);      // c.resume();
9        skip;        // tmp = (l + r) / 2;
10       await(p);    // c.advance();
11       skip;        // P[i] = tmp;
12       adv(p);
13       await(p);    // c.advance();
14     end); // for
15   end); // task
16 end); // outer loop
17 dereg(p); // revoke participation
18 end // program
```

consumer synchronisation pattern, sketched in Listing 3.3. Two groups of tasks, the producers and the consumers, synchronise their execution with a phaser p. Producer tasks only advance the phaser, while consumer tasks await consecutive phases of that phaser. Cyclic barriers cannot be used effectively to describe the producer-consumer pattern: since all participants of a cyclic barrier must wait for each other, then the execution of producers is constrained by the execution of consumers, which does not happen in Listing 3.3.

Pipeline parallelism is a parallel programming model based on the producer-consumer synchronisation pattern. In this programming model, computation is divided in stages that can run concurrently, where barrier synchronisation coordinates the execution order of different stages. Recent proposals of pipeline parallelism in the context of parallel programming languages include: OpenStream [87] for OpenMP, StreamX10 [107] and clocked variables [12] for X10, and phaser beams [96] for HJ.

Listing 3.3: Producer-consumer synchronisation with phasers.

```
1  p = newPhaser(); // c = new Phaser();
2  loop( // producers
3    t1 = newTid(); reg(t1, p); // producer
4    fork(t1,
5      loop( // for (i = 0; i < N; i++)
6        skip;       // B[i] = produce(i);
7        adv(p);     // signal consumer
8      end); // loop
9    end); // t1
10 end);
11 loop( // consumers
12   t2 = newTid(); reg(t2, p); // consumer
13   fork(t2,
14     loop( // for (i = 0; i < N; i++)
15       adv(p); await(p);
16       skip;       // consume(B[i]);
17     end); // loop
18   end); // t2
19 end);
20 dereg(p);
21 end // program
```

Syntax We propose the core language BRENNER[2] to reason about task parallelism with phasers. The language itself is very basic—not even Turing-complete!—but provides a sufficient programming model to reason about the barrier abstractions surveyed in Chapter 2. We abstain from adding constructs unrelated to synchronisation, like data types, since such additions only complicates the semantics without bringing into play any novelty.

Definition 3.1.1 (Language syntax)**.** *The grammar in Fig. 3.1 defines our language.*

The grammar specifies how to construct a program in BRENNER in an abstract syntax based *cf.* [86]. A *term* can be elementary or composed of other terms. A grammar defines *categories* (*i.e.,* sets) of terms. The set of all programs is

[2] Originating from the Star Trek television series, the minor character Brenner [8] is referred as a phaser specialist on the script for the episode "Balance of Terror". Brenner serves under the command of Captain James T. Kirk and is responsible for coordinating and maintaining the phaser weapons of the USS Enterprise.

b	::=	*Programs*
	| `end`	end program
	| $i; b$	construct program
i	::=	*Instructions*
	| $t =$ `newTid`$()$	new task identifier
	| `fork`(t, b)	spawns the execution of a task
	| $p =$ `newPhaser`$()$	create a phaser
	| `reg`(t, p)	register task with phaser
	| `dereg`(p)	deregister current task from phaser
	| `adv`(p)	advance local phase
	| `await`(p, n)	await for phase n
	| `await`(p)	await current phase
	| c	control the flow
c	::=	*Control flow*
	| `skip`	internal action
	| `loop`(b)	non-deterministic loop

Figure 3.1: Top-level syntax.

an example of a category of terms. In Brenner, a program is composed by instructions, which are themselves other terms. Notation ::= declares a term category: in the left-hand side there is a *meta-variable* (a letter) that ranges over the terms of that category; in the right-hand side the declaration of the alternative terms, separated by a vertical bar |, that reads as "or."

The grammar of Brenner consists of two categories of terms: programs ranged over by b, and instructions ranged over by i. The definition of a program b has two possible terms: it is either (i) an elementary term `end`, or (ii) a construct program term that is composed of an instruction i followed by the continuation program b. The alternatives in the right-hand side of ::= work as templates, so any meta-variable that appears in the right-hand side of ::= does not represent a specific instance, but a placeholder for a term of that category. For instance, term b that appears in $i; b$ represents a placeholder for any program term that can be constructed using the grammar Fig. 3.1.

The grammar relies on a base set of phaser names $\mathcal{P}$, ranged over by p and by q; a base set of task names $\mathcal{T}$, ranged over by t; and a set $\mathcal{N}$ of natural numbers, ranged over by n and by m.

$$
\begin{array}{rcll}
S & ::= & \big(M, T\big) & \textit{State} \\
M & ::= & \{p_1 \colon P_1, \ldots, p_n \colon P_n\} & \textit{Phaser maps} \\
T & ::= & \{t_1 \colon b_1, \ldots, t_n \colon b_n\} & \textit{Task maps} \\
P & ::= & \{t_1 \colon n_1, \ldots, t_m \colon n_m\} & \textit{Phaser value} \\
b & ::= & \cdots \mid \texttt{idle} &
\end{array}
$$

Figure 3.2: Syntax of the abstract machine.

3.2 Operational Semantics

The formalism that specifies the meaning of BRENNER is known as *operational semantics*, and it describes how computation develops. Operational semantics can be divided into two categories: *small-step* semantics that describes the individual steps of computation, and *big-step* semantics that describes how the overall results are obtained (*i.e.,* given an input state, what is the final outcome state). Concurrent languages are usually specified in small-step semantics since big-step semantics "hides" the intermediate steps that lead to a result. The gist of small-step operational semantics is to define (i) the state of an abstract machine (or abstract computer), and (ii) the effects of each possible action on a given state. A grammar specifies the state of an abstract machine. A (binary) reduction relation ($\rightarrow$) defines (ii), by relating the state of the machine before execution with the state of the machine after execution of a single indivisible action.

Definition 3.2.1 (Abstract machine)**.** *Fig. 3.2 depicts the syntax of the abstract machine.*

An abstract machine has a state S that pairs a phaser map M with a task map T. The phaser map M stores the available phasers, mapping addresses to phasers. A phaser P maps task identifiers to naturals. The task map T holds programs b, labelled by task names t. We extend the syntax of programs, by adding the runtime-only instruction `idle`, to represent a task that is ready to be started (a side effect of instruction `newTid`).

The following function loads a program into the abstract machine. We use notation $\stackrel{\text{def}}{=}$ for the definition of functions and constants. The initial state consists of an empty phaser map and a single task t_d. The program is loaded into task t_d, which commences without being registered with any phaser.

Definition 3.2.2 (Load function)**.**

$$\text{load}(b) \stackrel{\text{def}}{=} \big(\emptyset, \{t_d \colon b\}\big)$$

$$\big(M, T \uplus \{t\colon t' = \texttt{newTid}(); b\}\big) \to \big(M, T \uplus \{t\colon b\} \uplus \{t'\colon \texttt{idle}\}\big) \quad \text{(R-NEW-T)}$$

$$\big(M, T \uplus \{t\colon \texttt{fork}(t', b'); b\} \uplus \{t'\colon \texttt{idle}\}\big) \to \big(M, T \uplus \{t\colon b\} \uplus \{t'\colon b'\}\big) \quad \text{(R-FORK)}$$

$$\frac{c; b \to b'}{\big(M, T \uplus \{t\colon (B, c; b)\}\big) \to \big(M, T \uplus \{t\colon (B, b')\}\big)} \quad \text{(R-FLOW)}$$

Figure 3.3: The small-step semantics of BRENNER (tasks).

Let b be the program in Listing 3.2. An abstract machine running b has an initial state load(b), defined as follows.

$$\big(\emptyset, \big\{t_d\colon p = \texttt{newPhaser}(); \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end}\big\}\big) \tag{3.1}$$

Definition 3.2.3 (Domain, empty map, and update.)**.** *Given a map, we write* dom M *for the domain of map* M*. We use notation* $\emptyset$ *for the empty map, such that no element is in its domain. When* x *is not in the domain of map* M_1*, we write* $M_1 \uplus \{x\colon y\}$ *for map* M_2 *such that* $\text{dom}\, M_2 = \text{dom}\, M_1 \cup \{x\}$*,* $M_2(x) = y$*, and* $M_2(z) = M_1(z)$ *for all* $z \in \text{dom}\, M_1$*.*

The reduction relation ($\to$) for BRENNER is defined by a set of *inference rules* in Figs. 3.3 to 3.5. The rules are syntax-oriented, which means that there is only one rule per instruction i, *e.g.*, rule R-NEW-P describes the behaviour of instruction `newPhaser`.

An inference rule defines a conclusion C that follows from some premises $P_1, P_2, \ldots, P_n$. The general notation of an inference rule is

$$\frac{P_1 \qquad P_2 \qquad \cdots \qquad P_n}{C}$$

considering that symbol $\wedge$ is the logical conjunction and symbol $\implies$ is the logical implication the above notation is equivalent to

$$P_1 \wedge P_2 \wedge \cdots \wedge P_n \implies C$$

When there are no premises ($n = 0$), the rule is called an *axiom* and we omit the over bar, as in rule R-FORK.

Given a reduction $S \to S'$, state S is an input parameter and state S' an output parameter. A system of inference rules, such as an operational semantics, matches any input parameters and infers, or produces, any output parameters. Henceforth, we say task t (phaser p) as short for the task (phaser) labelled by t.

$$\big(M, T \uplus \{t\colon p = \texttt{newPhaser}(); b\}\big) \to \big(M \uplus \big\{p\colon \{t\colon 0\}\big\}, T \uplus \{t\colon b\}\big) \tag{R-NEW-P}$$

$$\frac{T(t') = \texttt{idle} \qquad P(t) = n \qquad P' = P \uplus \{t'\colon n\}}{\big(M \uplus \{p\colon P\}, T \uplus \{t\colon \texttt{reg}(t', p); b\}\big) \to \big(M \uplus \{p\colon P'\}, T \uplus \{t\colon b\}\big)} \tag{R-REG}$$

$$\big(M \uplus \big\{p\colon P \uplus \{t\colon n\}\big\}, T \uplus \{t\colon \texttt{dereg}(p); b\}\big) \to \big(M \uplus \{p\colon P\}, T \uplus \{t\colon b\}\big) \tag{R-DEREG}$$

$$\begin{aligned}&\big(M \uplus \big\{p\colon P \uplus \{t\colon n\}\big\}, T \uplus \{t\colon \texttt{adv}(p); b\}\big)\\ \to&\big(M \uplus \big\{p\colon P \uplus \{t\colon n+1\}\big\}, T \uplus \{t\colon b\}\big)\end{aligned} \tag{R-ADV}$$

$$\frac{M(p) = P \qquad \forall t' \in \operatorname{dom} P\colon P(t') \geq n}{\big(M, T \uplus \{t\colon \texttt{await}(p, n); b\}\big) \to \big(M, T \uplus \{t\colon b\}\big)} \tag{R-SYNC}$$

$$\frac{M(p)(t) = n}{\big(M, T \uplus \{t\colon \texttt{await}(p); b\}\big) \to \big(M, T \uplus \{t\colon \texttt{await}(p, n); b\}\big)} \tag{R-AWAIT}$$

Figure 3.4: The small-step semantics of BRENNER (phasers).

Recall the initial state of program b, defined in Formula 3.1, and let it be state S_1.

$$\big(\emptyset, \big\{t_d\colon p = \texttt{newPhaser}(); t_1 = \texttt{newTid}(); \texttt{reg}(t_1, p); \texttt{fork}(t_1, b_1); b_d\big\}\big)$$

If we can construct a state S_2 that is in the reduction relation with S_1

$$S_1 \to S_2$$

then we say that state S_1 reduces to state S_2. Yet, not all states reduce. In particular, since all reduction rules expect a task map with at least one task, state $\big(\emptyset, \emptyset\big)$ does not reduce.

To check that $S_1 \to S_2$ holds, we match the syntax of S_1 with every reduction rule. Multiple rules may match the syntax, so it is possible to have more than one state S_2 that is in relation with S_1—in fact, that is how we encode concurrency!

The inference rules have implicit universal quantification on every meta-variable that appears in an input parameter. In the case of rule R-NEW-P there is an implicit $\forall M, T, t, p, b$:

$$\big(M, T \uplus \{t\colon p = \texttt{newPhaser}(); b\}\big) \to \big(M \uplus \big\{p\colon \{t\colon 0\}\big\}, T \uplus \{t\colon b\}\big)$$

$$\texttt{skip}; b \to b \qquad \text{(R-SKIP)}$$
$$\texttt{loop}(b); b' \to b \cdot (\texttt{loop}(b); b') \qquad \text{(R-ITER)}$$
$$\texttt{loop}(b); b' \to b' \qquad \text{(R-ELIDE)}$$

Figure 3.5: Small step semantics for control flow instructions $\boxed{c; b \to b}$.

Hence, applying rule R-NEW-P to $S_1 \to S_2$ yields

$$\begin{aligned}&\big(\emptyset, \{t_d\colon p = \texttt{newPhaser}(); \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end}\}\big)\\ \to&\big(\{p\colon \{t_d\colon 0\}\}, \{t_d\colon \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end}\}\big)\end{aligned}$$

with $M \stackrel{\text{def}}{=} \emptyset$, $T \stackrel{\text{def}}{=} \emptyset$, $t \stackrel{\text{def}}{=} t_d$, $p \stackrel{\text{def}}{=} p$, and $b \stackrel{\text{def}}{=} \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end}$. Rule R-NEW-P allocates a new phaser, with a single registered task (the creator of the phaser).

Let S_3 be such that relation $S_2 \to S_3$. The relation holds with rule R-FLOW, but we must show that the control flow instruction $\texttt{loop}\ b_l$ reduces. Reduction for control flow instructions is defined in Fig. 3.5. Program concatenation is defined as expected.

Definition 3.2.4 (Sequence concatenation)**.**

$$(i; b) \cdot b' \stackrel{\text{def}}{=} i; (b \cdot b')$$
$$\texttt{end} \cdot b \stackrel{\text{def}}{=} b$$

Applying rule R-ITER yields

$$\texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end} \to t_1 = \texttt{newTid}(); \texttt{reg}(t_1, p); \texttt{fork}(t_1, b_f); b_d$$

where

$$b_l \cdot \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end} \stackrel{\text{def}}{=} t_1 = \texttt{newTid}(); \texttt{reg}(t_1, p); \texttt{fork}(t_1, b_f); b_d$$

Hence, with rule R-FLOW we have that $S_2 \to S_3$.

$$\begin{aligned}&\big(\{p\colon \{t_d\colon 0\}\}, \{t_d\colon \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end}\}\big)\\ \to&\big(\{p\colon \{t_d\colon 0\}\}, \{t_d\colon t_1 = \texttt{newTid}(); \texttt{reg}(t_1, p); \texttt{fork}(t_1, b_f); b_d\}\big)\end{aligned}$$

Let S_4 be such that relation $S_3 \to S_4$ holds. With rule R-NEW-T we get the following formula

$$\begin{aligned}&\big(\{p\colon \{t_d\colon 0\}\}, \{t_d\colon t_1 = \texttt{newTid}(); \texttt{reg}(t_1, p); \texttt{fork}(t_1, b_f); b_d\}\big)\\ \to&\big(\{p\colon \{t_d\colon 0\}\}, \{t_d\colon \texttt{reg}(t_1, p); \texttt{fork}(t_1, b_f); b_d, t_1\colon \texttt{idle}\}\big)\end{aligned}$$

Rule R-NEW-T extends the task map with an idle task t_1. At the syntax level, `newTid` only displays a task name, but at the operational semantics level the instruction creates a special "idle" task to ensure that there are no task-name clashes. An idle task becomes a running one with a fork.

Let S_5 be such that relation $S_4 \rightarrow S_5$ holds. With rule R-REG we get the following formula

$$\begin{aligned}&\big(\{p\colon \{t_d\colon 0\}\}, \{t_d\colon \texttt{reg}(t_1, p); \texttt{fork}(t_1, b_f); b_d, t_1\colon \texttt{idle}\}\big)\\ \rightarrow&\big(\{p\colon \{t_d\colon 0, t_1\colon 0\}\}, \{t_d\colon \texttt{fork}(t_1, b_f); b_d, t_1\colon \texttt{idle}\}\big)\end{aligned}$$

Rule R-REG extends the phaser addressed by p with the new participant t_1. The local phase of the registered task is inherited from the registrant, so in this case both are at the local phase 0. The configuration of the phaser map indicates that the task invoking R-REG is registered with the phaser.

Let S_5 be such that relation $S_4 \rightarrow S_5$ holds. With rule R-FORK we get the following formula

$$\begin{aligned}&\big(\{p\colon \{t_d\colon 0, t_1\colon 0\}\}, \{t_d\colon \texttt{fork}(t_1, b_f); b_d, t_1\colon \texttt{idle}\}\big)\\ \rightarrow&\big(\{p\colon \{t_d\colon 0, t_1\colon 0\}\}, \{t_d\colon b_d, t_1\colon b_f\}\big)\end{aligned}$$

Rule R-FORK simply replaces the body of idle task t_1 with the parameter of `fork`, program b_f. The parameter of `idle` identifies the creator of the task name. The syntactic restriction in the rule ensures that only task t_d can fork a task named t_1.

Recall that

$$b_d \stackrel{\text{def}}{=} \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end}$$

and

$$b_f \stackrel{\text{def}}{=} \texttt{adv}(p); \texttt{await}(p); \texttt{adv}(p); \texttt{await}(p); \texttt{end}$$

There are two possible reductions for state S_5, one uses rule R-FLOW, another uses rule R-ADVANCE. Such nondeterminism represents the concurrency present in parallel execution. We continue reducing with task t_d to conclude the discussion of task membership. We place the rule next to the reduction operator

to help the reader.

$$
\begin{aligned}
&\big(\{p\colon \{t_d\colon 0, t_1\colon 0\}\},\\
&\ \{t_d\colon \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end}, t_1\colon b_f\}\big)\\
\text{R-FLOW} \rightarrow\ &\big(\{p\colon \{t_d\colon 0, t_1\colon 0\}\},\\
&\ \{t_d\colon t_2 = \texttt{newTid}(); \texttt{reg}(t_2, p); \texttt{fork}(t_2, b_f); b_d, t_1\colon b_f\}\big)\\
\text{R-NEW-T} \rightarrow\ &\big(\{p\colon \{t_d\colon 0, t_1\colon 0\}\},\\
&\ \{t_d\colon \texttt{reg}(t_2, p); \texttt{fork}(t_2, b_f); b_d, t_1\colon b_f, t_2\colon \texttt{idle}\}\big)\\
\text{R-REG} \rightarrow\ &\big(\{p\colon \{t_d\colon 0, t_1\colon 0, t_2\colon 0\}\},\\
&\ \{t_d\colon \texttt{fork}(t_2, b_f); b_d, t_1\colon b_f, t_2\colon \texttt{idle}\}\big)\\
\text{R-FORK} \rightarrow\ &\big(\{p\colon \{t_d\colon 0, t_1\colon 0, t_2\colon 0\}\},\\
&\ \{t_d\colon \texttt{loop}(b_l); \texttt{dereg}(p); \texttt{end}, t_1\colon b_f, t_2\colon b_f\}\big)\\
\text{R-FLOW} \rightarrow\ &\big(\{p\colon \{t_d\colon 0, t_1\colon 0, t_2\colon 0\}\},\\
&\ \{t_d\colon \texttt{dereg}(p); \texttt{end}, t_1\colon b_f, t_2\colon b_f\}\big)
\end{aligned}
$$

Let $S_6 \rightarrow S_7$ hold With rule R-DEREG.

$$
\begin{aligned}
&\big(\{p\colon \{t_d\colon 0, t_1\colon 0, t_2\colon 0\}\}, \{t_d\colon \texttt{dereg}(p); \texttt{end}, t_1\colon b_f, t_2\colon b_f\}\big)\\
\rightarrow\ &\big(\{p\colon \{t_1\colon 0, t_2\colon 0\}\}, \{t_d\colon \texttt{end}, t_1\colon b_f, t_2\colon b_f\}\big)
\end{aligned}
$$

Rule R-DEREG removes the issuer task t_d from the phaser addressed by p. The syntactic configuration of the phaser after reduction indicates that t_d revoked its membership with phaser p.

We proceed by reducing state S_7. Let $b_2 \stackrel{\text{def}}{=} \texttt{adv}(p); \texttt{await}(p); \texttt{end}$. At this point we can reduce term with either task t_1 or task t_2. We choose to reduce with task t_1. With rule R-ADVANCE and then with R-AWAIT we get the following formula.

$$
\begin{aligned}
&\big(\{p\colon \{t_1\colon 0, t_2\colon 0\}\}, \{t_d\colon \texttt{end}, t_1\colon \texttt{adv}(p); \texttt{await}(p); b_2, t_2\colon b_f\}\big)\\
\rightarrow\ &\big(\{p\colon \{t_1\colon 1, t_2\colon 0\}\},\\
&\ \{t_d\colon \texttt{end}, t_1\colon \texttt{await}(p); b_2, t_2\colon \texttt{adv}(p); \texttt{await}(p); b_2\}\big)\\
\rightarrow\ &\big(\{p\colon \{t_1\colon 1, t_2\colon 0\}\},\\
&\ \{t_d\colon \texttt{end}, t_1\colon \texttt{await}(p, 1); b_2, t_2\colon \texttt{adv}(p); \texttt{await}(p); b_2\}\big)
\end{aligned}
$$

Rule R-ADVANCE increments the local phase of the registered task t_1, enforced by the syntactic structure of the phaser. Rule R-AWAIT rewrites the `await` by making the wait explicit at the local phase of task t_1.

The only rule that can be applied to the state above is R-ADVANCE, which executes task t_2. Task t_1 must wait for task t_2 to advance its phase to 1. With rule R-ADVANCE we get the next formula.

$$\begin{aligned}&\big(\{p\colon \{t_1\colon 1, t_2\colon 0\}\},\\ &\quad \{t_d\colon \texttt{end}, t_1\colon \texttt{await}(p,1); b_2, t_2\colon \texttt{adv}(p); \texttt{await}(p); b_2\}\big)\\ \to &\big(\{p\colon \{t_1\colon 1, t_2\colon 1\}\}, \{t_d\colon \texttt{end}, t_1\colon \texttt{await}(p,1); b_2, t_2\colon \texttt{await}(p); b_2\}\big)\end{aligned}$$

Synchronisation happens with rule R-SYNC.

$$\begin{aligned}&\big(\{p\colon \{t_1\colon 1, t_2\colon 1\}\}, \{t_d\colon \texttt{end}, t_1\colon \texttt{await}(p,1); b_2, t_2\colon \texttt{await}(p); b_2\}\big)\\ \to &\big(\{p\colon \{t_1\colon 1, t_2\colon 1\}\}, \{t_d\colon \texttt{end}, t_1\colon b_2, t_2\colon \texttt{await}(p); b_2\}\big)\end{aligned}$$

Rule R-SYNC consumes the `await` when its premise is enabled, by checking that every registered task is at least at phase 1.

Coq mechanisation The definitions and examples of this chapter are all formalised in Coq[3]. The interested reader can exercise our definitions and alter the examples in this section.

[3] `https://bitbucket.org/cogumbreiro/brenner-coq/`

Chapter Four

Runtime deadlock verification

The purpose of a runtime deadlock verification tool is to continuously check whether the concurrency constraints of the running tasks are unsatisfiable, in which case there is a deadlock. Runtime verification tools obtain concurrency constraints from concurrency dependencies among tasks and blocking operations, *e.g.*, task A waits in a phaser p for tasks B and C, or task A impedes tasks B and C from synchronising with phaser q. Graph-based techniques check the unsatisfiability of concurrency constraints by analysing a graph of concurrency dependencies. We propose and implement a graph-based technique that performs cycle detection to check for deadlocks.

The following section proposes an intermediate general abstraction called a resource-dependency state to capture the relationship between tasks and resources, and defines a translation from a Brenner state S to a resource-dependency state. Section 4.2 discusses some necessary graph-theoretical notions. Section 4.3 describes two graph models that can be extracted from a resource-dependency state. Section 4.4 that puts forward two important results: (i) cycle detection is equivalent in the two graph models, and (ii) the deadlock verification is sound and complete, against a Brenner program. Section 4.5 presents Armus, a deadlock verification tool capable of fault-tolerant and distributed detection, and also of deadlock avoidance. Armus to performs a novel graph model selection to dramatically improve the performance of deadlock verification. The chapter closes with an evaluation of the performance of Armus in local and distributed settings.

4.1 Resource dependencies

State-of-the-art runtime verification tools gather concurrency dependencies between tasks and barriers by monitoring the status of blocked tasks and by bookkeeping the participants of each barrier. Tracking the latter poses a problem to distributed verification, as the information about the participants of a barrier can be distributed among various computation nodes [6, 50]. Instead, we propose

gathering dependencies among timestamps, in the sense of Lamport's logical clocks [67], as it improves the performance of our verification algorithm.

A logical clock orders events by associating a different timestamp (a monotonic integer) per event. We consider a phaser to be a logical clock, and a phase to be a timestamp. When tasks synchronise on a phase number n of a phaser p each participant observes a synchronisation event that occurred at timestamp n of the logical clock associated with phaser p. Under this view, blocked tasks *wait* for a specific event to be observed. But since a waiting task cannot arrive at other registered phasers, then waiting tasks also *impede* the observation of events. Thus, any event a task awaits *precedes* all events that this task impedes. A deadlock corresponds to any circular dependencies found in such ordering of events.

Resource-dependency states A resource-dependency state D describes the relationship between tasks $t \in \mathcal{T}$ and resources[1] $r \in \mathcal{R}$. Let $\wp$ be the power set function. Let $W \colon \mathcal{T} \mapsto \wp(\mathcal{R})$ be a function from tasks into sets of resources. The set $W(t)$ contains the resources that task t is blocked on. In the case of Brenner, tasks can be blocked at most on one phaser at a time.

Tasks can impede the synchronisation of another task. Let $I \colon \mathcal{R} \mapsto \wp(\mathcal{T})$ be a function from barriers into sets of tasks. The set $I(r)$ contains the tasks that impede the synchronisation of any task using resource r, *e.g.*, the set of tasks that remain to arrive at a barrier.

Definition 4.1.1 (Resource-dependency)**.** *A resource-dependency D consists of a pair (I, W).*

For example, consider the deadlocked state $\big(M_1, T_1\big)$ defined below, where tasks t_1, t_2, and t_3 wait on a phaser p at phase 2 for task t_4, which waits on a phaser q at phase 3 for tasks t_1, t_2, and t_3.

$$\begin{aligned} M_1 &= \big\{p \colon \{t_1 \colon 2, t_4 \colon 1\}, q \colon \{t_1 \colon 1, t_2 \colon 2, t_3 \colon 1\}\big\}, \\ T_1 &= \{t_1 \colon \texttt{await}(p, 2); b_1, t_2 \colon \texttt{await}(p, 2); b_2, \\ &\qquad t_3 \colon \texttt{await}(p, 2); b_3, t_4 \colon \texttt{await}(q, 3); b_4, \} \end{aligned}$$

To construct a resource-dependency (I_1, W_1) from $\big(M_1, T_1\big)$ we look into any task awaiting on a phaser to identify a resource. Let resource r_1 represent awaiting on phaser p at phase 2 and resource r_2 represent awaiting on phaser q at phase 3. Hence, $W_1 = \big\{t_1 \colon \{r_1\}, t_2 \colon \{r_1\}, t_3 \colon \{r_1\}, t_4 \colon \{r_2\}\big\}$. To construct the structure of impeding tasks we inspect the phaser map. Resource r_1 (phaser p

[1] We use term "resource" to be consistent with the accompanying literature [58, 62]. A better suiting term in our context would be "event."

at phase 2) is an impediment (for synchronisation) because of task t_4 and not because of t_1 (whose phase is at least 2). Similarly, resource r_2 (phase q at phase 3) is an impediment because of tasks t_1, t_2, and t_3, since all are registered with a phase below 3. Thus,

$$I_1 = \{r_1 \colon \{t_4\}, r_2 \colon \{t_1, t_2, t_3\}\}$$

We put forward this notion in the following definition. Let γ be a bijection that maps pairs of phaser names and naturals (the phase) to resources.

Definition 4.1.2 (Resource-dependency construction)**.** *Let ψ be a function from states into resource-dependencies.*

$$\begin{aligned}
\psi\left(M, T\right) &\stackrel{\text{def}}{=} (I, W) \\
I &\stackrel{\text{def}}{=} \left\{r \colon \{t \mid M(p)(t) < n\} \;\middle|\; T(t') = \texttt{await}(p, n); _ \textit{ and } \gamma(p, n) = r\right\} \\
W &\stackrel{\text{def}}{=} \left\{t \colon \{r\} \;\middle|\; T(t) = \texttt{await}(p, n); _ \textit{ and } \gamma(p, n) = r\right\}
\end{aligned}$$

By Definition 4.1.2 we have that $\psi\left(M_1, T_1\right) = (I_1, W_1)$.

4.2 Basic graph theory

Following are some graph theory concepts based on [15].

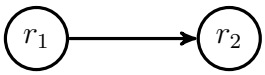
Definition 4.2.1 (Graph, vertex, and edge.)**.** *A (directed)* graph $G = (V, E)$ *consists of a nonempty finite set of vertices V (where $r \in V$), and of a finite set of edges E (where $e \in E$). An* edge $e = (r, r')$ *directs from the head r to the tail r'.*

For instance, the following graph has two vertices, r_1 and r_2, and two edges, edge (r_1, r_2) from r_1 to r_2 and edge (r_2, r_1) from r_2 to r_1.

$$G_1 = \left(\{r_1, r_2\}, \{(r_1, r_2), (r_2, r_1)\}\right)$$

For the graphical notation we can construct the depiction in two steps. The first step is to depict vertexes, by drawing a circle around each vertex. In the case of G_1, we get the following illustration.

The second step is to depict edges, by drawing an arrow directed from the circle representing the head to the circle representing the tail. For instance, an edge (r_1, r_2) yields the next depiction.



The graphical representation of G_1 follows.

Definition 4.2.2 (Subgraph relation)**.** *Graph (V, E) is a* subgraph *of graph (V', E') if (i) $V \subset V'$, (ii) $E \subset E'$, and (iii) $\forall (r, r') \in E \implies r \in V \wedge r' \in V$.*

For example, graph G_1 is a subgraph of

$$\big(\{r_1, r_2, r_3\}, \{(r_1, r_2), (r_2, r_1), (r_1, r_3)\}\big)$$

Definition 4.2.3 (Walk, cycle, and length.)**.** *A* walk *w on graph (V, E) is an alternating sequence $r_1 r_2 \cdots r_{n-1} r_n$ of vertices $r_i \in V$ such that $n > 1$ and $(r_i, r_{i+1}) \in E$ for every $i = 1, 2, \ldots, n-1$. We may specify the first and last vertices of a walk by saying a r-r' walk, for the walk $r \cdots r'$. A* cycle *is a walk $r \cdots r'$ where $r = r'$. We may specify the first and last vertex of a cycle by saying a r-cycle, for the cycle $r \cdots r$. The* length *of a walk corresponds to the number of its edges. We say that $r \in w$ if, and only if, $w = r_1 \cdots r_n$ and there exists a r_i such that $r = r_i$ and $1 \leq i \leq n$. We say that $(r, r') \in w$ if, and only if, $w = r_1 \cdots r_n$ and there exists a r_i and r_{i+1} such that $r = r_i$, $r' = r_{i+1}$, and $1 \leq i < n$.*

An example of a walk on G_1 is $w_1 = r_1 r_2 r_1 r_2 r_1$. Walk w_1 is a cycle with length 4. We have that vertex $r_1 \in w_1$ and edge $(r_1, r_2) \in w_1$. Note that, by definition, any cycle has a positive length.

Definition 4.2.4 (In-degree and out-degree)**.** *The in-degree n of a vertex r counts the number of edges whose tail is r. The out-degree n of a vertex r counts the number of edges whose head is r.*

Definition 4.2.5 (Reachable)**.** *We say that vertex r' is* reachable *from r, or vertex r* reaches *r', if there exists a r-r' walk on graph G.*

4.3 Graph-based deadlock identification

Graph-based approaches perform cycle detection on the concurrency dependencies between tasks and synchronisation events. The Wait-For Graph [63] (WFG) only models dependencies between tasks. The State Graph [58] (SG) only models dependencies between synchronisation events. Since the performance of cycle

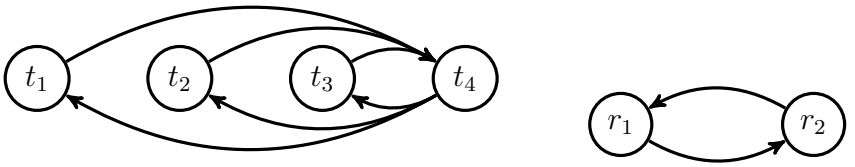
detection depends on the size of the graph, the ratio between the number of synchronisation events and the number of tasks impacts the graph model choice. We discuss three scenarios of applications that use barrier synchronisation.

Parallel applications designed following the Single Program Multiple Data (SPMD) programming model share two characteristics: there is a fixed number of tasks and a fixed number of cyclic barriers throughout the whole computation; and the number of tasks is a parameter of the program, yet the number of cyclic barriers is not. All of the benchmarks found in Section 4.6 share these characteristics. Scaling a parallel program usually involves adding more tasks, whilst maintaining the same number of cyclic barriers; hence SG becomes beneficial at a larger scale.

The appropriate graph model for fork/join applications is harder to predict. For instance, in *nested* fork/join programming models, such as in X10, where join barriers (finishes) are lexically scoped, each task is registered with all join barriers that are enclosing its spawn location, *e.g.*, an X10 task spawned within the scope of three finishes is registered with three join barriers. The case complicates when join barriers are created dynamically in a recursive function call. For instance, languages with futures turn each function call into a join barrier, so it can happen that there are as many join barriers (resources) as there are tasks. In general, it is not possible to statically predict the ratio between resources and tasks in fork/join (and future) applications.

Java and X10 include multiple barrier abstractions to let applications choose from different programming models. Recent proposals of abstractions that use barrier synchronisation, in the context of X10 programming, make the case difficult for a fixed graph representation (be it the WFG or the SG). Atkins *et al.* design and implement *clocked variables* [13] that mediate the access of shared memory cells with barrier synchronisation in the context of X10. We benchmark three parallel algorithms that use clocked variables in Section 4.6 and the average edge count of each is different: in `SE` the edge count is similar between WFG and the SG; in `FI` the SG is on average twice as smaller; and in `FT` the average edge count of the WFG is ten times as smaller. Additionally, in the context of HJ, Shirako *et al.* propose using phasers for point-to-point synchronisation [94], so we expect the WFG to be more beneficial, and for the implementation of parallel reduction operations [95] that should favour the SG model.

The WFG and the SG We now rigorously define the WFG and the SG. The WFG is task-centric, so an edge (t_1, t_2) represents that task t_1 waits for task t_2 to synchronise, meaning that there exists a resource r such that $r \in W(t_1)$ and $t_2 \in I(r)$. Fig. 4.1a illustrates the WFG for state (I_1, W_1). The SG is resource-centric, so an edge (r_1, r_2) represents that resource r_1 impedes any



(a) A WFG representation. (b) An SG representation.

Figure 4.1: Two different graphs representing a deadlocked system.

task from synchronising via resource r_2, meaning that there exists a task t such that $r_1 \in W(t)$ and $t \in I(r_2)$. Fig. 4.1b depicts the SG for state (I_1, W_1).

Next, we formalise the notions of constructing a WFG and an SG from a resource-dependency.

Definition 4.3.1 (WFG construction)**.** *Let* wfg *be a function from resource-dependencies into WFG's:*

$$\text{wfg}\,(I, W) \stackrel{\text{def}}{=} \big(\mathcal{T}, \{(t_1, t_2) \mid r \in W(t_1) \wedge t_2 \in I(r)\}\big)$$

Formula $\text{wfg}\,(I_1, W_1)$ yields the graph in Fig. 4.1a:

$$\big(\mathcal{T}, \{(t_1, t_4), (t_2, t_4), (t_3, t_4), (t_4, t_1), (t_4, t_2), (t_4, t_3)\}\big)$$

Definition 4.3.2 (SG construction)**.** *Let* sg *be a function from resource-dependencies into SG's:*

$$\text{sg}\,(I, W) \stackrel{\text{def}}{=} \big(\mathcal{R}, \{(r_1, r_2) \mid t \in I(r_1) \wedge r_2 \in W(t)\}\big)$$

We apply Definition 4.3.2 and get the graph in Fig. 4.1b:

$$\text{sg}\,(I_1, W_1) = \big(\mathcal{R}, \{(r_1, r_2), (r_2, r_1)\}\big)$$

4.4 Results

Cycle detection in a graph has a complexity of O($e + v$) [101], for a graph with e edges and v vertices. From [15], we know that for any graph $e \leq v^2$, thus we can simplify the complexity to O($v + v^2$). And because in the WFG the vertices are tasks, then a deadlock detection algorithm that uses the WFG has a complexity of O($T + T^2$) for a system with T tasks.

Finding a cycle on the WFG is equivalent to finding a cycle in the SG.

Definition 4.4.1 (General Resource Graph (GRG) construction)**.** *Let* grg *be a function from resource-dependencies into GRG's:*

$$\text{grg}\,(I, W) \stackrel{\text{def}}{=} \big(\mathcal{R} \cup \mathcal{T}, \{(t, r) \mid r \in W(t)\} \cup \{(r, t) \mid t \in I(r)\}\big)$$

Lemma 4.4.1. *We have that $t_1 t_2$ is a walk on* wfg (D) *if, and only if, there exists a resource r such that $t_1 r t_2$ is a walk on* grg (D)*.*

Proof. Let grg $(D) = (V, E)$ and wfg $(D) = (V', E')$. ($\Longrightarrow$) Since $t_1 t_2$ is a walk on wfg (D), then $t_1 t_2 \in E'$. By Definition 4.3.1 there exists a vertex r such that $r \in W(t_1)$ and $t_2 \in I(r)$. Thus, by Definition 4.4.1 $(t_1, r) \in E$ and $(r, t_2) \in E$, and therefore $t_1 r t_2$ is a walk on grg (D).

($\Longleftarrow$) Since $t_1 r t_2$ is a walk on grg (D), then $(t_1, r) \in E$ and $(r, t_2) \in E$. From Definition 4.4.1 $r \in W(t_1)$ and $t_2 \in I(r)$. Thus, from Definition 4.3.1 $t_1 t_2 \in E'$ and therefore $t_1 t_2$ is a walk on wfg (D). □

Lemma 4.4.2. *We have that $r_1 r_2$ is a walk on* sg (D) *if, and only if, there exists a task t such that $r_1 t r_2$ is a walk on* grg (D)*.*

Proof. The proof follows an analogous reasoning to that of Lemma 4.4.1. □

Lemma 4.4.3. *If $w = t_1 \cdots t_n$ is a walk with a positive length on* wfg (D) *and $1 < k < n$, then there exists a walk $w' = r_1 \cdots r_k$ on* sg (D) *such that for all i where $1 \leq i \leq k$ we have $t_i r_i t_{i+1}$ is a walk on* grg (D)*.*

Proof. We prove by induction on k.

- Base case $k = 2$. Thus, $w = t_1 t_2 t_3 \cdots t_n$ and $n \geq 3$. By hypothesis, $t_1 t_2$ is a walk on wfg (D), so Lemma 4.4.1 yields that there exists a resource r_1 such that $t_1 r_1 t_2$ is a walk on grg (D). Similarly, from the hypothesis and using Lemma 4.4.1 $t_2 t_3$ is a walk on wfg (D), we get that there exists a resource r_2 such that $t_2 r_2 t_3$ is a walk on grg (D). Finally, we have that $r_1 t_2 r_2$ is a walk on grg (D), hence by Lemma 4.4.2, $r_1 r_2$ is a walk on sg (D).

- Inductive case $k = j + 1$. Hence, $w = t_1 \cdots t_j t_{j+1} \cdots t_n$, and $n > j \geq 2$. By the induction hypothesis we have that there exists a walk $r_1 \cdots r_j$ on sg (D) such that (i) for all i where $1 \leq i \leq j$ we have $t_i r_i t_{i+1}$ is a walk on grg (D). From (i) we have that (ii) $t_j r_j t_{j+1}$ is a walk on grg (D).

  By hypothesis, we also have that $t_{j+1} t_{j+2}$ is a walk on wfg (D), thus from Lemma 4.4.1, there exists a resource r_{j+1} such that (iii) $t_{j+1} r_{j+1} t_{j+2}$ is a walk on grg (D).

From (ii) $t_j r_j t_{j+1}$ and (iii) $t_{j+1} r_{j+1} t_{j+2}$ walks on grg (D), we get that $r_j t_j r_{j+1}$ is a walk on grg (D). Applying Lemma 4.4.2 to the latter, yields that $r_j r_{j+1}$ is a walk on sg (D). Thus, $r_1 \cdots r_j r_{j+1}$ is a walk on sg (D) and we are left with proving for all i where $1 \leq i \leq j+1$ we have $t_i r_i t_{i+1}$ is a walk on grg (D). But, we already know that (i) for all i where $1 \leq i \leq j$ we have $t_i r_i t_{i+1}$ is a walk on grg (D), so we just need to prove that $t_{j+1} r_{j+1} t_{j+2}$ is a walk on grg (D), which we have already shown with (iii).

□

Theorem 4.4.1. *There exists a cycle w on graph* wfg (D) *if, and only if, there exists a cycle w' on graph* sg (D).

Proof. ($\Longrightarrow$) The proof follows by induction on the length of w.

- Case length is 1, where $w = tt$ for some task t. By hypothesis tt is a walk on wfg (D). From Lemma 4.4.1 there exists a resource r such that trt is a walk on grg (D). Since trt is a walk on grg (D), then we know that (t, r) and (r, t) are edges on grg (D), and therefore rtr is also a walk on grg (D). Thus, from Lemma 4.4.2 and rtr is a walk, we get that rr is a walk on sg (D) and a cycle.

- Case length is greater than 1, where $w = t_1 \cdots t_n t_{n+1} t_1$ and $n \geq 2$. Applying Lemma 4.4.3 to $t_1 \cdots t_n t_{n+1}$, we get that $r_1 \cdots r_n$ is a walk on sg (D) such that (i) for all i where $1 \leq i \leq n$ we have $t_i r_i t_{i+1}$ is a walk on grg (D). Since $t_1 \cdots t_n t_{n+1}$ is a walk on wfg (D), thus from (i) we get that (ii) $t_1 r_1 t_2$ and (iii) $t_n r_n t_{n+1}$ are walks on grg (D). From $t_1 \cdots t_n t_{n+1} t_1$ is a walk on wfg (D), we get that $t_{n+1} t_1$ is a walk on wfg (D) and from Lemma 4.4.1, there exists a resource r such that (iv) $t_{n+1} r t_1$ is a walk on grg (D). From (iii) $t_n r_n t_{n+1}$ and (iv) $t_{n+1} r t_1$, we get that $r_n t_{n+1} r$; thus from Lemma 4.4.2 we get that (v) $r_n r$ is a walk on sg (D).

  From (iv) $t_{n+1} r t_1$ and (ii) $t_1 r_1 t_2$, we get that $r t_1 r_1$ is a walk on grg (D). Applying Lemma 4.4.2 to the latter, yields that (vi) $r r_1$ is a walk on sg (D).

  Finally, since (vi) $r r_1$, (v) $r_n r$, and $r_1 \cdots r_n$ are walks on sg (D), we get that $r r_1 \cdots r_n r$ is a walk on sg (D) and a cycle.

The proof for ($\Longleftarrow$) follows an analogous reasoning. □

The two crucial properties of our deadlock detection algorithm are: soundness (Theorem 4.4.2), where finding a cycle in the SG corresponds to a deadlocked state; and completeness (Theorem 4.4.3), where the SG of any deadlocked state contains a cycle.

We distinguish between a totally deadlocked state (Definition 4.4.2) and a deadlocked state (Definition 4.4.3), formalised in the following two definitions.

Definition 4.4.2 (Totally deadlocked state)**.** *A state* $\big(M, T\big)$ *is totally deadlocked if, and only if,* $T \neq \emptyset$*, and for all* $t \in \operatorname{dom} T$ *we have that* $T(t) = \texttt{await}(p, n); b$ *and there is a task* $t' \in \operatorname{dom} T$ *where* $M(p)(t') < n$*.*

Any totally deadlocked state is also a deadlocked state.

Definition 4.4.3 (Deadlocked state)**.** *State* $\big(M, T' \uplus T\big)$ *is deadlocked on task map* T *if, and only if, state* $\big(M, T\big)$ *is totally deadlocked.*

The relationship between a blocked task in a state and an edge in a WFG graph is fundamental for the results we establish in this section.

Lemma 4.4.4. *Let* $\psi\,\big(M, T\big) = (I, W)$*,* $\operatorname{wfg}(D) = (V, E)$*,* $\gamma^{-1}(r) = (p, n)$*. We have that* $(t_1, t_2) \in E$ *if, and only if,* $T(t_1) = \texttt{await}(p, n); b$ *and* $M(p)(t_2) < n$*.*

Proof. ($\implies$) We have that $(t_1, t_2) \in E$, thus by Definition 4.3.1 there is a resource r such that $r \in W(t_1)$ and $t_2 \in I(r)$. From Definition 4.1.2 and $r \in W(t_1)$, we get that $T(t_1) = \texttt{await}(p, n); b$ and $\gamma(p, n) = r$. From Definition 4.1.2 and $t_2 \in I(r)$, we obtain that $M(p)(t_2) < n$.

($\impliedby$) We have that $T(t_1) = \texttt{await}(p, n); b$ and $M(p)(t_2) < n$. From Definition 4.1.2 and $T(t_1) = \texttt{await}(p, n); b$, we get that is a resource r such that $\gamma(p, n) = r$ and $r \in W(t_1)$. From Definition 4.1.2 and $M(p)(t_2) < n$, we get that $t_1 \in I(r)$. We apply Definition 4.3.1 to $t_1 \in I(r)$ and $r \in W(t_2)$ and get that $(t_1, t_2) \in E$. □

Theorem 4.4.2 (Soundness)**.** *If* w *is closed on* $\operatorname{wfg}\left(\psi\,\big(M, T\big)\right)$ *with a positive length, then there exists task map* T' *and* T'' *such that* $T = T' \uplus T''$*,* $\operatorname{dom} T' = \{t \mid \forall t \in w\}$*, state* $\big(M, T\big)$ *is locally deadlocked on* T'*.*

Proof. Let $\operatorname{wfg}\left(\psi\,\big(M, T\big)\right) = (V, E)$ and

$$X \stackrel{\text{def}}{=} \{t_1 \colon t_2 \mid \forall (t_1, t_2) \in w\} \tag{4.1}$$

First, we show that $\operatorname{dom} X \subseteq \operatorname{dom} T$. Let $t_1 \in \operatorname{dom} X$, we need to show that $t_1 \in \operatorname{dom} T$. If $X(t_1) = t_2$, then by Eq. (4.1) $(t_1, t_2) \in w$ and therefore $(t_1, t_2) \in E$. Thus, by Lemma 4.4.4 $T(t_1) = \texttt{await}(p, n); b$.

Now that we showed that $\operatorname{dom} X \subseteq \operatorname{dom} T$, then let $T = T_1 \uplus T_2$ such that $\operatorname{dom} T_1 = \operatorname{dom} X$. We have that $T_1 \neq \emptyset$, since the length of w is $|\operatorname{dom} X| > 0$. Second, we prove that $\big(M, T_1\big)$ is globally deadlocked. By Definition 4.4.2 for any task $t_1 \in \operatorname{dom} T_1$, we need to show that (1) $T_1(t_1) = \texttt{await}(p, n); b$ and that (2) there exists a task t_2 such that $M(p)(t_2) < n$.

1. Let $t_1 \in \operatorname{dom} T_1$, then $t_1 \in \operatorname{dom} X$ and therefore there is a task t_2 such that $(t_1, t_2) \in w$ and therefore $(t_1, t_2) \in E$. Applying Lemma 4.4.4 to $(t_1, t_2) \in E$, yields that $T(t_1) = \texttt{await}(p, n); b$ and $M(p)(t_2) < n$. Since $t_1 \in \operatorname{dom} T_1$, then $T_1(t_1) = \texttt{await}(p, n); b$.

2. We are left with showing that $t_2 \in \operatorname{dom} T_1$ (since we already know that $M(p)(t_2) < n$). By hypothesis w is a cycle, thus there exists a task t_3 such that $(t_2, t_3) \in w$. We apply Eq. (4.1) to $(t_2, t_3) \in w$ and get that $t_2 \in \operatorname{dom} X$. Therefore, $t_2 \in \operatorname{dom} T_1$.

Finally, applying Definition 4.4.3 to $\big(M, T_1\big)$ is globally deadlocked, yields that $\big(M, T_1 \uplus T_2\big)$ is locally deadlocked on T_1. □

Completeness

The intuition behind the proof of completeness can be divided into two parts. First, by showing that any globally deadlocked state has a cycle. Second, by establishing the subgraph relation between a globally deadlocked state and a locally deadlocked state.

It is easy to see that any globally deadlocked task t has a positive out-degree.

Lemma 4.4.5. *Let* $(V, E) = \operatorname{wfg}(\psi(S))$. *If* S *is globally deadlocked and* $t \in V$, *then* t *has a positive out-degree.*

Proof. Let $S = \big(M, T\big)$. By Definition 4.4.2 there exists a task t such that $T(t) = \texttt{await}(p, n); b$ and there is a task $t' \in \operatorname{dom} T$ where $M(p)(t') < n$.

From Lemma 4.4.4, we get that $(t, t') \in E$ and t has a positive out-degree. □

A graph in which all vertexes have a positive out-degree is cyclic.

Lemma 4.4.6. *Let* $G = \operatorname{wfg}(\psi(S))$. *If* S *is globally deadlocked, then there exists a cycle* w *on* G.

Proof. Let $G = (V, E)$. Applying Lemma 4.4.5 to the hypothesis yields that every vertex has a positive out-degree. Hence, by the contrapositive of [15, Proposition 1.4.2], (V, E) has a cycle w. □

Next, is an auxiliary lemma to establish the subgraph relationship between WFG's.

Lemma 4.4.7. *For all* $t \notin \operatorname{dom} T$, *we have that* $\operatorname{wfg}(\psi\big(M, T\big))$ *is a subgraph of graph* $\operatorname{wfg}(\psi\big(M, T \uplus \{t\colon b\}\big))$.

Proof. Let $\text{wfg}\left(\psi\left(M, T\right)\right) = (V, E)$ and $\text{wfg}\left(\psi\left(M, T \uplus \{t\colon b\}\right)\right) = (V', E')$. Graph (V, E) is a subgraph of (V', E') if 1) $V \subseteq V'$, 2) $E \subseteq E'$, 3) $\forall (t, t') \in E \implies t \in V \wedge t' \in V$.

1. We have that $V \subseteq V'$ holds, since $V = V' = \mathcal{T}$.
2. If $(t_1, t_2) \in E$, then $(t_1, t_2) \in E'$. By Lemma 4.4.4 and $(t_1, t_2) \in E$, we have that $T(t_2) = \texttt{await}(p, n); b'$ and $M(p)(t_1) < n$. We have that $t \notin \operatorname{dom} T$, thus $T \uplus \{t\colon b\}(t_2) = \texttt{await}(b', n); b$. From $T \uplus \{t\colon b\}(t_2) = \texttt{await}(p, n); b'$, $M(p)(t_1) < n$, $\text{wfg}\left(\psi\left(M, T \uplus \{t\colon b\}\right)\right) = (V', E')$, and Lemma 4.4.4, we get that $(t_1, t_2) \in E'$.
3. We show that $\forall (t, t') \in E \implies t \in V \wedge t' \in V$. By definition $t \in \mathcal{T}$ and $t' \in \mathcal{T}$.

□

Lemma 4.4.8. *Graph* $\text{wfg}\left(\psi\left(M, T\right)\right)$ *is a subgraph of* $\text{wfg}\left(\psi\left(M, T \uplus T'\right)\right)$.

Proof. The proof follows by induction on the structure of T'. Let

$$\text{wfg}\left(\psi\left(M, T\right)\right) = (V, E) \quad \text{and} \quad \text{wfg}\left(\psi\left(M, T \uplus T'\right)\right) = (V', E')$$

We inspect T'.

- Case T' is $\emptyset$. To show that (V, E) is subgraph of itself, we just need to show that $\forall (t, t') \in E \implies t \in V \wedge t' \in V$, which holds by Definition 4.3.1, since $V = V' = \mathcal{T}$.
- Case T' is $T'' \uplus \{t\colon b\}$. By the induction hypothesis, graph $\text{wfg}\left(\psi\left(M, T\right)\right)$ is a subgraph of $\text{wfg}\left(\psi\left(M, T \uplus T''\right)\right)$. By Lemma 4.4.7 this case holds.

□

Finally, we can establish the completeness theorem.

Theorem 4.4.3 (Completeness)**.** *If state S is locally deadlocked on T and $t \in \operatorname{dom} T$, then there exists a t'-cycle on* $\operatorname{wfg}(\psi(S))$ *such that t' is reachable from t.*

Proof. By Definition 4.4.3 we have that $S = \big(M, T \uplus T'\big)$ and that $\big(M, T\big)$ is globally deadlocked. Let $(V_1, E_1) = \operatorname{wfg}(\psi(S))$. Let (V_2, E_2) be the subgraph of (V_1, E_1) of all vertices reachable from t. It is easy to see that V_2 is nonempty. From Definition 4.4.2 there is a task $t'' \in \operatorname{dom} T$ such that $M(p)(t'') < n$. Applying Lemma 4.4.4 to $T(t) = \texttt{await}(p, n); b$ and $M(p)(t'') < n$, we get that (t'', t), so $t'' \in V_2$ and $(t'', t) \in E_2$.

We have that every $V_2 \subseteq \operatorname{dom} T$. Let $T_2 = \{T(t) \mid t \in \operatorname{dom} V_2\}$. We now show that T_2 is globally deadlocked. For that it is enough to pick $t_1 \in V_2$ and show that (i) $T_2(t_1) = \texttt{await}(p, n); b$ and there exists a task t_2 such that (ii) $t_2 \in \operatorname{dom} T_2$ and (iii) $M(p)(t_2) < n$. Since $t_1 \in \operatorname{dom} T$ and $\big(M, T\big)$ is globally deadlocked, then by Definition 4.4.2 $T(t_1) = \texttt{await}(p, n); b$ and there exists a task t_2 such that $t_2 \in \operatorname{dom} T$ and (iii) $M(p)(t_2) < n$. Given that $T(t_1) = T_2(t_1)$, then (ii) $T_2(t_1) = \texttt{await}(p, n); b$. We still need to show (i). Applying Lemma 4.4.4 to $T(t_1) = \texttt{await}(p, n); b$, $t_2 \in \operatorname{dom} T$, and (iii) yields $(t_1, t_2) \in E_1$. Thus, t_1 reaches t_2; and therefore, $t_2 \in V_2$ and $(t_1, t_2) \in E_2$. Hence,(i) $t_2 \in \operatorname{dom} T_2$.

From Lemma 4.4.6 and globally deadlocked state $\big(M, T_2\big)$, we get that there exists a t'-cycle on graph $\operatorname{wfg}\big(\psi\big(M, T_2\big)\big)$. By definition, we also know that any t' is reachable from t. We apply Lemma 4.4.8 and obtain that $\operatorname{wfg}\big(\psi\big(M, T_2\big)\big)$ is a subgraph of $\operatorname{wfg}\big(\psi\big(M, T' \uplus T\big)\big)$, hence w on $\operatorname{wfg}\big(\psi\big(M, T' \uplus T\big)\big)$. □

4.5 Armus: a tool for runtime deadlock verification

Armus is a dynamic verification tool of barrier deadlocks that implements the theory in Section 4.3. Our tool verifies more barrier synchronisation patterns than current state-of-the-art and improves the scalability of graph-based verification. We introduce Armus-X10 and JArmus as two applications of Armus. These are the *first barrier deadlock verification tools* for X10 and Java. The applications feature distributed deadlock detection, and local deadlock avoidance.

The main limitations of state-of-the-art runtime verification of barrier deadlocks are: (i) a representation of concurrency constraints that assumes static barrier membership, and (ii) a commitment to the WFG model, which is optimised for concurrency constraints with more barriers than tasks (a rare situation for classical parallel programs). Naive extensions to resolve (i) face the problem of maintaining the membership status of barriers consistently and efficiently;

this issue is compounded in the distributed setting, which is a key design point of deadlock verification for languages like X10/HJ. Issue (ii) is related to the dynamic nature of such barrier applications, where the number of tasks and barrier synchronisations may not be known until run-time and may vary during execution. Committing to a particular graph model can thus hinder the scalability of dynamic verification. In the general case we cannot determine which model is most suitable statically; moreover, this property may change as execution proceeds.

To address (i), Armus uses our novel representation of concurrency dependencies, based on events in the sense of Lamport's logical clocks (see Section 4.1). The analysis of dynamic membership is simplified because it avoids tracking the arrival status, that in a distributed system is a global state (*i.e.,* scattered among many sites), thus a challenging procedure to maintain.

Armus addresses (ii) with a new technique that automatically selects between two graph models according to the monitored concurrency constraints. The standard graph model used in graph analysis, the WFG, comes from distributed databases [62], a setting with a fixed number of tasks and dynamic resource creation. The underlying assumptions of the WFG no longer hold for languages with dynamic tasks and dynamic barrier creation (first-class barriers), such as X10 and Java. For these applications, Armus proposes a technique that selects either the WFG or the SG depending on the ratio between tasks and barriers. The difference on the size of the graph can be dramatic. For instance, in benchmark PS, the average edge count decreases from 781 edges to 6 edges (see Section 4.6). In our evaluation, the automatic model selection outperforms the usual approach of a fixed graph representation.

Architectural overview The architecture of Armus is divided into two layers: the *application layer* that receives a trace of operations from the running program, and the *verification layer* that receives a set of concurrency constraints from the application layer. The application layer is specific to each language we check. The verification layer is our library that checks for deadlocks in a resource-dependency state D.

The verification algorithm can be used to *avoid* and to *detect* deadlocks. In the former, Armus throws an exception *before* deadlocks happen. The programmer can treat the exceptional situation to develop applications resilient to deadlocks. In the latter, verification is performed periodically and can only report already existing deadlocks, with the benefit of a lower performance overhead.

Verification library Armus' deadlock verification library implements the theory described in Section 4.3. The main features of the library are (i) a deadlock

detection algorithm that is fault-tolerant and distributed; and (ii) a scalable deadlock verification technique (*i.e.,* the adaptive graph representation).

The input of the verification library is a trace of the running program. Essentially, whenever a task of the program blocks, the application layer invokes the verification library by producing its blocked status: a set of waiting $W(t)$ and set of impeding resources $\{r \mid \forall r \colon t \in I(r)\}$. The library is divided into two services: the *edge buffer* maintains the blocked status of all tasks, implementing the resource-dependency state D; the *deadlock checker* analyses the edge buffer for any deadlock, using Definition 4.3.1 and Definition 4.3.2. Maintaining the blocked status is more frequent than checking for deadlocks, so the edge buffer rearranges D per task to optimise updates. The deadlock checker internally transforms the dependencies into a graph and then performs cycle detection with JGraphT [2].

The verification library provides two graph selection modes: fixed or automatic. In the former, the verification always uses the same graph model. State-of-the-art tools are fixed to the WFG model. In the automatic mode, the verification library selects the graph model according to the ratio between blocked tasks and registered phasers. This means that the graph model used for cycle detection can change over time.

We briefly describe the implementation of each mode. In the *fixed to WFG mode* (see Definition 4.3.1), the algorithm iterates over a copy of the blocked tasks twice. First, uses the impeding resource of each blocked task to construct map I. Second, generates a WFG-edge from each waiting resource r to each task in $I(r)$. In the *fixed to SG mode* (see Definition 4.3.2), it iterates over each blocked task (available in the edge buffer) and generates an SG-edge from each impeding resource to each blocked resource. The *adaptive mode* tries to build an SG first; if during the construction of the SG it reaches a size threshold, then it builds a WFG instead. The size threshold is reached if at any time there are more SG-edges than twice the number of tasks processed thus far. The value of the threshold was obtained based on experiments on the available benchmarks.

Distributed deadlock detection Armus adapts the traditional one-phase deadlock detection [65] to barrier synchronisation and introduces support for fault tolerance. We briefly describe our *adapted* one-phase deadlock detection algorithm. A distributed program is composed of various *sites* that communicate among each other, each runs a copy of Armus. Every Armus instance of a distributed program has access to a remote data store server Redis,[3] called the *global edge buffer*. Tasks update their blocked status, as usual, but target an

[2] `http://jgrapht.org/`

[3] `http://redis.io/`

edge buffer local to their site. While the distributed program runs, each site periodically publishes a snapshot of its local edge buffer to the global edge buffer. The deadlock checker, that runs from each Armus instance, requires a global view of the system, so it operates on the blocked status of the global edge buffer.

The differences with reference to the original algorithm in [65] are:

- Armus uses logical clocks to represent barrier synchronisations (see Section 4.1) and maintain global data consistency; the original algorithm requires vector clocks to represent lock synchronisations.
- For fault-tolerance concerns, the global status of Armus is maintained in a dedicated server, and all sites check for deadlocks. In contrast, in [65] there is a designated control site that collects the global status and performs graph analysis. Our benchmarks, in Section 4.6, show that the verification overhead has a negligible impact for 64 tasks.

The verification algorithm is fault-tolerant, since it continues executing despite (i) site-failures and (ii) data store-failures. Such feature is of special interest for checking fault-tolerant applications, like Resilient X10 [35]. The algorithm resists (i) because the deadlock checker executes at each site and does not depend on the cooperation of other sites to function. The algorithm resists (ii) because Redis itself is fault-tolerant.

Verifying X10 and Java We present two verification applications to check for barrier deadlocks: JArmus for Java programs and Armus-X10 for X10 programs. These tools work by "weaving" the verification into programs. The input is a compiled program to be verified (Java bytecode); the output is a verified program (Java bytecode) that includes dynamic checks for deadlock verification. JArmus and Armus-X10 layers implement the resource-dependency construction from Section 4.1.

JArmus and Armus-X10 share the same usage and design. The implementation of each of these verification tools is divided into two components: the resource mapper and the task observer. The resource mapper converts synchronisation events to resources. The task observer intercepts blocking calls to inform Armus that the current task is blocked with a set of resource edges. The task observer is programmed with Aspect-Oriented programming, through AspectJ [61].

Armus-X10 can verify any program written in X10 that uses: clocks, finishes, and the `SPMDBarrier`; the tool can verify distributed applications. Unlike in Java, automatic instrumentation is possible. The X10 runtime provides information about the registered clocks and registered finishes of a given task, which is required to construct the concurrency dependencies of each task. X10 can be

compiled to Java bytecode, called Managed X10, and to machine code, called Native X10. Currently, our application only supports Managed X10.

JArmus supports `CountDownLatch`, `CyclicBarrier`, `Phaser`, and `ReentrantLock` class operations. In Java, the relationship between the participants of barrier synchronisation and tasks is implicit. For example, when using a `CyclicBarrier` the programmer declares the number of participants and then shares the object with those many tasks. It is not specified which tasks participate in the synchronisation. JArmus has no way of reconstructing this information for the `CountDownLatch`, `CyclicBarrier`, and `Phaser` classes, so the programmer must annotate its code to supply the barriers the each task is registered with. Each task, upon starting up, must invoke `JArmus.register(b)` per barrier `b` it uses (similarly to the X10 `clocked`). Instances of the class `ReentrantLock` do not require annotations.

4.6 Evaluation

The aim of the evaluation process is to 1) ascertain whether the performance impact of Armus scales with the increase in the number of tasks, 2) evaluate the performance overhead of distributed deadlock detection, and 3) compare execution impact the SG with the WFG and with adaptive approach.

The hardware used to run the benchmarks has four AMD Opteron 6376 processors, each with 16 cores, making a total of 64 cores. There are 64GB of available RAM. The operating system used is Ubuntu 13.10. For the languages, we used Java build 1.8.0_05-b13, and X10 version 2.4.3.

We follow the *start-up performance* methodology detailed in [43]. We take 31 samples of the execution time of each benchmark and discard the first sample. Next, we compute the mean of the 30 samples with a confidence interval of 95%, using the standard normal z-statistic.

Impact of non-distributed verification

The two goals of this evaluation are: to measure the impact of verification on standard Java benchmarks, and ii) to measure whether the verification scales with the increase of the number of tasks. We run the verification algorithm against a set of standard parallel benchmarks available for Java. JArmus is run in the detection mode (every 100 milliseconds) and in the avoidance mode, both use the adaptive graph model. Note that the Java applications we checked are not distributed.

We select benchmarks from the NASA Parallel Benchmark (NPB) suite [42] and the Java Grande Forum (JGF) [99] benchmark suite. The NPB ranges from

Table 4.1: Relative execution overhead in detection mode.

Threads	**2**	**4**	**8**	**16**	**32**	**64**
BT	3%	-4%	0%	-5%	0%	7%
CG	7%	0%	7%	15%	12%	9%
FT	1%	0%	-1%	-7%	0%	0%
MG	-5%	0%	0%	0%	11%	13%
RT	-4%	0%	0%	0%	0%	8%
SP	-1%	4%	4%	2%	0%	

kernels to pseudo-applications, taken primarily from representative Computational Fluid Dynamics (CFD) parallel applications. The JGF is divided into three groups of applications: micro-benchmarks, computational kernels, and pseudo-applications. All benchmarks proceed iteratively, and use a fixed number of cyclic barriers to synchronise stepwise. Furthermore, all benchmarks check the validity of the produced output.

For the sake of reproducibility we list the parameters of the benchmarks run as specified in [42, 99]: BT uses size A, CG uses size C, the Java version of FT uses size B, MG uses size C, RT uses B, and SP uses size W. The input set chosen for benchmark SP only allows it to scale up to 31 tasks. For simplicity, in the evaluation we consider that this benchmark scales up to 32 tasks.

Fig. 4.2 summarises the comparative study of the execution time for each benchmark. Tables 4.1 and 4.2 list the relative runtime overhead in detection and in avoidance. The results for the NPB and JGF benchmark suites are depicted in Figs. 4.2a to 4.2f. In detection mode, since there is a dedicated task to perform verification, we observe that the overhead does not increase linearly as we add more tasks. The relative runtime overhead sits below 15% and in most cases is negligible. In avoidance mode, each task checks the graph whenever it blocks, so as we add more tasks, the execution overhead increases. Still, in the worst case, benchmark CG, the overhead is 50%, which is acceptable for application testing purposes.

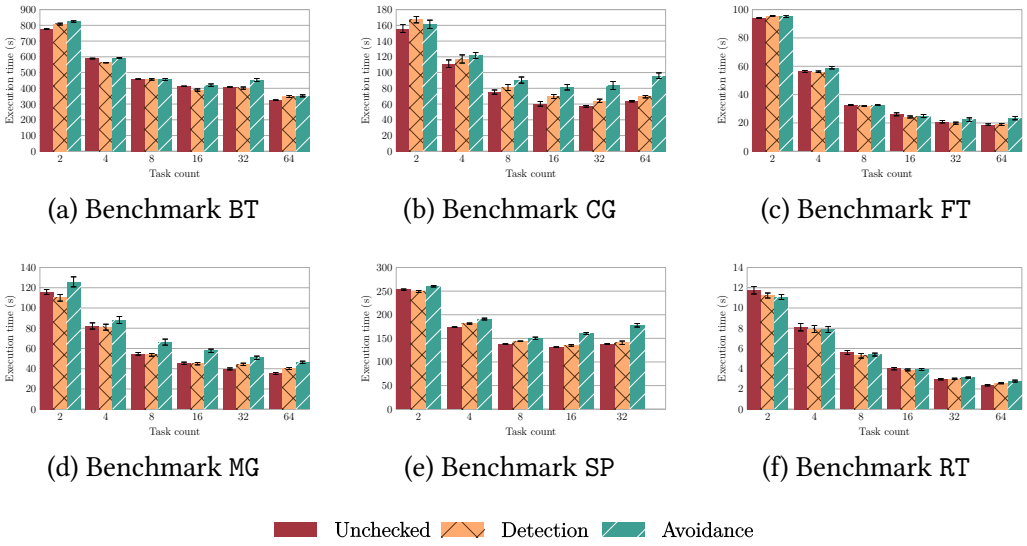
Impact of distributed verification

The goal of the evaluation is to measure the runtime overhead of deadlock detection in available X10 distributed applications. Armus-X10 is configured with the distributed deadlock detection mode, running the verification algorithm every 200 milliseconds. The chosen benchmarks are available via the X10 source code repository [4]. Deadlock avoidance is unavailable in the distributed setting.

[4] https://svn.code.sf.net/p/x10/code/trunk/

Table 4.2: Relative execution overhead in avoidance mode.

Threads	**2**	**4**	**8**	**16**	**32**	**64**
BT	5%	0%	0%	0%	11%	8%
CG	0%	9%	20%	34%	46%	50%
FT	1%	4%	0%	0%	7%	25%
MG	8%	7%	21%	27%	27%	30%
RT	-5%	0%	0%	0%	5%	16%
SP	2%	9%	8%	22%	28%	



(a) Benchmark BT (b) Benchmark CG (c) Benchmark FT

(d) Benchmark MG (e) Benchmark SP (f) Benchmark RT

Figure 4.2: Comparative execution time for non-distributed benchmarks (lower means faster).

Benchmarks FT and STREAM come from the HPC Challenge benchmark [75], SSAC2 is an HPCS Graph Analysis Benchmark [14], JACOBI and KMEANS are available from the X10's website. For reproducibility purposes the non-default parameters we select are: FT magnitude 11; KMEANS 25k points, 3k clusters to find, and 5 iterations; JACOBI matrix of size 40, maximum iterations are 40; SSCA2 2^{15} vertices, a with a probability of 7%, and no permutations; STREAM with size of 524k.

Fig. 4.3 depicts the execution time of each benchmark with and without verification. There is no statistical evidence of an execution overhead with running deadlock detection mode.

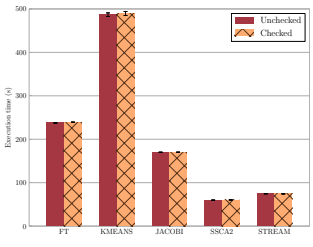


Figure 4.3: Comparative execution time for distributed deadlock detection (lower means faster).

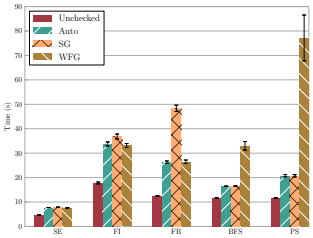


Figure 4.4: Comparative execution time for different graph model choices (lower means faster), using deadlock avoidance.

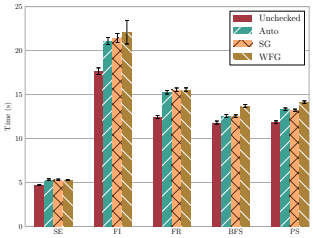


Figure 4.5: Comparative execution time for different graph model choices (lower means faster), using deadlock detection.

Impact of the graph model choice

The goal of this evaluation is to measure the impact of the graph model in the verification procedure. To this end we analyse the worst case behaviour: programs that generate graphs with thousands of edges. In particular, we evaluate our adaptive model selection against the usual fixed model selection (WFG and SG).

We select a suite of programs that spawn tasks and create barriers as needed, depending on the size of the program, unlike the classical parallel applications we benchmark in Sections 4.6 and 4.6 where the number of tasks should correspond to the number of available processing units (cores). The suite of programs exercises different worst case scenarios for the verification algorithm: many tasks *versus* many barriers.

The chosen benchmarks are educative programs taken from the course on *Principles and Practice of Parallel Programming*, taught by Martha A. Kim and Vijay A. Saraswat, Fall 2013 [109]. `BFS` performs a parallel breadth-first search on a randomly generated graph. There is a task per node being visited and a barrier per depth-level of the graph. `FI` computes a Fibonacci number iteratively with a shared array of *clocked variables* (each pairs a barrier with a number). Each element of the array holds the outcome of a Fibonacci number. When the program starts it launches n tasks. The i-th task stores its Fibonacci number in the i-th clocked variable and synchronises with task $i + 1$ and task $i + 2$ that read the produced value. `FR` computes a Fibonacci number recursively. Recursive calls are executed in parallel and a clocked variable synchronises the caller with the callee. `SE` implements the Sieve of Eratosthenes using clocked variables. There is a task per prime number and one clocked variable per task. `PS` computes the prefix sum—or cumulative sum—for a given number of tasks.

Table 4.3: Edge count and verification overhead per benchmark per graph mode.

		SE	FI	FR	BFS	PS
Auto	Edges	23	1074	140	5	7
	Avoidance	75%	94%	117%	45%	82%
	Detection	25%	24%	25%	9%	18%
SG	Edges	51	2137	1643	7	6
	Avoidance	75%	112%	300%	45%	82%
	Detection	25%	24%	25%	9%	18%
WFG	Edges	23	1281	94	579	781
	Avoidance	75%	94%	117%	200%	600%
	Detection	25%	29%	25%	18%	27%

Given an input array with as many elements as there are tasks, the outcome of task i is the partial sum of the array up to the i-th element. All tasks proceed stepwise and are synchronised by a global barrier.

Figs. 4.4 and 4.5 depict the execution time of each benchmark verified by Armus-X10 in avoidance and detection modes (respectively) where we vary the selection method of the graph model. Table 4.3 lists the average number of edges used in verification and the relative execution time overhead of each benchmark.

We can classify the benchmarks in three groups according to the ratio between the number of tasks and the number of resources: i) similar count of tasks and resources, benchmark SE; ii) much more resources than tasks, benchmarks FI and FT; and iii) much more tasks than resources, benchmarks BFS and PS. When i) there are as many resources as there are tasks, then all graph models perform equally well. When ii) there are more resources than tasks, and iii) vice-versa, the choice of the graph model is of major importance for a verification with low impact on the execution time.

Even in the worst case behaviour for analysis the largest verification overhead with deadlock detection is 25%; for deadlock avoidance the largest is 117%. For both cases we consider adaptive graph selection. Overall, the approach of the adaptive graph model outperforms the fixed graph model approach. The adaptive approach can save up to 9% of execution overhead in deadlock detection *versus* a fixed model. The graph model choice severely amplifies the verification overhead in deadlock avoidance. The case in point is benchmark PS, where the verification overhead ranges from 600% (fixed) down to 82% (adaptive).

Chapter Five

Deadlock prevention

A way to prevent deadlocks is by restricting the expressiveness of synchronisation mechanisms. We propose a minimal language, called SBrenner, that incorporates three techniques to achieve deadlock freedom.

Nested fork/join. This programming model consists of two primitives: the *async* forks tasks, and the *finish* joins the execution of tasks. A finish accepts a program as a parameter; the instructions are executed sequentially. After executing the instructions in a finish block, the task waits in a join barrier for the termination of any task spawned within the scope of the finish block. This restriction prevents deadlocks that arise from the interaction between multiple join barriers.

Await on all registered phasers. Task can only await on phasers they are registered with. Additionally, a task must await on every phaser it is registered with at once. This restriction prevents deadlocks that arise from the interaction between multiple cyclic barriers.

Cyclic barrier visibility. Instructions within the body of a finish cannot access phaser names declared outside (before) of that finish. This restriction prevents deadlocks that arise from the interaction between join-barriers and phasers.

The next section introduces the design of SBrenner: we show deadlocked programs written in Brenner that motivate extensions to the language constructs. Sections 5.2 and 5.3 introduce the syntax and the semantics of SBrenner. Section 5.4 presents a mechanism, called a type system, to specify (and enforce) a discipline on phaser usage, inspired by the X10 and HJ languages.

5.1 Language restrictions

Fork/join deadlocks. Brenner makes it trivial to write a fork/join program that deadlocks. In the next listing, task `t1` launches task `t2` and then waits for

it to finish. Task `t2` decides to wait for task `t1` to conclude and therefore both tasks reach a deadly embrace.

Listing 5.1: A fork/join deadlock.

```
1  // t1 -- parent task
2  p1 = newPhaser(); // join barrier for t1
3  t2 = newTid();
4  p2 = newPhaser(); // join barrier for t2
5  reg(p2, t2);
6  fork(t2,          // child task
7    await(p1, 1);   // await t1 to finish
8    adv(p2);        // signal end of t2
9    end
10 ); // end of t2
11 dereg(p2);        // t2 is the only participant
12 await(p2, 1);     // await t2 to finish
13 adv(p1);          // signal end of t1
14 end
```

The first proposal of a fork/join programming model [84] includes syntactic restrictions to render its programming model deadlock free. This restricted programming model is known as the *nested* fork/join. The idea behind the nested model is to:

- assign a parent-child relationship between the *parent* task that forks, and the forked *child* task;
- disallow tasks from awaiting ancestor and sibling tasks.

SBrenner defines the nested fork/join model of X10 and HJ. To limit tasks from awaiting siblings and ancestors, we remove task names from the syntax of SBrenner. Such alteration affects task creation and phaser registration (whose discussion we postpone). Next, we introduce specialised constructs to be used instead of phasers.

SBrenner uses instruction `ASYNC` to fork a task. Instruction `FINISH` is used to await the termination of a group of tasks. The instruction expects a program as a parameter, called the body. Any task (indirectly) spawned within the body of a finish is registered with its join barrier. After executing the body, the task awaits at the implicit join barrier.

The next example, written in SBrenner, fixes the fork/join deadlock in Listing 5.1. The child task can no longer wait for its parent task, because there

are no task names to refer to. When a task terminates it automatically signals the join barrier of the enclosing finish, as in Line 4. The parent task awaits its child in Line 6 after executing the finish body.

```
1 // t1 -- parent task
2 finish(
3   async(     // child task
4     end      // signals end of child
5   );
6   end        // awaits child to finish
7 );
8 end
```

Cyclic barrier deadlocks. The simplest way to prevent deadlocks that arise from the interaction between cyclic barriers is to restrict the language to have a single, global cyclic barrier. The Bulk-Synchronous Parallel programming model [103] champions the use of such restriction.

X10 introduces a technique that copes with multiple cyclic barriers: the language enforces each task to wait on all registered barriers at the same time. The increase of expressiveness, with respect to having a single cyclic barrier, is that two groups of tasks can synchronise independently from one another, as long as each group uses different barriers. The work in [92] formalises the technique, but lacks a formal proof of the deadlock-freedom property.

Waiting only on all registered phasers is not enough to prevent deadlocks in Brenner. One reason it fails is because a task can wait for any phase, as in the following one liner example.

```
1 p = newPhaser(); await(p, 2); end
```

For phasers it makes sense to restrict the phase number a task can wait for. Let instruction AWAIT (without arguments) be such that it awaits on every registered phaser using the task's local phase, thereby preventing the deadlock in the previous example.

```
1 p = newPhaser();
2 await; // await(p, 0);
3 end;
```

This restriction is still not enough to prevent the following deadlocked program. Task `t1` waits for task `t2` to advance to phase 1 on phaser `p`, while at the same time task `t2` waits for task `t1` to advance to phase 1 on phaser `q`. In Line 3 we note that async accepts a sequence of phaser names in which the spawned task becomes registered with. Here, the spawned task is registered with `p` and `q`.

Listing 5.2: Deadlocked program using two phasers.

```
1  // task t1
2  p = newPhaser(); q = newPhaser();
3  async(p,q, // task t2
4    adv(p);
5    await;    // await(p, 1) await(q, 0);
6    end
7  );
8  adv(q);
9  await;      // await(p, 0) await(q, 1);
10 end
```

SBrenner proposes a single change to counter this source of deadlocks: a task must advance the exact same number of times all registered phasers before awaiting. That is, before awaiting if the task is registered on two phasers p and q and it advances p twice, then it must also advance phaser q twice. We introduce a new language construct `next` that is used to check that all phasers are advanced exactly once. If the programmer forgets to advance a phaser, then the program is invalid and must be rejected by the typechecker.

In X10, there is a single operation that advances all phasers and then awaits on all phasers. The novelty of our technique is twofold. First, a task can advance multiple phases before awaiting. Second, a task can suppress waiting altogether and perform the producer-consumer synchronisation pattern, something unfeasible with X10's cyclic barriers.

In the next example, we fix the deadlock of Listing 5.2. SBrenner enforces the programmer two advance both phasers before awaiting. Instruction `next` demarcates that all phasers have been advanced once.

Dangling participant deadlocks. A simpler form of deadlock has to do with dangling participants, where a task awaits forever terminated tasks. Java and MPI suffer from this problem. For example, if in Section 5.1 we remove the two instructions of task `t2`, then task `t1` becomes deadlock waiting for task `t2`.

Listing 5.3: Fixed deadlock with two phasers.

```
1  // task t1
2  p = newPhaser(); q = newPhaser();
3  async(p,q,  // task t2
4    adv(p); adv(q);
5    await;     // await(p, 1) await(q, 1);
6    next;
7    end
8  );
9  adv(p); adv(q);
10 await;       // await(p, 1) await(q, 1);
11 next;
12 end
```

```
1  // task t1
2  p = newPhaser(); q = newPhaser();
3  async(p,q,  // task t2
4    end       // forgets to deregister
5  );
6  adv(p); adv(q);
7  await;       // forever waiting for t2
8  next;
9  end
```

In X10 and HJ every task implicitly deregisters from all barriers at the end of its execution; there is no way to identify crashed tasks. SBrenner enforces that the programmer explicitly deregisters from registered phasers before terminating every task.

Fork/join and cyclic barrier deadlocks. Combining cyclic barriers with the fork/join programming model introduces yet another form of deadlocks. The next program deadlocks because the tasks that forks (task `t1`) is waiting for the forked task `t2` to finish, while task `t2` is waiting for task `t1` to advance phaser `p`.

SBrenner employs a technique introduced by HJ: the "Immediately Enclosing Finish (IEF) scope rule" states that a task cannot be registered with phasers declared outside their immediately enclosing finish scope. The previous program is invalid, because in Line 3 we are registering a task with a phaser name `p` that is declared outside the finish.

```
p = newPhaser();
finish(
  async(p,
    adv(p);
    awaitAll; // await(p, 1);
  end);
end);
dereg(p);
end
```

Pipeline parallelism. The changes produced so far still allow us to write producer-consumer synchronisation patterns. We revisit Listing 3.3 by rewriting the programs of the producer and consumer tasks in SBrenner. The producer signals the consumers after producing an item with phaser p.

```
loop( // for (i = 0; i < N; i++)
  skip;            // B[i] = produce(i);
  adv(p);          // signal consumer
  next; end)       // loop
```

The consumer awaits consecutively for each signal from the producer before consuming the next element. Adapting the program of the observer tasks only amounts to replacing instruction `AWAIT(p)` with instruction `AWAIT`.

```
loop( // for (i = 0; i < N; i++)
  adv(p); await;   // await
  skip;            // consume(B[i]);
end)               // loop
```

Removing the possibility to await on an arbitrary phase hinders SBrenner's ability to perform the *bounded* producer-consumer pattern, important for pipeline parallelism. Shirako *et al.* explored in [96] the notion of *bounded phasers* to describe the bounded producer-synchronisation pattern. We adapt this idea to SBrenner under a deadlock-free setting. Instruction bound(p) lets a task be ahead of the *slowest* task up to a certain number of phases. We leave the exact bound number unspecified, as this detail does not affect our goal of showing deadlock freedom, and this way we avoid introducing natural numbers in the language.

We can now rewrite the producer to be ahead of the consumers up to a certain number of phasers. There are two changes we must do. First, the producer must

change its bound. Second, the producer must await after advancing. In practice, a producer that is at phase i and with a bound of k only blocks if there is at least one consumer behind phase $k - i$.

```
1 bound(n); // bound(n, SIZE); // set the bound to SIZE
2 loop(     // for (i = 0; i < N; i++)
3   skip;           // B[i] = produce(i);
4   adv(p);         // signal consumer
5   await;          // wait for slow consumers
6   next; end)      // loop
```

We underline that BRENNER is capable of writing the bounded producer-consumer pattern without introducing any notion of bound phasers. The required addition is to simply introduce arithmetic expressions over natural numbers. To await at a phase `i` with a bound `k` we write `AWAIT(p, i - k)`. This example highlights the expressive semantics of BRENNER when compared with the original semantics of phaser.

5.2 Syntax

Following is a discussion of the new terms of the language, with respect to BRENNER which we highlight using a box.

Definition 5.2.1 (Abstract syntax of SBRENNER)**.** *Fig. 5.1 defines the syntax of the language.*

Instruction `async` replaces instructions `reg`, `fork`, and `newTid`. We remove task names from the language to avoid potential deadlocks in fork/join and in phaser synchronisation. The parameter s of async specifies the phasers in which the forked task is to be registered with. Consider the following program written in BRENNER.

```
1 t = newTid();
2 reg(t, p1); reg(t, p2); reg(t, p3);
3 fork(t, b);
4 end
```

It can be translated into SBRENNER as follows.

Instruction p = `newPhaser()` creates a phaser. Instead of `await`(p, n) SBRENNER has instructions `await` and `bound`(p). Instruction `await` awaits on

b	::=	*Programs*
	| `end`	empty program
	| $i; b$	construct program
i	::=	*Instructions*
	| `async`(s, b)	fork the execution of a task
	| $p =$ `newPhaser()`	create a phaser
	| `dereg`(p)	deregister from phaser
	| `adv`(p)	advance phase
	| `bound`(p)	update bound
	| `await`	await on all phasers
	| `next`	enter next phase
	| `finish`(b)	join barrier
	| c	control the flow
c	::=	*Control flow*
	| `skip`	internal action
	| `loop`(b)	non-deterministic loop

Figure 5.1: SBrenner syntax.

```
1 async(p1, p2, p3, b);
2 end
```

every registered phaser at the task's local phase. For instance, a task that is registered with phaser `p` at phase `n`, with phaser `q` at phase `m`, and executes `await` corresponds, in Brenner, to a task that executes `AWAIT(p,n)` followed by `AWAIT(q,m)`. Awaiting on `q` followed by `p` produces the same effect. SBrenner lacks an instruction to await on phasers the task is not registered with.

Instruction `bound`(p) lets the task wait for a phase other than its local phase. The idea is that any task may wait on a smaller phase than its local phase without deadlocking. For each phaser p the task is registered with there is a bound m associated with p. The bound is a natural number and starts at zero. Consider a task that is registered with `p` has a local phase of `n` and a bound of `m`. Let `o` $=$ `n` $-$ `m`. Instruction `await` can be translated into Brenner as `AWAIT(p, o)`.

Listing 5.4: Matrix multiplication programmed in SBrenner.

```
1  finish(
2    loop( // for (i = 0; i < 21; i++)
3      async(
4        loop( // for (j = 0; j < 21; i++)
5          loop( // for (k = 0; k < 21; k++)
6            skip; // C[i][j] += A[i][k] * B[k][j];
7          end); // inner loop
8        end); // outer loop
9      end); // async
10     end); // loop
11   end); // finish
12 end // program
```

Instruction `next` marks all phasers as unarrived and represents the beginning of a new phase. Instruction `finish`(b) accepts a parameter called the *body* of the finish and declares a new phaser name scope.

Example

Running example: matrix multiplication, Listing 5.4. The program starts with a finish block, Lines 2 to 11, that awaits the computation of each row of the matrix. A loop in Line 2 spawns the tasks. The parallel computation of each row is done with an async, in Line 3, in which there is no more synchronisation. The main task sits waiting at the end of the finish, in Line 11, for the termination of each spawned task. The two main differences with reference to the Brenner version of the same example, in Listing 3.1, are: the async replaces the direct manipulation of tasks names, and the finish block replaces the phaser used to await the termination of spawned tasks.

Running example: iterative averaging, Listing 5.5. Phaser `p` is used as a cyclic barrier. The driver task creates a phaser `p`, executes the outer loop, in Lines 2 to 17, to spawn the worker tasks and then deregisters itself from `p`. The worker tasks are all registered with phaser `p` and synchronise together while performing the inner loop, in Lines 4 to 14. The two main differences with reference to the Brenner version of the same example, in Listing 3.2, are: using async instead of direct task name manipulation, and the inserting a `next` after synchronising.

Running example: producer-consumer, Listing 5.6. The driver task creates phaser `p` and launches two groups of tasks. The first group, spawned in Lines 2

Listing 5.5: Iterative averaging in SBrenner.

```
1  p = newPhaser(); // c = new Clock();
2  loop( // for (i = 0; i < N; i++)
3    async(p, // async clocked(c)
4      loop(     // for (k=1; k <= M; k++)
5        skip;     // l=P[(i-1) % N];r=P[(i+1) % N];
6        adv(p);   // c.resume();
7        skip;     // tmp = (l + r) / 2;
8        await;
9        next;     // c.advance();
10       skip;     // P[i] = tmp;
11       adv(p);
12       await;
13       next;     // c.advance();
14     end); // for
15     dereg(p);
16   end); // async
17 end); // for
18 dereg(p); // do not influence other tasks
19 end // program
```

to 12, consists of the producer tasks that synchronise with the consumers after producing an element. The second group, spawned in Lines 13 to 20, consists of the consumer tasks that await for all producer tasks to produce an event before consuming it. The main difference with reference to the Brenner version of the same example, in Listing 5.6, is that we introduced a *bounded* producer-consumer. Each producer task starts by setting its bound, in Line 4. This means that each producer can be ahead of the slowest task a certain number of phases when it invokes the await in Line 8. In Listing 3.3, the producer is *unbounded*, so the task advances without waiting for the consumers. The program in Listing 5.6 can be adapted to an unbounded producer by simply removing the await in Line 8.

5.3 Operational Semantics

We begin with name binding.

Definition 5.3.1 (Binding). *In* $p = \texttt{newPhaser}(); b$*, the displayed occurrence of phaser name* p *is a* binding *with* scope b*. An occurrence of a phaser name is* bound

Listing 5.6: Bounded producer-consumer synchronisation with phasers.

```
1  p = newPhaser(); // c = new Phaser();
2  loop( // producers
3    async(p,
4      bound(p);
5      loop( // for (i = 0; i < N; i++)
6        skip;        // B[i] = produce(i);
7        adv(p);      // signal consumer
8        await;
9        next;
10     end); // loop
11   end); // async
12 end);
13 loop( // consumers
14   async(p,
15     loop( // for (i = 0; i < N; i++)
16       adv(p); await(p);
17       skip;        // consume(B[i]);
18     end); // loop
19   end); // async
20 end);
21 dereg(p);
22 end // program
```

if it lies within the scope of a binding occurrence of the phaser name. Otherwise, the phaser name is free.

Bound phaser names can be computed by the next inductively defined function.

Definition 5.3.2 (Bound phaser names function)**.**

$$\begin{aligned}
\mathrm{bn}(\texttt{async}(s, b); b') &\stackrel{\text{def}}{=} s \cup \mathrm{bn}(b) \cup \mathrm{bn}(b') \\
\mathrm{bn}(p = \texttt{newPhaser}(); b) &\stackrel{\text{def}}{=} \{p\} \cup \mathrm{bn}(b) \\
\mathrm{bn}(\texttt{dereg}(p); b) &\stackrel{\text{def}}{=} \mathrm{bn}(\texttt{adv}(p); b) \stackrel{\text{def}}{=} \mathrm{bn}(\texttt{bound}(p); b) \stackrel{\text{def}}{=} \mathrm{bn}(b) \\
\mathrm{bn}(\texttt{await}; b) &\stackrel{\text{def}}{=} \mathrm{bn}(\texttt{next}; b) \stackrel{\text{def}}{=} \mathrm{bn}(b) \\
\mathrm{bn}(\texttt{finish}(b); b') &\stackrel{\text{def}}{=} \mathrm{bn}(b) \cup \mathrm{bn}(b') \\
\mathrm{bn}(\texttt{end}) &\stackrel{\text{def}}{=} \emptyset \\
\mathrm{bn}(\texttt{skip}; b) &\stackrel{\text{def}}{=} \mathrm{bn}(b) \\
\mathrm{bn}(\texttt{loop}(b); b') &\stackrel{\text{def}}{=} \mathrm{bn}(b) \cup \mathrm{bn}(b')
\end{aligned}$$

Free phaser names can be computed as:

Definition 5.3.3 (Free phaser names function)**.**

$$\begin{aligned}
\mathrm{fn}(\texttt{async}(s, b); b') &\stackrel{\text{def}}{=} p \cup \mathrm{fn}(b) \cup \mathrm{fn}(b') \\
\mathrm{fn}(p = \texttt{newPhaser}(); b) &\stackrel{\text{def}}{=} \mathrm{fn}(b) \setminus \{p\} \\
\mathrm{fn}(\texttt{dereg}(p); b) &\stackrel{\text{def}}{=} \mathrm{fn}(\texttt{adv}(p); b) \stackrel{\text{def}}{=} \mathrm{fn}(\texttt{bound}(p); b) \stackrel{\text{def}}{=} \mathrm{fn}(b) \\
\mathrm{fn}(\texttt{await}; b) &\stackrel{\text{def}}{=} \mathrm{fn}(\texttt{next}; b) \stackrel{\text{def}}{=} b \\
\mathrm{fn}(\texttt{finish}(b); b') &\stackrel{\text{def}}{=} \mathrm{fn}(b) \cup \mathrm{fn}(b') \\
\mathrm{fn}(\texttt{end}) &\stackrel{\text{def}}{=} \emptyset \\
\mathrm{fn}(\texttt{skip}; b) &\stackrel{\text{def}}{=} \mathrm{fn}(b) \\
\mathrm{fn}(\texttt{loop}(b); b') &\stackrel{\text{def}}{=} \mathrm{fn}(b) \cup \mathrm{fn}(b')
\end{aligned}$$

For example, in program

```
DEREG(p);
q = NEWPHASER();
ASYNC (q)
  DEREG(p);
  END;
END
```

phaser name p is free and phaser name q is bound.

A phaser name may occur both free and bound in the same expression. In the following example phaser name p appears bound and free:

```
DEREG(p);
p = NEWPHASER();
ASYNC(p) END
END
```

From top to bottom, the first displayed occurrence of phaser name p is free, at DEREG(p); the second and third displayed occurrences of p are bound, at p = NEWPHASER() and ASYNC(p). In the async the use of phaser name p targets the second occurrence and not the first.

Definition 5.3.4 (Substitution)**.** *A* substitution *for programs σ is a function that is the identity except on a finite set, defined from phaser names to phaser names. We write $[q/p]$ for the substitution σ such that $\sigma(p) = q$ and $\sigma(r) = r$ for $r \neq p$.*

A formula $b\sigma$ represents the application of substitution σ to program b, replacing each free occurrence of phaser name p in program b by $\sigma(p)$. We define the application of a substitution to a program (see the phaser name convention *below) as:*

$$
\begin{aligned}
(\texttt{async}(\{p_1,\ldots,p_n\},b);b')\sigma &\stackrel{\text{def}}{=} \texttt{async}(\{\sigma(p_1),\ldots,\sigma(p_n)\},(b\sigma));(b'\sigma)\\
(p=\texttt{newPhaser}();b)\sigma &\stackrel{\text{def}}{=} p=\texttt{newPhaser}();(b\sigma)\\
(\texttt{dereg}(p);b)\sigma &\stackrel{\text{def}}{=} \texttt{dereg}(\sigma(p));(b\sigma)\\
(\texttt{adv}(p);b)\sigma &\stackrel{\text{def}}{=} \texttt{adv}(\sigma(p));(b\sigma)\\
(\texttt{bound}(p);b)\sigma &\stackrel{\text{def}}{=} \texttt{bound}(\sigma(p));(b\sigma)\\
(\texttt{await};b)\sigma &\stackrel{\text{def}}{=} \texttt{await};(b\sigma)\\
(\texttt{next};b)\sigma &\stackrel{\text{def}}{=} \texttt{next};(b\sigma)\\
(\texttt{finish}(b);b')\sigma &\stackrel{\text{def}}{=} \texttt{finish}((b\sigma));(b'\sigma)\\
(\texttt{skip};b)\sigma &\stackrel{\text{def}}{=} \texttt{skip};(b\sigma)\\
(\texttt{loop}(b);b')\sigma &\stackrel{\text{def}}{=} \texttt{loop}((b\sigma));(b'\sigma)
\end{aligned}
$$

For example, substituting phaser name p for phaser name q in program

$\texttt{adv}(p); \texttt{await}; \texttt{dereg}(p); \texttt{end}$ yields the following result, where $\sigma = [q/p]$:

$$
\begin{aligned}
&(\texttt{async}(\{p, r\}, \texttt{await}; \texttt{dereg}(p); \texttt{end}); \texttt{end})\sigma \\
\stackrel{\text{def}}{=}&\texttt{async}(\{\sigma(p), r\}, (\texttt{await}; \texttt{dereg}(p); \texttt{end})\sigma); (\texttt{end}\sigma) \\
\stackrel{\text{def}}{=}&\texttt{async}(\{q, r\}, (\texttt{await}; \texttt{dereg}(p); \texttt{end})\sigma); (\texttt{end}\sigma) \\
\stackrel{\text{def}}{=}&\texttt{async}(\{q, r\}, \texttt{await}; (\texttt{dereg}(p); \texttt{end})\sigma); (\texttt{end}\sigma) \\
\stackrel{\text{def}}{=}&\texttt{async}(\{q, r\}, \texttt{await}; \texttt{dereg}(\sigma(p)); (\texttt{end}\sigma)); (\texttt{end}\sigma) \\
\stackrel{\text{def}}{=}&\texttt{async}(\{q, r\}, \texttt{await}; \texttt{dereg}(q); (\texttt{end}\sigma)); (\texttt{end}\sigma) \\
\stackrel{\text{def}}{=}&\texttt{async}(\{q, r\}, \texttt{await}; \texttt{dereg}(q); \texttt{end}); (\texttt{end}\sigma) \\
\stackrel{\text{def}}{=}&\texttt{async}(\{q, r\}, \texttt{await}; \texttt{dereg}(q); \texttt{end}); \texttt{end}
\end{aligned}
$$

Definition 5.3.5 (Change of bound phaser names)**.** *A change of bound names in program b is the replacement of a program that occurs in b of the form*

$$p = \texttt{newPhaser}(); b'$$

by $q = \texttt{newPhaser}(); b'[q/p]$, where q is not bound nor free in b'.

Definition 5.3.6 (α-convertibility)**.** *Programs b and b' are* α-convertible*, $b \equiv_\alpha b'$, if program b can be obtained from program b' by a finite number of changes of bound names.*

The two following expressions are α-convertible.

$$
\begin{aligned}
&\texttt{loop}(p = \texttt{newPhaser}(); \texttt{skip}; b\texttt{adv}(p); \texttt{end}); \texttt{end} \\
\equiv_\alpha&\texttt{loop}(q = \texttt{newPhaser}(); \texttt{skip}; b\texttt{adv}(q); \texttt{end}); \texttt{end}
\end{aligned}
$$

Phaser name convention. For any given mathematical context (*e.g.*, definition, proof), terms are taken up to α-convertibility and assume a convention (Barendregt's name convention [16]), in which all bound phaser names are chosen to be different from the free phaser names and also different from each other.

For example, program

$$\texttt{loop}((p = \texttt{newPhaser}(); \texttt{skip}; \texttt{adv}(p); \texttt{end})); \texttt{dereg}(p); \texttt{end}$$

breaks the convention, since the displayed occurrence p is both bound and free. The following α-convertible term conforms to the name convention:

$$\texttt{loop}((q = \texttt{newPhaser}(); \texttt{skip}; \texttt{adv}(q); \texttt{end})); \texttt{dereg}(p); \texttt{end}$$

Substitution for programs is only defined for terms respecting the name convention.

$$
\begin{array}{rcll}
S & ::= & \big(M, T\big) & \textit{Abstract machine} \\
M & ::= & \{p_1 \colon P_1, \ldots, p_n \colon P_n\} & \textit{Phaser maps} \\
T & ::= & \{t_1 \colon \tau_1, \ldots, t_n \colon \tau_n\} & \textit{Task maps} \\
P & ::= & \{t_1 \colon v_1, \ldots, t_n \colon v_n\} & \textit{Phaser value} \\
v & ::= & \langle p; a \rangle & \textit{Local view} \\
a & ::= & \mathbf{u} \mid \mathbf{a} & \textit{Flags} \\
\tau & ::= & & \textit{Tasks} \\
 & \mid & (B, b) & \text{regular task} \\
 & \mid & S \triangleright (B, b) & \text{finish task} \\
B & ::= & \{p \colon n, \ldots, p \colon n\} & \textit{Bounds}
\end{array}
$$

Figure 5.2: Syntax of the abstract machine.

Definition 5.3.7 (Abstract machine)**.** *Fig. 5.2 depicts the syntax of the state.*

The state S of a computation pairs a phaser map M and a task map T. A phaser map M stores the available phasers, mapping addresses to phasers. Phasers P map task names to local views v, that pair a wait phase n with an arrive flag a. Flag **a**, for arrived, denotes a task that advanced its phase. Flag **u**, for unarrived, denotes a task that can advance the phaser.

Task maps T hold tasks τ, named by task names t. There are two kinds of tasks. A regular task (B, b) holds a map of bounds B for each phaser the task is registered with, and also the program b it is executing. A finish task $S \triangleright (B, b)$ denotes a blocked regular task (B, b) that is waiting for the tasks in state S to conclude their execution.

The nested fork/join execution model can be represented as a tree of tasks in which leaf nodes can run concurrently and branch nodes wait for its children to terminate. A regular task is a leaf node. A finish task $\big(M, T\big) \triangleright (B, b)$ is a branch node and its children are the tasks in task map T. Fig. 5.3 illustrates such a tree: the root is finish task $\tau_1 \stackrel{\text{def}}{=} \big(M_1, \{t_2 \colon \tau_2, t_3 \colon \tau_3\}\big) \triangleright (B_1, b_1)$; tasks τ_2, τ_5, τ_6, and τ_7 run concurrently; tasks τ_1, τ_3, and τ_4 are blocked on a join barrier of a finish.

The reduction for states, $S \to S'$, allows for the non-deterministic choice of which tasks to evaluate.

Definition 5.3.8 (Small-step semantics)**.** *The small step reduction relation for* SBrenner *is defined in Figs. 5.4 to 5.6.*

Before explaining reduction rule R-ASYNC we require some auxiliary definitions.

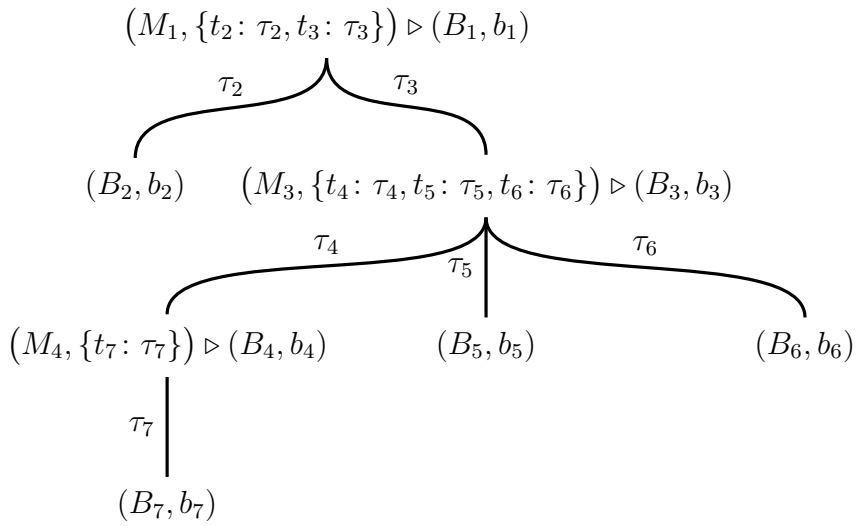


Figure 5.3: A dependency tree of tasks.

Definition 5.3.9 (Copy local views)**.**

$$\frac{\text{copy}(s, t, t', M) = M' \qquad P(t) = v}{\text{copy}(s \uplus \{p\}, t, t', M \uplus \{p\colon P\}) = M' \uplus \big\{p\colon P \uplus \{t'\colon v\}\big\}} \quad \text{(Cpy-cons)}$$

$$\frac{\text{copy}(s, t, t', M) = M' \qquad p \notin s}{\text{copy}(s, t, t', M \uplus \{p\colon P\}) = M' \uplus \{p\colon P\}} \quad \text{(Cpy-skip)}$$

$$\text{copy}(\emptyset, t, t', \emptyset) = \emptyset \quad \text{(Cpy-nil)}$$

For instance, in Line 3 of Listing 5.5, let the phaser map available in that state be $M_1 \stackrel{\text{def}}{=} \big\{p\colon P_1\big\}$, where $P_1 \stackrel{\text{def}}{=} \{t_d\colon (0, \mathbf{u})\}$.

We have that $\text{copy}(\{p\}, t_d, t_1, M_1) = M_2$, where

$$\frac{\dfrac{}{\text{copy}(\{p\}, t_d, t_1, \emptyset) = \emptyset}\ \textsc{Cpy-nil} \qquad P_1(p) = (0, \mathbf{u})}{\text{copy}(\{p\}, t_d, t_1, \{p\colon P_1\}) = \{p\colon P_1 \uplus \{t_1\colon (0, \mathbf{u})\}\}}\ \textsc{Cpy-cons}$$

Definition 5.3.10 (Bound creation)**.** $\text{bounds}(s) \stackrel{\text{def}}{=} \{p\colon 0 \mid \forall p \in s\}$

It is easy to see that $\text{bounds}(\{p\}) = \{p\colon 0\}$.

Rule R-Async governs the spawning of a new task that executes b' and is registered with every phaser $p \in s$. In SBrenner the spawned task inherits (copies) the local phase of the task that spawns when becoming registered

$$\begin{aligned}&\big(M, T \uplus \{t\colon (B, \texttt{async}(s, b'); b)\}\big)\\ \to&\big(\text{copy}(s, t, t', M), T \uplus \{t\colon (B, b)\} \uplus \{t'\colon (\text{bounds}(s), b')\}\big)\end{aligned} \quad \text{(R-ASYNC)}$$

$$\frac{q \notin \text{bn}(b)}{\begin{aligned}&\big(M, T \uplus \{t\colon (B, p = \texttt{newPhaser}(); b)\}\big)\\ \to&\big(M \uplus \{q\colon \{t\colon (0, \mathbf{u})\}\}, T \uplus \{t\colon (B \uplus \{q\colon 0\}, b[q/p])\}\big)\end{aligned}} \quad \text{(R-PHASER)}$$

$$\begin{aligned}&\big(M \uplus \{p\colon P \uplus \{t\colon v\}\}, T \uplus \{t\colon (B \uplus \{p\colon n\}, \texttt{dereg}(p); b)\}\big)\\ \to&\big(M \uplus \{p\colon P\}, T \uplus \{t\colon (B, b)\}\big)\end{aligned} \quad \text{(R-DEREG)}$$

$$\begin{aligned}&\big(M \uplus \{p\colon P \uplus \{t\colon (n, \mathbf{u})\}\}, T \uplus \{t\colon (B, \texttt{adv}(p); b)\}\big)\\ \to&\big(M \uplus \{p\colon P \uplus \{t\colon (n, \mathbf{a})\}\}, T \uplus \{t\colon (B, b)\}\big)\end{aligned} \quad \text{(R-ADVANCE)}$$

$$\frac{n \in \mathcal{N}}{\big(M, T \uplus \{t\colon (B \uplus \{p\colon \_\}, \texttt{bound}(p); b)\}\big) \to \big(M, T \uplus \{t\colon (B \uplus \{p\colon n\}, b)\}\big)} \quad \text{(R-BOUND)}$$

$$\frac{\text{awaitAll}(M, t, B)}{\big(M, T \uplus \{t\colon (B, \texttt{await}; b)\}\big) \to \big(M, T \uplus \{t\colon (B, b)\}\big)} \quad \text{(R-AWAIT)}$$

$$\big(M, T \uplus \{t\colon (B, \texttt{next}; b)\}\big) \to \big(\text{commit}(M, t), T \uplus \{t\colon (B, b)\}\big) \quad \text{(R-NEXT)}$$

Figure 5.4: Small step semantics for states (phaser related) $\boxed{S \to S}$.

with $p \in s$—this is performed by $\text{copy}(s, t, t', M)$. Function $\text{bounds}(s)$, used to build the forked task, sets the initial bound of each registered phaser to zero.

Hence, with rule R-ASYNC we get

$$\begin{aligned}&\big(M_1, T \uplus \{t_d\colon (B, \texttt{async}(\{p\}, b'); b)\}\big)\\ \to&\big(\text{copy}(\{p\}, t_d, t_1, M_1), T \uplus \{t_d\colon (B, b)\} \uplus \{t_1\colon (\text{bounds}(\{p\}), b')\}\big)\end{aligned}$$

And if $T = \emptyset$, we can simplify the state to

$$\big(M_1, \{t_d\colon (B, \texttt{async}(\{p\}, b'); b)\}\big) \to \big(M_2, \{t_d\colon (B, b), t_1\colon (\{p\colon 0\}, b')\big)$$

A task that executes $p = \texttt{newPhaser}()$ becomes registered with p and is able to advance this phaser. The reduction allocates a phaser with one participant, t, that starts at phase zero and is marked as unarrived; the bound for the new

phaser also starts at zero. We select a phaser name q that is not known in b and in the domain of M to ensure the locality of names, so that no other task can refer to it unless the phaser name is explicitly shared via async.

For example, at Line 1 in Listing 5.5, let the initial phaser map be empty and let it there a single task, named t_d, executing Listing 5.5. Let q be such that $q \notin \text{bn}(b)$. Thus,

$$\frac{q \notin \text{bn}(b)}{\begin{array}{l}\big(\emptyset, \{t_d\colon (\emptyset, p = \texttt{newPhaser}(); b)\big) \\ \rightarrow \big(\big\{q\colon \{t_d\colon (0, \mathbf{u})\}\big\}, \{t_d\colon (\{q\colon 0\}, b[q/p])\}\big)\end{array}}$$

Tasks deregister from phaser name p with expression $\texttt{dereg}(p)$. Consider Line 18 in Listing 5.5 and assume we have three tasks in the system: task named t_d executes Line 18, and tasks t_1 and t_2 are executing their loop. Let there only be a phaser in the system, named p, $M \stackrel{\text{def}}{=} \big\{p\colon \{t_1\colon v_1, t_2\colon v_2, t_d\colon v_d\}\big\}$. It is easy to see that there exists a phaser P such that $M(p) = P \uplus \{t_d\colon v_d\}$. Therefore,

$$\frac{}{\begin{array}{l}\big(\{p\colon P \uplus \{t_d\colon v_d\}\}, T \uplus \big\{t_d\colon (\{p\colon 0\}, \texttt{dereg}(p); b)\big\}\big) \\ \rightarrow \big(\{p\colon P\}, T \uplus \big\{t_d\colon (\emptyset, b)\big\}\big)\end{array}}\ \text{R-DEREG}$$

A task performing an $\texttt{adv}(p)$ simply turns the flag from **u** to arrived **a**, denoting it ready to synchronise (rule R-ADVANCE). Instruction $\texttt{bound}(p)$ is a non-deterministic choice of a new bound value, rule R-BOUND. Other tasks can observe the phase of p advancing. Assume that in Listing 5.5 we have two tasks, named t_1 and t_2, in parallel, both tasks executing Line 6. Let the state of the phasers be

$$M_1 \stackrel{\text{def}}{=} \big\{p\colon \{t_1\colon (0, \mathbf{u}), t_2\colon (0, \mathbf{u})\}\big\} \tag{5.1}$$

There exists a phaser P such that $M_1(p) = P \uplus \{t_2\colon (0, \mathbf{u})\}$. Hence,

$$\frac{}{\begin{array}{l}\big(\big\{p\colon P \uplus \{t_2\colon (0, \mathbf{u})\}\big\}, T \uplus \{t_2\colon (B, \texttt{adv}(p); b)\}\big) \\ \rightarrow \big(\big\{p\colon P \uplus \{t_2\colon (0, \mathbf{a})\}\big\}, T \uplus \{t_2\colon (B, b)\}\big)\end{array}}\ \text{R-ADV}$$

After the task named t_2 advances phaser p the state of the phasers is given by

$$M_2 \stackrel{\text{def}}{=} \big\{p\colon \{t_1\colon (0, \mathbf{u}), t_2\colon (0, \mathbf{a})\}\big\} \tag{5.2}$$

Definition 5.3.11 (Local phase)**.** *The* local phase *of a local view is computed as:*

$$\text{localPhase}\,(n, a) \stackrel{\text{def}}{=} \begin{cases} n+1 & \text{if } a = \mathbf{a} \\ n & \text{if } a = \mathbf{u} \end{cases}$$

The local phase of task t_1 for phaser p in M_2 is

$$\text{localPhase}\, M_2(p)(t_1) = 0$$

and of task t_2 is

$$\text{localPhase}\, M_2(p)(t_2) = 1$$

Definition 5.3.12 (Await predicate)**.**

$$\frac{\forall t \in \text{dom}\, P\colon\ \text{localPhase}\,(P(t)) \geq n}{\text{await}(P, n)}$$

The await predicate holds for phase 0 of phaser p

$$\text{await}(M_2(p), 0) \tag{5.3}$$

but it does *not* hold for

$$\text{await}(M_2(p), 1) \tag{5.4}$$

as localPhase $M_2(p)(t_1) < 1$.

The following predicate embodies `await`.

Definition 5.3.13.

$$\frac{\begin{array}{l}\forall p \in \text{dom}\, M \wedge t \in \text{dom}\, M(p)\colon \\ \text{await}(M(p), n) \wedge n = \text{localPhase}(M(p)(t)) - B(p)\end{array}}{\text{awaitAll}(M, t, B)}$$

Let $B = \{p\colon 0\}$. We know that $B(p) = 0$ and from Eq. (5.3) we also know that await$(M_2(p), 0)$, thus

$$\frac{\text{await}(M_2(p), 0) \qquad \text{localPhase}\, M_2(p)(t_1) = 0}{\text{awaitAll}(M_2, t_1, B)}$$

We also know that predicate awaitAll(M_2, t_2, B) does *not* hold, since

$$\text{localPhase}\, M_2(p)(t_2) = 1$$

and from Eq. (5.4) await$(M_2(p), 1)$ fails.

Rule R-AWAIT makes the task await at the local phase of every phaser it is registered with.

Consider the following phaser map

$$M_3 \stackrel{\text{def}}{=} \{p\colon \{t_1\colon (0, \mathbf{a}), t_2\colon (0, \mathbf{a})\}\} \tag{5.5}$$

as the state of Listing 5.5 while task t_1 executes Line 8. Also, let $B \stackrel{\text{def}}{=} \{p\colon n\}$. It is easy to see that for any bound n

$$\frac{\text{await}(M_3(p), 1-n) \qquad \text{localPhase}\, M_3(p)(t_1) = 1}{\text{awaitAll}(M_3, t_1, B)}$$

then

$$\frac{\text{awaitAll}(M_3, t_1, B)}{\big(M_3, T \uplus \{t_1\colon (B, \texttt{await}; b)\}\big) \to \big(M_3, T \uplus \{t_1\colon (B, b)\}\big)}\ \text{R-AWAIT}$$

Definition 5.3.14 (Commit phase)**.**

$$\frac{\text{commit}(M, t) = M'}{\text{commit}(M \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{a})\}\big\}, t) = M' \uplus \big\{p\colon P \uplus \{t\colon (n+1, \mathbf{u})\}\big\}}\ \text{(COM-C)}$$

$$\frac{\text{commit}(M, t) = M' \qquad t \notin \text{dom}\, P}{\text{commit}(M \uplus \{p\colon P\}, t) = M' \uplus \{p\colon P\}}\ \text{(COM-S)}$$

$$\text{commit}(\emptyset, t) = \emptyset\ \text{(COM-N)}$$

Using rules COM-N and COM-C we get that

$$\text{commit}(M_3, t_1) = \big\{p\colon \{t_1\colon (1, \mathbf{u}), t_2\colon (0, \mathbf{a})\}\big\} = M_4 \tag{5.6}$$

Function commit does not affect the local phase of any task, as

$$\text{localPhase}\, M_3(p)(t_1) = \text{localPhase}\, M_4(p)(t_1)$$

and

$$\text{localPhase}\, M_3(p)(t_2) = \text{localPhase}\, M_4(p)(t_2)$$

Rule R-NEXT governs the application of function commit.
Thus, from Eq. (5.6) we have that

$$\frac{}{\big(M, T \uplus \{t_1\colon (B, \texttt{next}; b)\}\big) \to \big(\text{commit}(M_3, t), T \uplus \{t_1\colon (B, b)\}\big)}\ \text{R-NEXT}$$

Reduction unrelated to phasers is included in Fig. 5.5. Rule R-FINISH declares that the body of the finish b' is executed by task t' that exists in a new state S.

We define the notions of halted state to identify when the finish barrier concludes.

$$\frac{S \stackrel{\text{def}}{=} \big(\emptyset, \{t' \colon (\emptyset, b')\}\big)}{\big(M, T \uplus \{t \colon (B, \texttt{finish}(b'); b)\}\big) \to \big(M, T \uplus \{t \colon S \triangleright (B, b)\}\big)} \quad \text{(R-FINISH)}$$

$$\frac{S_1 \to S_2}{\big(M, T \uplus \{t \colon S_1 \triangleright (B, b)\}\big) \to \big(M, T \uplus \{t \colon S_2 \triangleright (B, b)\}\big)} \quad \text{(R-RUN)}$$

$$\frac{S \text{ is halted}}{\big(M, T \uplus \{t \colon S \triangleright (B, b)\}\big) \to \big(M, T \uplus \{t \colon (B, b)\}\big)} \quad \text{(R-JOIN)}$$

$$\frac{c; b \to b'}{\big(M, T \uplus \{t \colon (B, c; b)\}\big) \to \big(M, T \uplus \{t \colon (B, b')\}\big)} \quad \text{(R-FLOW)}$$

Figure 5.5: Small step semantics for states (finish, control flow) $\boxed{S \to S}$.

$$\texttt{skip}; b \to b \quad \text{(R-SKIP)}$$

$$\texttt{loop}(b); b' \to b \cdot (\texttt{loop}(b); b') \quad \text{(R-ITER)}$$

$$\texttt{loop}(b); b' \to b' \quad \text{(R-ELIDE)}$$

Figure 5.6: Small step semantics for control flow instructions $\boxed{c; b \to b}$.

Definition 5.3.15 (Halted state)**.** *We say that state*

$$\big(M, \{t_1 \colon (\emptyset, \texttt{end}), \ldots, t_n \colon \langle \emptyset; \texttt{end} \rangle\}\big)$$

is halted.

Given a finish task $S \triangleright (B, b)$, we have that state S reduces until it becomes halted with rule R-RUN, and then the finish task becomes the regular task (B, b) with rule R-JOIN.

Rule R-FLOW governs the reduction of instructions that alter the control flow c. Rules in Fig. 5.6 are trivial.

5.4 Type System

A *type system* [23] is a formalism used to analyse the source code of a program according to a set of rules, called *typing rules*. The analysis abstracts each term of a program with respect to its *type*. For instance, the "integer" type can denote any mathematical expression that yields an integer. This way a type system can

$$\Gamma \vdash \emptyset : \emptyset \qquad \frac{\Gamma(p) = a \qquad \Gamma \vdash s : \Gamma'}{\Gamma \vdash (s \uplus \{p\}) : \Gamma' \uplus \{p : a\}} \qquad \text{(T-A-C,T-A-N)}$$

Figure 5.7: Typing rules for arguments $\boxed{\Gamma \vdash s : \Gamma}$.

abstract an expression $3 + 5 \times 2$ as an integer type. *Typechecking* is the process of verifying if a given source code conforms with some type information.

Type systems are used to establish properties about the programs of a language. In our case, the objective is to show the absence of deadlocked states for programs written in SBrenner. To this end, we specify a phaser usage policy by means of typing rules. Programs that adhere to this phaser usage policy are guaranteed to be deadlock free.

The operational semantics of SBrenner is not defined for all states. For instance, rule R-dereg expects the phaser name to be in the domain of the phaser map, and rule R-advance assumes the phaser to be unarrived. The reduction rules *assume* a certain policy and the semantics is undefined for any behaviour that breaks these assumptions, *i.e.,* the state will not be able to reduce. We define a typing system that rejects programs violating these assumptions.

Definition 5.4.1 (Type system)**.** *The type system for SBrenner is defined in Figs. 5.7 to 5.9.*

A typing Γ maps phaser names to arrival flags. The type system uses a typing to obtain (and record) the arrival flag for each phaser the task is registered with.

A *typing relation* $\Gamma \vdash s : \Gamma'$ assigns a typing Γ' to a term s given a typing Γ. On the left-hand side of the turnstile, we have the assumptions Γ under which the term s is checked; typing Γ' is inferred. Usually a typing relation yields a type (like an integer type). In our type system the outcome is a typing that represents the arrival flag of each *registered* phaser name. Judgement $\Gamma \vdash s : \Gamma'$ is used in the context of typing an async and yields the smallest typing Γ' that can typecheck s.

Typing relations are defined by cases, usually syntax-oriented on the term we are checking. Each case is covered by a *typing rule*. Fig. 5.7 consists of two rules: the base case T-A-N states that for an empty set of arguments we need the empty typing $\emptyset$. The inductive case T-A-C allows us to compose arguments s with distinct phaser names that are in the typing Γ. Similarly to reduction rules, typing rules can be axioms or have preconditions. Rule T-A-N is an axiom. Rule T-A-C has two preconditions, $\Gamma(p) = a$ and $\Gamma \vdash s : \Gamma'$.

To prove the validity of a type judgement (*i.e.,* whether it holds), we construct a *typing derivation* applying the typing rules. Let typing Γ be

$$\Gamma \stackrel{\text{def}}{=} \{p\colon \mathbf{u}, q\colon \mathbf{u}, r\colon \mathbf{a}\} \tag{5.7}$$

For example, typing judgement $\Gamma \vdash \{p\}\colon \{p\colon \mathbf{u}\}$ holds and its derivation tree (its proof) is

$$\dfrac{\dfrac{\dfrac{\Gamma(p) = \mathbf{u} \qquad \dfrac{}{\Gamma \vdash \emptyset\colon \emptyset}\,\textsc{T-a-n}}{\Gamma \vdash (\emptyset \uplus \{p\})\colon \emptyset \uplus \{p\colon \mathbf{u}\}}\,\textsc{T-a-c}}{\Gamma \vdash \{p\}\colon \{p\colon \mathbf{u}\}}\,\stackrel{\text{def}}{=}$$

Judgement $\Gamma \vdash \{q\}\colon \{q\colon \mathbf{a}\}$ does not hold, since $\Gamma(q) = \mathbf{u}$.

To type instructions we use $\Gamma \vdash i\colon \Gamma'$, defined in Fig. 5.8. Typing Γ' captures the effects on the arrival flags after the execution of an instruction. Most typing rules are straightforward. Rule T-ASYNC checks the forked instruction b under the assumptions inferred while typechecking s. This means that phaser names in s must all be registered, in typing Γ. Moreover, the spawned task must deregister from all phasers upon termination, so its effects are the empty typing. Rule T-PHASER ensures that phaser name p is unknown and initialises it as unarrived, thus in the effects we extend the typing with p assigned to $\mathbf{u}$. Similarly, rule T-DEREG removes the phaser name p from the typing of the effects to disallow further manipulation, as only phasers names in the typing Γ can be (de)registered and advanced. Rule T-ADV enforces that an advance on p is interleaved by a next. The effects of advancing p is marking it as arrived. Instruction `bound`(p) leaves the arrival flags of the phaser names unaltered, rule T-BOUND. Tasks can only await after advancing all registered phasers, rule T-AWAIT. This instruction produces no effect on the state of the phasers. Rule T-NEXT ensures all registered phasers are advanced and then marks these as unarrived. Instruction `skip` has no side effects, rule T-SKIP. Finally, rule T-LOOP states that the body of the loop must preserve the arrival flags of the registered phasers, plus it must not deregister from any phaser name in Γ.

To type programs we use $\Gamma \vdash i\colon \Gamma'$, defined in Fig. 5.8. The rules ensure that the effects of an instruction are enough to type its continuation, *cf.* rules T-CONS and T-END.

To summarise, the type system enforces four rules:

1. a task can only (de)register and advance phasers it is registered with, *cf.* rules T-PHASER, T-DEREG, T-ADV, T-ASYNC;

2. every registered phaser is advanced exactly once before an invocation of `next`, *cf.* rules T-ADV and T-NEXT;

$$\frac{\Gamma \vdash s\colon \Gamma' \qquad \Gamma' \vdash b\colon \emptyset}{\Gamma \vdash \texttt{async}(s, b)\colon \Gamma} \quad \text{(T-ASYNC)}$$

$$\Gamma \vdash p = \texttt{newPhaser}()\colon \Gamma \uplus \{p\colon \mathbf{u}\} \quad \text{(T-PHASER)}$$

$$\Gamma \uplus \{p\colon a\} \vdash \texttt{dereg}(p)\colon \Gamma \quad \text{(T-DEREG)}$$

$$\Gamma \uplus \{p\colon \mathbf{u}\} \vdash \texttt{adv}(p)\colon \Gamma \uplus \{p\colon \mathbf{a}\} \quad \text{(T-ADV)}$$

$$\Gamma \vdash \texttt{bound}(p)\colon \Gamma \quad \text{(T-BOUND)}$$

$$\frac{\forall p \in \operatorname{dom}\Gamma\colon \Gamma(p) = \mathbf{a}}{\Gamma \vdash \texttt{await}\colon \Gamma} \quad \text{(T-AWAIT)}$$

$$\{p_1\colon \mathbf{a}, \ldots, p_n\colon \mathbf{a}\} \vdash \texttt{next}\colon \{p_1\colon \mathbf{u}, \ldots, p_n\colon \mathbf{u}\} \quad \text{(T-NEXT)}$$

$$\frac{\emptyset \vdash b\colon \emptyset}{\Gamma \vdash \texttt{finish}(b)\colon \Gamma} \quad \text{(T-FINISH)}$$

$$\Gamma \vdash \texttt{skip}\colon \Gamma \quad \text{(T-SKIP)}$$

$$\frac{\Gamma \vdash b\colon \Gamma}{\Gamma \vdash \texttt{loop}(b)\colon \Gamma} \quad \text{(T-LOOP)}$$

Figure 5.8: Typing rules for instructions $\boxed{\Gamma \vdash i\colon \Gamma}$.

$$\frac{\Gamma \vdash i\colon \Gamma' \qquad \Gamma' \vdash b\colon \Gamma''}{\Gamma \vdash i; b\colon \Gamma''} \quad \text{(T-CONS)}$$

$$\Gamma \vdash \texttt{end}\colon \Gamma \quad \text{(T-END)}$$

Figure 5.9: Typing rules for programs $\boxed{\Gamma \vdash b\colon \Gamma}$.

3. the body of a finish and of forked tasks must deregister from all phasers before terminating, *cf.* rules T-ASYNC and T-FINISH;

4. each iteration of a loop must terminate registered with the same phasers it starts registered with, *cf.* T-LOOP.

We give an example for each of these four rules.

Rule 1. An empty typing $\emptyset$ means that there are no registered phasers. The only phaser-related instruction that can be typed (*i.e.,* checked) is phaser creation.

$$\frac{}{\emptyset \vdash p = \texttt{newPhaser}() \colon \{p \colon \mathbf{u}\}}\ \text{T-PHASER} \tag{5.8}$$

Let $\Gamma_1 \stackrel{\text{def}}{=} \{p \colon \mathbf{u}\}$ and $\Gamma_2 \stackrel{\text{def}}{=} \{p \colon \mathbf{a}\}$. The effects of creating a phaser are enough to type an advance.

$$\frac{\dfrac{}{\Gamma_1 \vdash \texttt{adv}(p) \colon \{p \colon \mathbf{a}\}}\ \text{T-ADV} \qquad \dfrac{}{\Gamma_2 \vdash \texttt{end} \colon \Gamma_2}\ \text{T-END}}{\Gamma_1 \vdash \texttt{adv}(p); \texttt{end} \colon \Gamma_2}\ \text{T-CONS}$$

And therefore,

$$\frac{\emptyset \vdash p = \texttt{newPhaser}() \colon \Gamma_1 \qquad \Gamma_1 \vdash \texttt{adv}(p); \texttt{end} \colon \Gamma_2}{\emptyset \vdash p = \texttt{newPhaser}(); \texttt{adv}(p); \texttt{end} \colon \Gamma_2}\ \text{T-CONS}$$

Rule 2. Typing Γ_1 cannot be used to type a next, yet the outcome of advancing p, that is typing Γ_2, can type instruction `next`.

$$\frac{\dfrac{}{\{p \colon \mathbf{a}\} \vdash \texttt{next} \colon \{p \colon \mathbf{u}\}}\ \text{T-NEXT}}{\Gamma_2 \vdash \texttt{next} \colon \Gamma_1}\ \stackrel{\text{def}}{=}$$

Rule 3. The following tree holds.

$$\frac{\dfrac{}{\Gamma_1 \vdash \texttt{dereg}(p) \colon \emptyset}\ \text{T-DEREG} \qquad \dfrac{}{\emptyset \vdash \texttt{end} \colon \emptyset}\ \text{T-END}}{\Gamma_1 \vdash \texttt{dereg}(p); \texttt{end} \colon \emptyset}\ \text{T-CONS}$$

And we also know that

$$\frac{\dfrac{\dfrac{\Gamma_1(p) = \mathbf{a} \qquad \dfrac{}{\Gamma_1 \vdash \emptyset \colon \emptyset}\ \text{T-A-N}}{\Gamma_1 \vdash (\emptyset \uplus \{p\}) \colon \emptyset \uplus \{p \colon \mathbf{a}\}}\ \text{T-A-C}}{\Gamma_1 \vdash \{p\} \colon \Gamma_1}\ \stackrel{\text{def}}{=}$$

Thus,

$$\frac{\Gamma_1 \vdash \{p\} : \Gamma_1 \qquad \Gamma_1 \vdash \texttt{dereg}(p); \texttt{end} : \emptyset}{\Gamma_1 \vdash \texttt{async}(\{p\}, \texttt{dereg}(p); \texttt{end}) : \Gamma_1}\ \text{T-ASYNC}$$

Rule 4. Enforcing that the iteration must terminate registered with the same phasers it starts registered with does not disallow phaser creation inside the loops. For instance, the following tree holds.

$$\frac{\dfrac{\emptyset \vdash p = \texttt{newPhaser}() : \Gamma_1 \qquad \Gamma_1 \vdash \texttt{dereg}(p); \texttt{end} : \emptyset}{\emptyset \vdash p = \texttt{newPhaser}(); \texttt{dereg}(p); \texttt{end} : \emptyset}\ \text{T-CONS}}{\emptyset \vdash \texttt{loop}((p = \texttt{newPhaser}(); \texttt{dereg}(p); \texttt{end})) : \emptyset}\ \text{T-LOOP}$$

Chapter Six

Type system properties

The type system for instructions represents a specification on a phaser usage we deem as valid. Applying the typing relation to a program corresponds to checking if the program conforms with this phaser usage. Yet, to reason about the semantics of SBrenner we need to establish a relation between a program b and its state S.

Let $S_1 \stackrel{\text{def}}{=} \big(M, \{t\colon (\emptyset, \texttt{end})\}\big)$ and $S_2 \stackrel{\text{def}}{=} \big(\emptyset, \{t\colon (B, \texttt{dereg}(p); \texttt{end})\}\big)$. Neither of these states can reduce. State S_1 is halted, there are no reduction rules for program end. State S_2 cannot reduce because phaser name p is not in the domain of phaser map $\emptyset$, w.r.t rule R-DEREG. We can distinguish between these two states in terms of validity (next, we define this notion precisely with a type system for states). State S_1 is valid because it is not doing anything unexpected, there is not a rule for program end because terminated tasks should remain halted. State S_2 is invalid because a task is manipulating an unknown phaser name. A type system enjoys the property of *Subject Reduction* if the reduction relation preserves validity. A type system enjoys the property of *Progress* if any valid state either reduces or is halted, which implies the absence of deadlocks.

In this chapter we build a type system for states that enjoys the properties of subject reduction and progress. This requires the definition of a type system for phaser maps in Section 6.1, and another for task maps in Section 6.2. We conclude the chapter establishing some basic results in Sections 6.3 to 6.5.

6.1 Typing phaser maps

The operational semantics of SBrenner lays some assumptions not only about a phaser usage, but also on the configuration of phaser maps. In particular, the semantics expects the absence of dangling task names. For instance, the following state cannot reduce because task name t' is not assigned to any task.

$$\big(\{p\colon \{t\colon (1, \mathbf{a}), t'\colon (0, \mathbf{a})\}\}, \{t\colon (\{p\colon 0\}, \texttt{await}; b)\}\big)$$

A type system for phaser maps that enjoys progress (*i.e.,* valid states must reduce or be halted) must rule out such ill-formed phaser map.

The operational semantics also expects the phasers in the phaser map to be the outcome of the instructions, not an arbitrary phaser map. The next state cannot reduce.

$$\big(\{p\colon \{t\colon (1,\mathbf{a}), t'\colon (0,\mathbf{a})\}, q\colon \{t\colon (1,\mathbf{a}), t'\colon (2,\mathbf{a})\}\},$$
$$\{t\colon (\{p\colon 0, q\colon 0\}, \texttt{await}; b), t'\colon (\{p\colon 0, q\colon 0\}, \texttt{await}; b')\}\big)$$

Tasks t and t' are deadlocked. Task t requires task t' to advance phaser p and, at the same time, task t' requires task t to advance phaser q. Task t has a local phase of 2 for phaser p and a local phase of 2 for phaser q. We can say that for t the local phase difference between p and q is zero, as $2 - 2 = 0$. Similarly, for t' the local phase difference between p and q is minus two, as $1 - 3 = -2$. Following, we explain why is it that a program that follows the phaser usage of SBrenner cannot reach a state that contains a phaser map with disparate phase differences.

Our language restricts tasks to advance their registered phasers stepwise. This means that for any two phasers, whenever a task executes a `next` their relative local phase difference stay the same. Say task t creates phasers p and q one after the other, it advances both phasers, and then issues a next. The local phase difference between p and q while executing the `next` is zero, as task t has a local phase of one for both phasers. That is, there exists a phaser map M such that $M(p)(t) = (1, \mathbf{a})$, $M(q)(t) = (1, \mathbf{a})$, and $1 - 1 = 0$. During the lifetime of the task named t, and while it is registered with phasers p and q, the local phase difference between both of these phasers will remain zero whenever it executes a `next`. The type system for instruction will reject any program that tries otherwise, *e.g.*, advancing p more than once before of a next.

If every task maintains the local phase difference between any pair of phasers, and since phasers can only be shared by spawning tasks, then the spawned tasks have "inherit" the local phase differences of their parent tasks. Let task t have a local phase difference between phasers p and q of zero. If it spawns task t' and registers t' with p and q, then task t' also possess a local phase difference of zero between p and q.

When the local phase difference property is respected for all task names in a phaser map, then we can establish an ordering between task names. We introduce the notion of *supersteps*, borrowed from the Bulk-Synchronous Parallel model. In this programming model, there is a single, global barrier for all tasks to synchronise, so all tasks proceed stepwise, or in supersteps. In SBrenner, a group of tasks that share at least one phaser synchronises together. There are two differences, with respect to the Bulk-Synchronous Parallel model. First, multiple groups can be defined each with their own "global" barrier. Second, tasks can be executing in different supersteps, because waiting is optional.

Given a phaser map M, a task t is one superstep ahead of another t' if for every phaser p in which both tasks are registered we have that $M(p)(t) = (n+1, \_)$ and $M(p)(t') = (n, \_)$. Let z be an integer, and operations $=$, $+$, and $-$ be the usual equality, addition, and subtraction on integers, respectively. The superstep difference Δ is a map from pairs of task names (t_1, t_2) to integers $\mathcal{Z}$.

Definition 6.1.1. *Let* $(N, \leq_\Delta)$ *be defined as*

$$\forall t_1, t_2 \in N \colon \Delta(t_1, t_2) \in \mathcal{Z}_{\leq 0} \iff t_1 \leq_\Delta t_2$$

Definition 6.1.2. *Let* $(N, =_\Delta)$ *be defined as*

$$\forall t_1, t_2 \in N \colon \Delta(t_1, t_2) = 0 \iff t_1 =_\Delta t_2$$

Definition 6.1.3 (Total ordering)**.** *The relation structure* $(N, \leq_\Delta)$ *is a total ordering if, and only if, it is*

reflexive $\forall t \in N \colon t \leq_\Delta t$

transitive $\forall t_1, t_2, t_3 \in N \colon t_1 \leq_\Delta t_2 \wedge t_2 \leq_\Delta t_3 \implies t_1 \leq_\Delta t_3$

anti-symmetric $\forall t_1, t_2 \in N \colon t_1 \leq_\Delta t_2 \wedge t_2 \leq_\Delta t_1 \implies t_1 =_\Delta t_2$

compatible $\forall t_1, t_2 \in N \colon t_1 \leq_\Delta t_2 \vee t_2 \leq_\Delta t_1$

Definition 6.1.4 (Type system for phaser maps)**.** *The type system for phaser maps is defined in Fig. 6.1.*

We are only interested in superstep differences Δ that are well-formed under task names N, notation $N \vdash \Delta$, that is superstep differences where the relation structure $(N, \leq_\Delta)$ forms a total ordering, rule D-WF.

For example,

- any difference map Δ is well formed for the empty set, $\emptyset \vdash \Delta$;
- the difference map $\Delta_1 \stackrel{\text{def}}{=} \{(t, t) \colon 0\}$ is well formed for $\{t\}$, so we have $\{t\} \vdash \Delta_1$;
- the difference map $\Delta_2 \stackrel{\text{def}}{=} \{(t, t) \colon 4\}$ is ill formed for $\{t\}$ because $\Delta(t, t) \neq -\Delta(t, t)$;
- let Δ_3 be such that $\Delta_3(t, t) = \Delta_3(t', t') = 0$, $\Delta_3(t, t') = 1$, $\Delta_3(t', t) = -1$, then this difference map is well formed for $\{t, t'\}$, or $\{t, t'\} \vdash \Delta_3$;

Well-formed phase difference $\boxed{N \vdash \Delta}$:

$$\frac{(N, \leq_\Delta) \text{ is a total ordering} \quad \forall t_1, t_2 \in N : \Delta(t_1, t_2) = z \implies \Delta(t_2, t_1) = -z}{N \vdash \Delta} \qquad \text{(D-WF)}$$

Phase difference for labels $\boxed{\Delta; t; n \vdash P}$:

$$\frac{\Delta; t_1; n_1 \vdash P \qquad \Delta(t_1, t_2) = (n_1 - n_2)}{\Delta; t_1; n_1 \vdash P \uplus \{t_2 : \langle n_2; \_ \rangle\}} \qquad \Delta; t; n \vdash \emptyset \quad \text{(D-L-CONS,D-L-NIL)}$$

Typing rules for phasers $\boxed{\Delta \vdash P}$:

$$\frac{\Delta \vdash P \qquad \Delta; t; n \vdash P}{\Delta \vdash P \uplus \{t : \langle n; \_ \rangle\}} \qquad \Delta \vdash \emptyset \qquad \text{(D-PH-CONS,D-PH-NIL)}$$

Typing rules for phaser maps $\boxed{\Delta; N \vdash M}$:

$$\frac{\Delta \vdash P \qquad \operatorname{dom} P \subseteq N \qquad \Delta; N \vdash M}{\Delta; N \vdash M \uplus \{p : P\}} \qquad \Delta; N \vdash \emptyset$$

(T-P-MAP-CONS,T-P-MAP-NIL)

Figure 6.1: Typing rules for phasers and phaser maps.

- the difference map $\Delta_4 \stackrel{\text{def}}{=} \{(t, t') : 4\}$ is ill formed for $\{t, t'\}$ since $\Delta_4(t', t)$ is undefined;
- the difference map $\{(t, t') : 0\}$ is ill formed for $\{t, t'\}$ because $\Delta_1(t', t)$ is undefined.

Judgement $\Delta; t; n \vdash P$ checks whether the superstep difference in P matches the one in Δ between task name t and every $t' \in \operatorname{dom} P$ (rule D-L-CONS). Judgement $\Delta \vdash P$ verifies if the superstep differences of the task names in P respect the ones in Δ, by picking each task name in $\operatorname{dom} P$ and comparing it the others in P (rule D-PH-CONS). Judgement $\Delta \vdash M$ checks if the superstep differences in M respect the ones in Δ, by checking each phaser independently (rule D-PM-CONS).

To summarise, the type system enforces two rules:

1. there are no dangling task names mentioned in the domain of each phaser of the phaser map, with respect to a given set N;
2. the phaser map respects a given difference map Δ and a set N.

For example, let $N = \{t_1, t_2, t_3\}$, $N \vdash \Delta$, $\Delta(t_1, t_3) = 2$, and $\Delta(t_3, t_2) = -1$. Phaser $\{t_1 \colon (4, \mathbf{u}), t_2 \colon (3, \mathbf{a})\}$ is well-typed under superstep difference Δ.

$$\dfrac{\dfrac{}{\Delta \vdash \emptyset}\,\text{D-PH-NIL} \quad \dfrac{}{\Delta; t_2; 3 \vdash \emptyset}\,\text{D-L-NIL}}{\dfrac{\Delta \vdash \{t_2 \colon (3, \mathbf{a})\} \qquad \Delta; t_1; 4 \vdash \{l_2 \colon (3, \mathbf{a})\}}{\Delta \vdash \{t_1 \colon (4, \mathbf{u}), t_2 \colon (3, \mathbf{a})\}}\,\text{D-PH-CONS}}\,\text{D-PH-CONS}$$

where

$$\dfrac{\dfrac{}{\Delta; t_2; 3 \vdash \emptyset}\,\text{D-L-NIL} \qquad \dfrac{\Delta(t_1, t_3) = 2 \qquad \Delta(t_3, t_2) = -1}{\Delta(t_1, t_2) = 4 - 3}\,\text{transitivity}}{\Delta; t_1; 4 \vdash \{t_2 \colon (3, \mathbf{a})\}}\,\text{D-L-CONS}$$

6.2 Typing states

For typing task maps, there are three ill-formed task map configurations to consider. First, that the bounds of each task match exactly the registered phasers in the phaser map of the state. For example, the state

$$\big(\{p \colon \{t \colon (0, \mathbf{a})\}\}, \{t \colon (\emptyset, \texttt{await}; b)\}\big)$$

cannot reduce because by rule R-AWAIT and Definition 5.3.13 phaser name p must be in the domain of the bounds $\emptyset$ of task t, and we have $p \notin \emptyset$.

Second, that the program of each task mentions phaser names that are in the phaser map of the state. The next state cannot reduce because task t is trying to advance a non-existent phaser named p.

$$\big(\emptyset, \{t \colon (\emptyset, \texttt{adv}(p); b)\}\big)$$

Third, that each tasks deregisters from all of its phasers upon completing so as to avoid the creation of dangling task names. The state

$$\big(\{p \colon \{t \colon (1, \mathbf{a}), t' \colon (0, \mathbf{a})\}\}, \{t \colon (\{p \colon 0\}, \texttt{await}; b), t' \colon (\{p \colon 0\}, \texttt{end})\}\big)$$

cannot reduce because t' terminated without deregistering from phaser p.

Definition 6.2.1 (Typing rules for task maps)**.** *The typing rules for task maps is defined in Fig. 6.2.*

Judgement $\vdash_t M \colon \Gamma$ assigns a typing Γ to a phaser map M given a task name t. The typing is constructed in such a way that its domain contains the

Typing rules for activity permissions $\boxed{\vdash_t M : \Gamma}$:

$$\frac{\vdash_t M : \Gamma \qquad t \notin \operatorname{dom} P}{\vdash_t M \uplus \{p : P\} : \Gamma} \qquad \vdash_t \emptyset : \emptyset \qquad \text{(T-PERM-SKIP,T-PERM-NIL)}$$

$$\frac{\vdash_t M : \Gamma \qquad P(t) = (\_, a)}{\vdash_t M \uplus \{p : P\} : \Gamma \uplus \{p : a\}} \qquad \text{(T-PERM-CONS)}$$

Typing rules for bounds $\boxed{\Gamma \vdash B}$:

$$\frac{\Gamma \vdash B}{\Gamma \uplus \{p : a\} \vdash B \uplus \{p : n\}} \qquad \emptyset \vdash \emptyset \qquad \text{(T-B-C,T-B-N)}$$

Typing rules for tasks $\boxed{\Psi; \Gamma \vdash \tau}$:

$$\frac{\Gamma \vdash B \qquad \Gamma \vdash b : \emptyset}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, b)} \qquad \frac{\Psi \vdash S \qquad \langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, b)}{\Psi; \Gamma \vdash S \triangleright (B, b)} \qquad \text{(T-T-R,T-T-F)}$$

Typing rules for task maps $\boxed{\Sigma; M \vdash T}$:

$$\frac{\vdash_t M : \Gamma \qquad \Psi; \Gamma \vdash \tau \qquad \Sigma; M \vdash T}{\Sigma \uplus \{t : \Psi\}; M \vdash T \uplus \{t : \tau\}} \qquad \emptyset; M \vdash \emptyset \qquad \text{(T-TM-C,T-TM-N)}$$

Figure 6.2: Typing rules for permissions, bounds, tasks, and task maps.

phaser names in which task name t is registered. For example, we have that the next derivation holds.

$$\frac{\dfrac{}{\vdash_t \emptyset : \emptyset}\ \text{T-PERM-NIL}}{\vdash_t \{p : \{t : (0, \mathbf{a})\}\} : \{p : \mathbf{a}\}}\ \text{T-PERM-CONS} \tag{6.1}$$

Judgement $\Gamma \vdash B$ checks if the domain of typing Γ matches the domain of bounds B. For example, bounds $\{p : 0\}$ are well typed under typing $\{p : \mathbf{a}\}$.

$$\frac{\dfrac{}{\emptyset \vdash \emptyset}\ \text{T-B-N}}{\{p : \mathbf{a}\} \vdash \{p : 0\}}\ \text{T-B-C} \tag{6.2}$$

A (tree) node of differences $\Psi ::= \langle \Delta; \Sigma \rangle$ pairs a map of children Σ and map of differences Δ. A Σ maps task names to nodes Ψ. Judgement $\Psi; \Gamma \vdash \tau$ types a task under a node of differences Ψ and a typing Γ. Typing a finish task requires

Typing rules for abstract machines $\boxed{\Psi \vdash S}$:

$$\frac{\operatorname{dom} T \vdash \Delta \qquad \Delta; \operatorname{dom} T \vdash M \qquad \Sigma; M \vdash T}{\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)} \qquad \text{(T-AMACH)}$$

Figure 6.3: Typing rules for states.

a node of differences Ψ to be able to check a nested states (rule T-T-F), otherwise the node of differences must be empty $\langle \emptyset; \emptyset \rangle$. The type system checks whether typing Γ is enough to check the bounds B and the program b of regular tasks, rule T-T-R. Furthermore, the outcome of typing b is an empty typing, which means the task must deregister from every phaser before terminating.

Judgement $\Sigma; M \vdash T$ types a task map T under a map of nodes Σ and a phaser map M. The domain of Σ and of T must be equal, meaning that for each task τ there is a node of differences Ψ, rule T-TM-C. For each task τ named t, the type system checks task τ under its registered phasers Γ and node differences Ψ.

To summarise, the type system enforces three rules:

1. the phasers in which task t is registered with (by inspecting the phaser map) equals the ones in the task's bounds, rules T-B-C and T-TM-C;
2. the free phaser names of any tasks' instructions is registered phasers, rules T-T-R and T-PERM-CONS;
3. any task that terminates is not mentioned in the phaser map, rule T-T-R.

Definition 6.2.2 (Typing rules for states)**.** *The typing rules for task maps is defined in Fig. 6.3.*

Judgement $\Psi \vdash S$ types a state S under a node of differences Ψ. Given a node of differences $\langle \Delta; \Sigma \rangle$, the type system uses the map of differences Δ and the domain of the task maps T to type the phaser map M, thus it ensures the absence of dangling task names and that the phaser map respects the phase differences in Δ. The map of tasks T is typed under the map of nodes Σ and the phaser map M, meaning that for each task named t in the task map $T(t)$ there must be a node $\Sigma(t) = \Psi$.

6.3 Inversion

Inversion lemmas serve as the cornerstone for many of the results we establish. These are, nonetheless, an idiosyncrasy of our choice to represent composed

structures with maps (and sets). The gist behind these results are: given a typing relation for a map (like a task map) and a member of that map we can deconstruct the map obtain the typing relations of its constituents. For example, given a well-typed task map T and a member $T(t) = \tau$, then we show that there exists a task map T' such that $T = T' \uplus \{t : \tau\}$, task map T' is well typed, and τ is also well typed. The proofs for these results are uninteresting and follow by induction on the typing relation. In the remainder of the section we establish the inversion property for typing arguments, typing task maps, inferring the typing context of a phaser map, and typing the differences of a phaser.

Lemma 6.3.1. *If $\Gamma \vdash s : \Gamma'$ and $p \in s$, then there exist some arguments s' and a typing Γ'' such that $s = s' \uplus \{p\}$, $\Gamma' = \Gamma'' \uplus \{p : a\}$, $\Gamma(p) = a$, $\Gamma \vdash s' : \Gamma''$.*

Proof. The proof follows by induction on the typing relation. We perform an inversion of $\Gamma \vdash s : \Gamma'$ and a case analysis on the derivation.

- Case T-A-N

$$\Gamma \vdash \emptyset : \emptyset$$

  We reach a contradiction since we have that $p \in \emptyset$.

- Case T-A-C:

$$\frac{\Gamma(q) = a' \qquad \Gamma \vdash s_1 : \Gamma_1}{\Gamma \vdash (s_1 \uplus \{q\}) : \Gamma_1 \uplus \{q : a'\}}$$

  where s is $s_1 \uplus \{q\}$ and Γ' is $\Gamma_1 \uplus \{q : a\}$. If $p = q$, the case holds. Otherwise, $p \neq q$ and therefore $p \in s_1$. Applying the induction hypothesis to $\Gamma \vdash s_1 : \Gamma_1$ and $p \in s_1$ we get that there exist some arguments s_2 and a typing Γ_2 such that $s_1 = s_2 \uplus \{p\}$, $\Gamma_1 = \Gamma_2 \uplus \{p : a\}$, (i) $\Gamma(p) = a$, and $\Gamma \vdash s_2 : \Gamma_2$. We have that $\Gamma'' \stackrel{\text{def}}{=} \Gamma_2 \uplus \{q : a\}$ and $s' \stackrel{\text{def}}{=} s_2 \uplus \{q\}$.

  Hence,

$$\dfrac{\dfrac{\text{(i) } \Gamma(q) = a' \qquad \Gamma \vdash s_2 : \Gamma_2}{\Gamma \vdash (s_2 \uplus \{q\}) : \Gamma_2 \uplus \{q : a'\}}\ \text{T-A-C}}{\Gamma \vdash s' : \Gamma''}\ \stackrel{\text{def}}{=}$$

□

Lemma 6.3.2. *If $\Sigma_1; M \vdash T_1$ and $T_1(t) = \tau$, then*

1. $\Sigma_1 = \Sigma_2 \uplus \{t : \Psi\}$,
2. $T_1 = T_2 \uplus \{t : \tau\}$,
3. $\vdash_t M : \Gamma$,

4. $\Psi; \Gamma \vdash \tau$, *and*

5. $\Sigma_2; M \vdash T_2$.

Proof. The proof follows by induction on the typing relation. We perform an inversion of $\Sigma_1; M \vdash T$ and a case analysis on the derivation.

- Case T-TM-N:

$$\emptyset; M \vdash \emptyset$$

  where Σ_1 is $\emptyset$ and T is $\emptyset$. We have that $t \in \operatorname{dom} T$, yet $T \stackrel{\text{def}}{=} \emptyset$, so this case does not apply.

- Case T-TM-C:

$$\frac{\text{(i) } \vdash_{t_1} M : \Gamma_1 \qquad \text{(ii) } \Psi_1; \Gamma_1 \vdash \tau_1 \qquad \text{(iii) } \Sigma_3; M \vdash T_3}{\Sigma_3 \uplus \{t_1 : \Psi_1\}; M \vdash T_3 \uplus \{t_1 : \tau_1\}}$$

  where Σ_1 is $\Sigma_3 \uplus \{t_1 : \Psi_1\}$ and T_1 is $T_3 \uplus \{t_1 : \tau_1\}$. If $t = t_1$, we are done. Otherwise, we know that $t \neq t_1$. Hence, $t \in \operatorname{dom} T_3$ and by the induction hypothesis we have that: (iv) $\Sigma_3 = \Sigma_4 \uplus \{t : \Psi\}$, (v) $T_3 = T_4 \uplus \{t : \tau\}$ (3) $\vdash_t M : \Gamma$, (4) $\Psi; \Gamma \vdash \tau$, and (vi) $\Sigma_4; M \vdash T_4$.

  Let $\Sigma_2 \stackrel{\text{def}}{=} \Sigma_4 \uplus \{t_1 : \Psi_1\}$, hence (1) $\Sigma_1 = \Sigma_2 \uplus \{t : \Psi\}$. Let $T_2 \stackrel{\text{def}}{=} T_4 \uplus \{t_1 : \tau_1\}$, we have that (2) $T_1 = T_2 \uplus \{t : \tau\}$. We are left with showing (5), in the following.

$$\dfrac{\dfrac{\text{(i) } \vdash_{t_1} M : \Gamma_1 \qquad \text{(ii) } \Psi_1; \Gamma_1 \vdash \tau_1 \qquad \text{(vi) } \Sigma_4; M \vdash T_4}{\Sigma_4 \uplus \{t_1 : \Psi_1\}; M \vdash T_4 \uplus \{t_1 : \tau_1\}}\ \text{T-TM-C}}{\Sigma_2; M \vdash T_2}\ \stackrel{\text{def}}{=}$$

□

Lemma 6.3.3. *If* $\Gamma_1 \vdash B_1$ *and* $p \in \operatorname{dom} \Gamma_1 \vee p \in \operatorname{dom} B_1$, *then there exist* Γ_2 *and* B_2 *such that* $\Gamma_1 = \Gamma_2 \uplus \{p : a\}$, $B_1 = B_2 \uplus \{p : n\}$, *and* $\Gamma_2 \vdash B_2$.

Proof. The proof follows by induction on the relation $\Gamma_! \vdash B_1$. We perform a case analysis on the derivation tree of the last rule applied.

$$\frac{\text{(i) } \Gamma_3 \vdash B_3}{\Gamma_3 \uplus \{q : a'\} \vdash B_3 \uplus \{q : m\}}$$

where $\Gamma_1 = \Gamma_3 \uplus \{q : a'\}$ and $B_1 = B_3 \uplus \{q : m\}$. If $p = q$, we are done. Otherwise, we have that $p \neq q$ and therefore, (ii) $p \in \operatorname{dom} \Gamma_3 \vee p \in \operatorname{dom} B_3$. Applying the induction hypothesis to (i) $\Gamma_3 \vdash B_3$ and (ii) $p \in \operatorname{dom} \Gamma_3 \vee p \in$

$\operatorname{dom} B_3$, yields that there exist Γ_4 and B_4 such that (iii) $\Gamma_3 = \Gamma_4 \uplus \{p\colon a\}$, (iv) $B_3 = B_4 \uplus \{p\colon n\}$, and (v) $\Gamma_4 \vdash B_4$. Let $\Gamma_2 = \Gamma_4 \uplus \{q\colon a'\}$ and $B_2 = B_4 \uplus \{q\colon m\}$. Thus,

$$\dfrac{\dfrac{\text{(v) } \Gamma_4 \vdash B_4}{\Gamma_4 \uplus \{q\colon a'\} \vdash B_4 \uplus \{q\colon m\}}}{\Gamma_2 \vdash B_2}$$

□

Lemma 6.3.4. *If* $\vdash_t M_1 \colon \Gamma$ *and* $t \notin \operatorname{dom} M_1(p)$*, then there exist a phaser map* M_2 *and a phaser* P *such that*

1. $M_1 = M_2 \uplus \{p\colon P\}$,
2. $\vdash_t M_2 \colon \Gamma$.

Proof. The proof follows by induction on the typing relation $\vdash_t M_1 \colon \Gamma$. Next, we perform a case analysis on the derivation of the last rule applied.

- Case T-PERM-NIL:

  $$\vdash_t \emptyset \colon \emptyset$$

  where M is $\emptyset$ and Γ is $\emptyset$. We reach a contradiction because $p \in \operatorname{dom} \emptyset$.

- Case T-PERM-SKIP:

  $$\dfrac{\text{(i) } \vdash_t M_3 \colon \Gamma \qquad \text{(ii) } t \notin \operatorname{dom} P'}{\vdash_t M_3 \uplus \{q\colon P'\} \colon \Gamma}$$

  where M_1 is $M_3 \uplus \{q\colon P'\}$. If $p = q$, then we are done. Otherwise, $p \neq q$, thus $t \notin \operatorname{dom} M_3(p)$. Applying the induction hypothesis to $\vdash_t M_3 \colon \Gamma$, and $t \notin \operatorname{dom} M_3(p)$ yields that there exist a phaser map M_4 such that (iii) $M_3 = M_4 \uplus \{p\colon P\}$, (iv) $\vdash_t M_4 \colon \Gamma$. Let $M_2 = M_4 \uplus \{q\colon P'\}$ such that (1) $M_1 = M_2 \uplus \{p\colon P\}$.

  $$\dfrac{\dfrac{\text{(iv) } \vdash_t M_4 \colon \Gamma \qquad \text{(ii) } t \notin \operatorname{dom} P'}{\vdash_t M_4 \uplus \{q\colon P'\} \colon \Gamma}\ \text{T-PERM-SKIP}}{\text{(3) } \vdash_t M_2 \colon \Gamma}\ \stackrel{\text{def}}{=}$$

- Case T-PERM-CONS:

  $$\dfrac{\text{(i) } \vdash_t M_3 \colon \Gamma' \qquad \text{(ii) } P'(t) = (m, a')}{\vdash_t M_3 \uplus \{q\colon P'\} \colon \Gamma' \uplus \{q\colon a'\}}$$

where M_1 is $M_3 \uplus \{q\colon P'\}$ and Γ is $\Gamma' \uplus \{q\colon a'\}$. If $p = q$, we reach a contradiction. Otherwise, we have that $p \neq q$. Applying the induction hypothesis to $\vdash_t M_3 \colon \Gamma'$ and $t \notin \operatorname{dom} M_3(p)$, results in a phaser map M_4 and a phaser P such that (iii) $M_3 = M_4 \uplus \{p\colon P\}$, (iv) $\vdash_t M_4 \colon \Gamma'$. Let (1) $M_2 = M_4 \uplus \{q\colon P'\}$. Thus,

$$\dfrac{\dfrac{\text{(v) } \vdash_t M_4 \colon \Gamma' \qquad \text{(ii) } P'(t) = (m, a')}{\vdash_t M_4 \uplus \{q\colon P'\} \colon \Gamma' \uplus \{q\colon a'\}}\ \text{T-PERM-CONS} \quad \overset{\text{def}}{=}}{\vdash_t M_2 \colon \Gamma}$$

□

Lemma 6.3.5. *If $\vdash_t M_1 \colon \Gamma_1$ and $t \in \operatorname{dom} M_1(p)$, then there exist a phaser map M_2, a typing Γ_2, a phaser P, and a flag a such that*

1. $M_1 = M_2 \uplus \{p\colon P\}$,
2. $\Gamma_1 = \Gamma_2 \uplus \{p\colon a\}$, *and*
3. $\vdash_t M_2 \colon \Gamma_2$.

Proof. The proof follows by induction on the typing relation $\vdash_t M_1 \colon \Gamma_1$. Next, we perform a case analysis on the derivation of the last rule applied.

- Case T-PERM-NIL:

  $$\vdash_t \emptyset \colon \emptyset$$

  where M is $\emptyset$ and Γ is $\emptyset$. We reach a contradiction because $t \in \operatorname{dom} \emptyset(p)$.

- Case T-PERM-SKIP:

  $$\dfrac{\text{(i) } \vdash_t M_3 \colon \Gamma_1 \qquad \text{(ii) } t \notin \operatorname{dom} P'}{\vdash_t M_3 \uplus \{q\colon P'\} \colon \Gamma_1}$$

  where M_1 is $M_3 \uplus \{q\colon P'\}$. If $p = q$, then we are done, as we reach a contradiction. Otherwise, $p \neq q$, thus $t \in \operatorname{dom} M_3(p)$. Applying the induction hypothesis to $\vdash_t M_3 \colon \Gamma_1$, and $t \in \operatorname{dom} M_3(p)$ yields that there exist a phaser map M_4 and a typing Γ_2 such that (iii) $M_3 = M_4 \uplus \{p\colon P\}$, (2) $\Gamma_1 = \Gamma_2 \uplus \{p\colon a\}$, and (iv) $\vdash_t M_4 \colon \Gamma_2$. Let $M_2 = M_4 \uplus \{q\colon P'\}$ such that (1) $M_1 = M_2 \uplus \{p\colon P\}$.

  $$\dfrac{\dfrac{\text{(iv) } \vdash_t M_4 \colon \Gamma_2 \qquad \text{(ii) } t \notin \operatorname{dom} P'}{\vdash_t M_4 \uplus \{q\colon P'\} \colon \Gamma_2}\ \text{T-PERM-SKIP} \quad \overset{\text{def}}{=}}{\text{(3) } \vdash_t M_2 \colon \Gamma_2}$$

- Case T-PERM-CONS:

$$\frac{\text{(i) } \vdash_t M_3 \colon \Gamma_3 \qquad \text{(ii) } P'(t) = (m, a')}{\vdash_t M_3 \uplus \{q \colon P'\} \colon \Gamma_3 \uplus \{q \colon a'\}}$$

where M_1 is $M_3 \uplus \{q \colon P'\}$ and Γ is $\Gamma' \uplus \{q \colon a'\}$. If $p = q$, we are done. Otherwise, we have that $p \neq q$. Applying the induction hypothesis to $\vdash_t M_3 \colon \Gamma_3$ and $t \in \operatorname{dom} M_1(p)$, results in a phaser map M_4, a typing Γ_4, a phaser P, and a flag a such that (iii) $M_3 = M_4 \uplus \{p \colon P\}$, (iv) $\Gamma_3 = \Gamma_4 \uplus \{p \colon a\}$, and (v) $\vdash_t M_4 \colon \Gamma_4$. Let (1) $M_2 = M_4 \uplus \{q \colon P'\}$ and (2) $\Gamma_2 \stackrel{\text{def}}{=} \Gamma_4 \uplus \{q \colon a'\}$. Thus,

$$\dfrac{\dfrac{\text{(v) } \vdash_t M_4 \colon \Gamma_4 \qquad \text{(ii) } P'(t) = (m, a')}{\vdash_t M_4 \uplus \{q \colon P'\} \colon \Gamma_4 \uplus \{q \colon a'\}}\ \text{T-PERM-CONS} \qquad \stackrel{\text{def}}{=}}{\vdash_t M_2 \colon \Gamma_2}$$

□

Lemma 6.3.6. *If $\Delta; N \vdash M_1$ and $M_1(p) = P$, then there exists a phaser map M_2 such that:*

1. $M_1 = M_2 \uplus \{p \colon P\}$,
2. $\Delta \vdash P$,
3. $\operatorname{dom} P \subseteq N$, *and*
4. $\Delta; N \vdash M_2$.

Proof. We invert the hypothesis and get the following proof tree.

$$\frac{\text{(i) } \Delta \vdash P' \qquad \text{(ii) } \operatorname{dom} P' \subseteq N \qquad \text{(iii) } \Delta; N \vdash M_3}{\Delta; N \vdash M_3 \uplus \{p' \colon P'\}}$$

where phaser map M is $M_3 \uplus \{p' \colon P'\}$. If $p = p'$ we are done. Otherwise, $p \neq p'$, and therefore (iv) $p \in \operatorname{dom} M_3$.

Applying the induction hypothesis to (iii) $\Delta; N \vdash M_3$ and (iv) $p \in \operatorname{dom} M_3$, yields that (v) $M_3 = M_4 \uplus \{p \colon P\}$, (vi) $\Delta \vdash P$ and we get (2), (vii) $\operatorname{dom} P \subseteq N$ and we get (3), and (viii) $\Delta \vdash M_4$. We are left with showing (1) and (4). Let $M_2 = M_4 \uplus \{p' \colon P'\}$.

$$\dfrac{\dfrac{\text{(i) } \Delta \vdash P' \qquad \text{(ii) } \operatorname{dom} P' \subseteq N \qquad \text{(viii) } \Delta; N \vdash M_4}{\Delta; N \vdash M_4 \uplus \{p' \colon P'\}}\ \text{T-P-MAP-CONS} \qquad \stackrel{\text{def}}{=}}{(4)\ \Delta; N \vdash M_2}$$

□

Lemma 6.3.7. *If $\Delta; t_1; n_1 \vdash P_1$ and $P_1(t_2) = (n_2, a_2)$, then there exists a phaser P_2 such that*

1. $P_1 = P_2 \uplus \{t_2 \colon (n_2, a_2)\}$,
2. $\Delta; t_1; n_1 \vdash P_1$, *and*
3. $\Delta(t_1, t_2) = n_1 - n_2$.

Proof. The proof follows by induction on the derivation of the structure of first hypothesis. By inverting hypothesis $\Delta; t_1; n_1 \vdash P_1$ we get the following derivation.

$$\frac{\text{(i) } \Delta; t_1; n_1 \vdash P_3 \qquad \text{(ii) } \Delta(t_1, t_3) = (n_1 - n_3)}{\Delta; t_1; n_1 \vdash P_3 \uplus \{t_3 \colon (n_3, \_)\}}$$

If $t_3 = t_2$, then we are done. Otherwise, $t_3 \neq t_2$ and therefore $t_2 \in \operatorname{dom} P_3$. Let (iii) $P_3(t_2) = (n_2, a_2)$. Applying the induction hypothesis to (i) $\Delta; t_1; n_1 \vdash P_3$ and (iii), yields that there exists a phaser P_4 such that (iv) $P_3 = P_4 \uplus \{t_2 \colon (n_2, a_2)\}$, (v) $\Delta; t_1; n_1 \vdash P_4$, and (vi) $\Delta(t_1, t_2) = n_1 - n_2$ so we get (3). Let $P_2 = P_4 \uplus \{t_3 \colon (n_3, a_3)\}$. We have that (1) $P_2 \uplus \{t_2 \colon (n_2, a_2)\}$.

$$\dfrac{\dfrac{\text{(v) } \Delta; t_1; n_1 \vdash P_4 \qquad \text{(iii) } \Delta(t_1, t_3) = n_1 - n_3}{\Delta; t_1; n_1 \vdash P_4 \uplus \{t_3 \colon (n_3, a_3)\}}\ \textsc{d-l-cons}}{\text{(2) } \Delta; t_1; n_1 \vdash P_2}\ \stackrel{\text{def}}{=}$$

□

Lemma 6.3.8. *If $N \vdash \Delta$, $\Delta \vdash P_1$, $P_1(t) = (n, a)$, then there exists P_2 such that $P_1 = P_2 \uplus \{p \colon (n, a)\}$, $\Delta \vdash P_2$, and $\Delta; t; n \vdash P_2$.*

Proof. The proof follows by induction on the structure of $\Delta \vdash P_1$. We invert the hypothesis and obtain the following derivation.

$$\frac{\text{(i) } \Delta \vdash P_3 \qquad \text{(ii) } \Delta; t'; m \vdash P_3}{\Delta \vdash P_3 \uplus \{t' \colon (m, a')\}}$$

If $t = t'$, then we are done. Otherwise, $t \neq t'$ and therefore $t \in \operatorname{dom} P_3$. Hence (iii) $P_3(t) = (n, a)$. Applying the induction hypothesis to (i) $\Delta \vdash P_3$ and (iii) $P_3(t) = (n, a)$, yields that (iv) $P_3 = P_4 \uplus \{t \colon (n, a)\}$, (v) $\Delta \vdash P_4$, and (vi) $\Delta; t; n \vdash P_4$

Since (ii) $\Delta; t'; m \vdash P_3$ and (iii) $P_3(t) = (n, a)$, then by Lemma 6.3.7 there exists a phaser P_4' such that $P_3 = P_4' \uplus \{t \colon (n, a)\}$, $\Delta; t'; m \vdash P_4'$, and $\Delta(t', t) = m - n$. Given that $P_3 = P_4' \uplus \{t \colon (n, a)\}$ and that $P_3 = P_4 \uplus \{t \colon (n, a)\}$, then $P_4 = P_4'$. Thus, we have that (vii) $\Delta; t'; m \vdash P_4$.

Let $P_2 \stackrel{\text{def}}{=} P_4 \uplus \{t' \colon (m, a')\}$.

$$\dfrac{\dfrac{\text{(vi) } \Delta \vdash P_4 \qquad \text{(vii) } \Delta; t'; m \vdash P_4}{\Delta \vdash P_4 \uplus \{t' \colon (m, a')\}}\ \text{D-PH-CONS} \quad \stackrel{\text{def}}{=}}{\text{(2) } \Delta \vdash P_2}$$

Finally,

$$\dfrac{\text{(vi) } \Delta; t; n \vdash P_4 \qquad \dfrac{\Delta(t', t) = (m - n)}{\Delta(t, t') = (n - m)}\ \text{Inv. } N \vdash \Delta}{\dfrac{\Delta; t; n \vdash P_4 \uplus \{t' \colon (m, a')\}}{\text{(3) } \Delta; t; n \vdash P_2}\ \stackrel{\text{def}}{=}}\ \text{D-L-CONS}$$

□

6.4 The domain of typing contexts

We establish some properties about the domain of typing contexts.

1. any member of a well-typed s must be in the in the domain of its typing contexts;
2. any phaser in the inferred typing's domain of a phaser map, is also in the domain of that phaser map;
3. the typing relation $\vdash_t M \colon \Gamma$ constructs typing Γ as

$$\{p \colon a \mid \forall p \in \operatorname{dom} M \colon M(p)(t) = (n, a)\}$$

Lemma 6.4.1. *If* $\Gamma \vdash s \colon \Gamma'$*, then* $\Gamma' \subseteq \Gamma$ *and* $\operatorname{dom} \Gamma' = s$.

Proof. The proof follows by induction on the typing relation. We do a case analysis on the derivation of the last rule applied.

- Case T-A-N:

$$\Gamma \vdash \emptyset \colon \emptyset$$

  where s is $\emptyset$ and Γ' is $\emptyset$. We have that $s = \emptyset = \operatorname{dom} \emptyset$.

- Case T-A-C:

$$\dfrac{\text{(i) } \Gamma(p) = a \qquad \text{(ii) } \Gamma \vdash s' \colon \Gamma''}{\Gamma \vdash s' \uplus \{p\} \colon \Gamma'' \uplus \{q \colon a\}}$$

  where Γ' is $\Gamma'' \uplus \{q \colon a\}$ and s is $s' \uplus \{p\}$. Applying the induction hypothesis to (ii) $\Gamma \vdash s' \colon \Gamma''$ we get that $\Gamma'' \subseteq \Gamma$ and $\operatorname{dom} \Gamma'' = s'$. Thus, $\operatorname{dom} \Gamma'' \cup \{p\} = s' \cup \{p\}$. And, by definition we have $\operatorname{dom} \Gamma' = s$, so $\Gamma' \subseteq \Gamma$.

□

Lemma 6.4.2. *If* $\vdash_t M : \Gamma$, *then* $\operatorname{dom}\Gamma \subseteq \operatorname{dom} M$.

Proof. The proof follows by induction on the typing relation. We perform an inversion and a case analysis on the derivation of the last rule applied.

- Case T-PERM-NIL:

$$\vdash_t \emptyset : \emptyset$$

  where M is $\emptyset$ and Γ is $\emptyset$. We have that $\operatorname{dom}\emptyset \subseteq \operatorname{dom}\emptyset$ by definition.

- Case T-PERM-SKIP:

$$\frac{\vdash_t M' : \Gamma \qquad t \notin \operatorname{dom} P}{\Gamma \vdash_t M' \uplus \{p : P\} : \Gamma}$$

  where M is $M' \uplus \{p : P\}$. Applying the induction hypothesis to $\vdash_t M' : \Gamma$ yields that $\operatorname{dom}\Gamma \subseteq \operatorname{dom} M'$. Given that $\operatorname{dom}\Gamma \subseteq \operatorname{dom} M'$ and $\operatorname{dom} M' \subseteq \operatorname{dom} M$, then $\operatorname{dom}\Gamma \subseteq \operatorname{dom} M$.

- Case T-PERM-CONS:

$$\frac{\vdash_t M' : \Gamma' \qquad P(t) = (\_, a)}{\Gamma' \vdash_l M' \uplus \{p : P\} : \Gamma' \uplus \{p : a\}}$$

  where M is $M' \uplus \{p : P\}$ and Γ is $\Gamma' \uplus \{p : a\}$. Applying the induction hypothesis to $\vdash_t M' : \Gamma'$ yields that $\operatorname{dom}\Gamma' \subseteq \operatorname{dom} M'$. So, $\operatorname{dom}\Gamma' \cup \{p\} \subseteq \operatorname{dom} M' \cup \{p\}$ and therefore $\operatorname{dom}\Gamma \subseteq \operatorname{dom} M$.

□

Lemma 6.4.3. *If* $\vdash_t M : \Gamma$, *then* $\Gamma(p) = a \iff M(p)(t) = (n, a)$.

Proof. ($\Longrightarrow$)

The proof follows by induction on the typing relation $\vdash_t M : \Gamma$. Next, we perform a case analysis on the derivation of the last rule applied.

- Case T-PERM-SKIP:

$$\frac{\vdash_t M' : \Gamma \qquad t \notin \operatorname{dom} P}{\vdash_t M' \uplus \{q : P\} : \Gamma}$$

  where M is $M' \uplus \{q : P\}$.

  - Case $p = q$. In this case, we have that $p \notin \operatorname{dom} M'$. Hence, Lemma 6.4.2 and $\vdash_t M' : \Gamma$, we have that $p \notin \operatorname{dom}\Gamma$ and we reach a contradiction.

  - Case $p \neq q$. Applying the induction hypothesis to $\vdash_t M' : \Gamma$ and $p \in \operatorname{dom}\Gamma$ yields that $M'(p)(t) = (n, a)$ and $\Gamma(p) = a$. As $q \notin \operatorname{dom} M'$, then we get that $M(p)(t) = (n, a)$.

- Case T-PERM-NIL:

$$\vdash_t \emptyset : \emptyset$$

  where M is $\emptyset$ and Γ is $\emptyset$. We reach a contradiction because $p \in \operatorname{dom}\Gamma \stackrel{\text{def}}{=} p \in \operatorname{dom}\emptyset$.

- Case T-PERM-CONS:

$$\frac{\vdash_t M' : \Gamma' \qquad P(t) = (n, a)}{\vdash_t M' \uplus \{q : P\} : \Gamma' \uplus \{q : a'\}}$$

  where M is $M' \uplus \{q : P\}$ and Γ is $\Gamma' \uplus \{q : a'\}$. If $p = q$, we are done. Otherwise, we have that $p \neq q$. Applying the induction hypothesis to $\vdash_t M' : \Gamma'$ and $p \in \operatorname{dom}\Gamma'$, results in $M'(p)(t) = (n, a)$ and $\Gamma'(p) = a$. Thus, $M(p)(t) = (n, a)$ and $\Gamma(p) = a$.

($\Longleftarrow$) The proof follows by induction on the typing relation $\vdash_t M : \Gamma$. Next, we perform a case analysis on the derivation of the last rule applied.

- Case T-PERM-SKIP:

$$\frac{\vdash_t M' : \Gamma \qquad t \notin \operatorname{dom} P}{\vdash_t M' \uplus \{q : P\} : \Gamma}$$

  where M is $M' \uplus \{q : P\}$. We have that $q \neq p$, otherwise we get a contradiction, as $t \in \operatorname{dom} P$ and $t \notin \operatorname{dom} P$. Applying the induction hypothesis to $\vdash_t M' : \Gamma$ and $M'(p)(t) = (n, a)$ (since $p \in \operatorname{dom} M'$), yields $\Gamma(t) = a$.

- Case T-PERM-NIL:

$$\vdash_t \emptyset : \emptyset$$

  where M is $\emptyset$ and Γ is $\emptyset$. The case does not apply as $\operatorname{dom} M \neq \emptyset$.

- Case T-PERM-CONS:

$$\frac{\vdash_t M' : \Gamma' \qquad P(t) = (\_, a)}{\vdash_t M' \uplus \{q : P\} : \Gamma' \uplus \{q : a'\}}$$

  where M is $M' \uplus \{q : P\}$ and Γ is $\Gamma' \uplus \{q : a'\}$. If $p = q$, we are done. Otherwise, we have that $p \neq q$. Applying the induction hypothesis to $\vdash_t M' : \Gamma'$ and $M'(p)(t) = (n, a)$, results in $\Gamma'(t) = a$. Hence, $\Gamma(t) = a$.

□

6.5 Strengthening

Strengthening is when we are able to generalise the context necessary to type given a term. The intuition behind usefulness of these lemmas is that with them we can "forget" certain restrictions on the left-hand side of the turnstile (the context). We establish the strengthening of the typing for arguments, of the phaser map when inferring a typing, of the phaser map when checking task maps, of the task names when checking phaser maps, and of task names when checking differences.

Lemma 6.5.1. *If* $\Gamma \uplus \{p\colon a\} \vdash s\colon \Gamma'$ *and* $p \notin s$*, then* $\Gamma \vdash s\colon \Gamma'$.

Proof. By inversion of the hypothesis we get the following premises.

$$\frac{\text{(i) } (\Gamma \uplus \{p\colon a\})(q) = a' \qquad \text{(ii) } \Gamma \uplus \{p\colon a\} \vdash s'\colon \Gamma''}{\Gamma \uplus \{p\colon a\} \vdash (s' \uplus \{q\})\colon \Gamma'' \uplus \{q\colon a'\}}$$

where $s \stackrel{\text{def}}{=} s' \uplus \{q\}$. Since $p \notin s$ (hypothesis) and $s \stackrel{\text{def}}{=} s' \uplus \{q\}$, then $p \notin s'$. Applying the induction hypothesis to (ii) $\Gamma \uplus \{p\colon a\} \vdash s'\colon \Gamma''$ and the latter, we get that (iii) $\Gamma \vdash s'\colon \Gamma''$. Thus,

$$\dfrac{\dfrac{\text{(i) } (\Gamma \uplus \{p\colon a\})(q) = a' \qquad p \neq q}{\Gamma(q) = a'} \qquad \text{(iii) } \Gamma \vdash s'\colon \Gamma''}{\Gamma \vdash (s' \uplus \{q\})\colon \Gamma'' \uplus \{q\colon a'\}}\ \textsc{T-a-c}$$

□

Lemma 6.5.2. *If*

1. $\vdash_{t'} M \uplus \{p\colon P \uplus \{t\colon v\}\}\colon \Gamma$ *and*
2. $t \notin \operatorname{dom} T$,

then $\vdash_{t'} M \uplus \{p\colon P\}\colon \Gamma$.

Proof. Let $M_1 \stackrel{\text{def}}{=} M \uplus \{p\colon P \uplus \{t\colon v\}\}$. We test if $t \in \operatorname{dom} M_1(p)$:

- Case $t \in \operatorname{dom} M_1(p)$.

  From Lemma 6.3.5 and $\vdash_{t'} M_1\colon \Gamma$ and $t \in \operatorname{dom} M_1(p)$ yields that there exists a typing Γ' such that (i) $\Gamma = \Gamma' \uplus \{p\colon a\}$ and (ii) $\vdash_{t'} M\colon \Gamma'$. Hence,

$$\dfrac{\text{(i) } \vdash_{t'} M\colon \Gamma' \qquad \dfrac{(P \uplus \{t\colon v\})(t') = (n, a) \quad t \in \operatorname{dom} M_1(p)}{P(t') = (n, a)}}{\vdash_{t'} M \uplus \{p\colon P\}\colon \Gamma' \uplus \{p\colon a\}}\ \textsc{T-perm-cons}$$

- Case $t \notin \operatorname{dom} M_1(p)$.

  From Lemma 6.3.4 and $\vdash_{t'} M_1 : \Gamma$ and $t \notin \operatorname{dom} M_1(p)$ yields that (i) $\vdash_{t'} M : \Gamma$. Thus,

$$\frac{\text{(i) } \vdash_{t'} M : \Gamma \qquad \dfrac{t \notin \operatorname{dom} M_1(p)}{t' \notin \operatorname{dom} P}}{\vdash_{t'} M \uplus \{p : P\} : \Gamma}\ \text{T-PERM-SKIP}$$

□

Lemma 6.5.3. *If*

1. $\Sigma; M \uplus \{p : P \uplus \{t : v\}\} \vdash T$ *and*
2. $t \notin \operatorname{dom} T$,

then $\Sigma'; M \uplus \{p : P\} \vdash T$.

Proof. The proof follows by induction on the typing relation (1). Next, we perform a case analysis on the derivation of the last rule applied.

- Case T-TM-N:

$$\emptyset; M \uplus \{p : P \uplus \{t : v\}\} \vdash \emptyset$$

  The case holds by direct application of rule T-TM-N.

- Case T-TM-C:

$$\frac{\text{(i) } \vdash_{t'} M_1 : \Gamma \qquad \text{(ii) } \Psi; \Gamma \vdash \tau \qquad \text{(iii) } \Sigma''; M_1 \vdash T'}{\Sigma'' \uplus \{t' : \Psi\}; M_1 \vdash T' \uplus \{t' : \tau\}}$$

  where Σ is $\Sigma'' \uplus \{t' : \Psi\}$, M_1 is $M \uplus \{p : P \uplus \{t' : v\}\}$ and T is $T' \uplus \{t' : \tau\}$. By Lemma 6.5.2 and $t \notin \operatorname{dom} T$ we get that (iv) $\vdash_{t'} M \uplus \{p : P\} : \Gamma$. Applying the induction hypothesis to $\Sigma''; M_1 \vdash T'$ and $t \notin \operatorname{dom} T'$ (as $t \notin \operatorname{dom} T$) yields (v) $\Sigma''; M \uplus \{p : P\} \vdash T'$ Thus, with rule T-TM-C we get the following tree.

$$\frac{\text{(iv) } \vdash_{t'} M \uplus \{p : P\} : \Gamma \quad \text{(ii) } \Psi; \Gamma \vdash \tau \quad \text{(v) } \Sigma''; M \uplus \{p : P\} \vdash T'}{\dfrac{\Sigma'' \uplus \{t' : \Psi\}; M_1 \vdash T' \uplus \{t' : \tau\}}{\Sigma'; M \uplus \{p : P\} \vdash T} \overset{\text{def}}{=}}$$

□

Lemma 6.5.4. *If* $\Delta; N \vdash M$ *and* $\vdash_t M : \emptyset$*, then* $\Delta; N \setminus \{t\} \vdash M$.

Proof. The proof follows by induction on relation $\Delta; N \vdash M$. We do a case analysis on the derivation of the last rule applied.

- Case T-P-MAP-NIL:

$$\Delta; N \vdash \emptyset$$

  where M is $\emptyset$.

  The proof concludes with the application of rule T-P-MAP-NIL.

- Case T-P-MAP-CONS:

$$\frac{\text{(i) } \Delta \vdash P \qquad \text{(ii) } \operatorname{dom} P \subseteq N \qquad \text{(iii) } \Delta; N \vdash M'}{\Delta; N \vdash M' \uplus \{p : P\}}$$

  where M is $M' \uplus \{p : P\}$.

  Applying the induction hypothesis to (iii) $\Delta; N \vdash M'$ and $\vdash_t M : \emptyset$ yields that (iv) $\Delta; N \setminus \{t\} \vdash M'$. From Lemma 6.4.3 and $\vdash_t M : \emptyset$ we get that (v) $t \notin \operatorname{dom} P$. Hence, we can apply rule T-P-MAP-CONS to conclude this case.

$$\frac{\text{(i) } \Delta \vdash P \qquad \dfrac{\text{(v) } t \notin \operatorname{dom} P \quad \text{(ii) } \operatorname{dom} P \subseteq N}{\operatorname{dom} P \subseteq N \setminus \{t\}} \qquad \text{(iv) } \Delta; N \setminus \{t\} \vdash M'}{\Delta; N \setminus \{t\} \vdash M' \uplus \{p : P\}}$$

□

Lemma 6.5.5. *If* $N \vdash \Delta$*, then* $N \setminus \{t\} \vdash \Delta$.

Proof. We invert $N \vdash \Delta$ and obtain

$$\frac{\begin{array}{c}\text{(i) } (N, \leq_\Delta) \text{ is a total ordering} \\ \text{(ii) } \forall t_1, t_2 \in N : \Delta(t_1, t_2) = z \implies \Delta(t_2, t_1) = -z\end{array}}{N \vdash \Delta}$$

By Definition 6.1.3 and $(N, \leq_\Delta)$ is a total ordering, then $(N \setminus \{t\}, \leq_\Delta)$ is a total ordering. From (ii) we get $\forall t_1, t_2 \in N \setminus \{t\} : \Delta(t_1, t_2) = z \implies \Delta(t_2, t_1) = -z$.

Hence, we conclude the proof by applying rule D-WF. □

Lemma 6.5.6. *If* $\langle \Delta; \Sigma \rangle \vdash \big(M, T \uplus \{t : (B, \mathtt{end})\}\big)$*, then* $\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)$.

Proof. By inverting the hypothesis we get

$$\frac{\text{(i) } N \vdash \Delta \qquad \text{(ii) } \Delta; N \vdash M \qquad \text{(iii) } \Sigma; M \vdash T \uplus \{t\colon (B, \mathtt{end})\}}{\langle \Delta; \Sigma \rangle \vdash \big(M, T \uplus \{t\colon (B, \mathtt{end})\}\big)}$$

where $N = \operatorname{dom} T \cup \{t\}$. Inverting (iii) yields the following premises

$$\frac{\text{(iv) } \vdash_t M\colon \emptyset \qquad \dfrac{\emptyset \vdash \emptyset \qquad \emptyset \vdash \mathtt{end}\colon \emptyset}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (\emptyset, \mathtt{end})} \qquad \text{(v) } \Sigma'; M \vdash T}{\Sigma' \uplus \{t\colon \langle \emptyset; \emptyset \rangle\}; M \vdash T \uplus \{t\colon (\emptyset, \mathtt{end})\}}$$

where Σ is $\Sigma' \uplus \{t\colon \langle \emptyset; \emptyset \rangle\}$ and B is $\emptyset$. Since we have (ii) $\Delta; N \vdash M$ and (iv) $\vdash_t M\colon \emptyset$, then by Lemma 6.5.4 we have (vi) $\Delta; \operatorname{dom} T \vdash M$. Since we have (i) $N \vdash \Delta$, then by Lemma 6.5.5 we have (vii) $\operatorname{dom} T \vdash \Delta$. The proof concludes by applying (vii), (vi), and (v) to rule T-AMACH. □

Chapter Seven

Subject reduction

A type system enjoys the property of subject reduction if the reduction relation preserves well-typedeness. The objective is to show that, for any well-typed term, reduction always yields typable terms. This property works as a "sanity check" of type systems.

To understand the basic idea behind subject reduction, the reader can proceed to the proof of Lemma 7.10.2. The proof is trivial, as we need only to ensure that program concatenation preserves the typing Γ.

The chapter is divided into one section per reduction relation, plus a section for the main result (Theorem 7.11.1).

7.1 Async

Definition 7.1.1. *We define function* $\text{copy}_{\text{D}}(t_a, t_b, \Delta) \stackrel{\text{def}}{=} \Delta'$. *Let* $\Delta'(t, t') = \Delta(t\sigma, t'\sigma)$, *where* $\sigma = [t_a/t_b]$.

Lemma 7.1.1. *If* $N \vdash \Delta$, *then* $N \cup \{t'\} \vdash \text{copy}_{\text{D}}(t, t', \Delta)$.

Proof. Let $\text{copy}_{\text{D}}(t, t', \Delta) = \Delta'$.

1. We show that if $t_1, t_2 \in N \cup \{t'\}$ and $\Delta'(t_1, t_2) = z$, then $\Delta(t_2, t_1) = -z$. From Definition 7.1.1 and $\Delta'(t_1, t_2) = z$, we have that $\Delta(t_1\sigma, t_2\sigma) = z$. But we know that $N \vdash \Delta$, thus $\Delta(t_2\sigma, t_1\sigma) = -z$ and therefore From Definition 7.1.1 $\Delta'(t_1, t_2) = z$.

2. We now show that $(N \cup \{t'\}, \leq_{\Delta'})$ is a total ordering. Substitution $\sigma = [t/t']$ is an injective function over N, hence we can state that $\leq_{\Delta_2}$ is an ordering induced by σ on N, defined by:

$$\forall t_1, t_2 \in N \cup \{t'\} \colon t_1 \leq_{\Delta'} t_2 \iff t_1\sigma \leq_{\Delta} t_2\sigma$$

   Since $(N, \leq_{\Delta})$ is a total ordering and σ is an injective function from $N \cup \{t'\}$ to N, then $\leq_{\Delta'}$ (the ordering induced by σ) is also a total ordering.

Applying rule D-WF to (1) and (2) concludes this proof. □

Lemma 7.1.2. *If* $\Delta; t; n \vdash P$ *and* $t_2 \notin \operatorname{dom} P$*, then* $\operatorname{copy}_{\mathrm{D}}(t_1, t_2, \Delta); t; n \vdash P$.

Proof. The proof follows by induction on the typing relation. We perform a case analysis on the derivation of the last rule applied.

- Case D-L-NIL:

$$\Delta; t; n \vdash \emptyset$$

  The case holds with the application of rule D-L-NIL.

- Case D-L-CONS:

$$\frac{\text{(i) } \Delta; t; n \vdash P' \qquad \text{(ii) } \Delta(t, t') = (n - n')}{\Delta; t; n \vdash P' \uplus \{t \colon \langle n'; a \rangle\}}$$

  where P is $P' \uplus \{t \colon \langle n'; a \rangle\}$. Let $\Delta' = \operatorname{copy}_{\mathrm{D}}(t_1, t_2, \Delta)$. Applying the induction hypothesis to $\Delta; t; n \vdash P'$ and $t_2 \notin \operatorname{dom} P'$ (as $t_2 \notin \operatorname{dom} P$), we get (iii) $\Delta'; t; n \vdash P'$. Since $t \neq t_2$ and $t' \neq t_2$, then by Definition 7.1.1 $\Delta(t, t') = \Delta'(t, t')$ and therefore $\Delta'(t, t') = (n - n')$. Hence,

$$\frac{\text{(iii) } \Delta'; t; n \vdash P' \qquad \text{(iv) } \Delta'(t, t') = (n - n')}{\Delta'; t; n \vdash P' \uplus \{t \colon \langle n'; a \rangle\}} \text{ D-L-CONS}$$

□

Lemma 7.1.3. *If* $\Delta \vdash P$ *and* $t_2 \notin \operatorname{dom} P$*, then* $\operatorname{copy}_{\mathrm{D}}(t_1, t_2, \Delta) \vdash P$.

Proof. The proof follows by induction on $\Delta \vdash P$. We perform a case analysis on the derivation of the last rule applied.

- Case D-PH-NIL:

$$\Delta \vdash \emptyset$$

  The proof for this cases consists of the direct application of rule D-PH-NIL.

- Case D-PH-CONS:

$$\frac{\text{(i) } \Delta \vdash P' \qquad \text{(ii) } \Delta; t; n \vdash P'}{\Delta \vdash P' \uplus \{t \colon \langle n; a \rangle\}}$$

  where P is $P' \uplus \{t \colon \langle n; a \rangle\}$. Let $\Delta' = \operatorname{copy}_{\mathrm{D}}(t_1, t_2, \Delta)$. Applying the induction hypothesis on (i) $\Delta \vdash P'$ and $t_2 \notin \operatorname{dom} P$ yields (iii) $\Delta' \vdash P'$. With Lemma 7.1.2, (ii) $\Delta; t; n \vdash P'$, and $t_2 \notin \operatorname{dom} P$ (hypothesis), we get (iv) $\Delta'; t; n \vdash P'$.

  Therefore,

$$\frac{\text{(iii) } \Delta' \vdash P' \qquad \text{(iv) } \Delta'; t; n \vdash P'}{\Delta' \vdash P' \uplus \{t \colon \langle n; a \rangle\}}$$

□

Lemma 7.1.4. *If* $\Delta; t_1; n \vdash P$ *and* $t_2 \notin \operatorname{dom} P$*, then* $\operatorname{copy}_{\mathrm{D}}(t_1, t_2, \Delta); t_2; n \vdash P$.

Proof. The proof follows by induction on the typing relation. We perform a case analysis on the derivation of the last rule applied.

- Case D-L-NIL:

$$\Delta; t_1; n \vdash \emptyset$$

  The case holds with the application of rule D-L-NIL.

- Case D-L-CONS:

$$\frac{\text{(i) } \Delta; t_1; n \vdash P' \qquad \text{(ii) } \Delta(t_1, t) = (n - m)}{\Delta; t_1; n \vdash P' \uplus \{t \colon \langle m; a' \rangle\}}$$

  where P is $P' \uplus \{t \colon \langle m; a' \rangle\}$. Let $\Delta' = \operatorname{copy}_{\mathrm{D}}(t_1, t_2, \Delta)$. Applying the induction hypothesis to $\Delta; t_1; n \vdash P'$ and $t_2 \notin \operatorname{dom} P'$ (as $t_2 \notin \operatorname{dom} P$) results in (iii) $\Delta'; t_2; n \vdash P'$. Given that $t_1 \neq t_2$ and $t \neq t_2$, then by Definition 7.1.1 $\Delta'(t_1, t) = \Delta(t_1, t)$, thus (iv) $\Delta'(t_1, t) = (n - m)$. Thus,

$$\frac{\text{(iii) } \Delta; t_1; n \vdash P' \qquad \text{(iv) } \Delta(t_1, t) = (n - m)}{\Delta; t_1; n \vdash P' \uplus \{t \colon \langle m; a' \rangle\}} \text{ D-L-CONS}$$

□

Lemma 7.1.5. *If* $N \vdash \Delta$, $\Delta \vdash P$, $P(t_1) = v$, $t_2 \notin \operatorname{dom} P$*, then*

$$\operatorname{copy}_{\mathrm{D}}(t_1, t_2, \Delta) \vdash P \uplus \{t_2 \colon v\}$$

Proof. Let $v = \langle n; a \rangle$ and $\Delta' = \operatorname{copy}_{\mathrm{D}}(t_1, t_2, \Delta)$. Since we have $N \vdash \Delta$, $\Delta \vdash P$, and $P(t_1) = \langle n; a \rangle$, then there exists P' such that $P = P' \uplus \{p \colon \langle n; a \rangle\}$, (i) $\Delta \vdash P'$, and (ii) $\Delta; t_1; n \vdash P'$. By hypothesis we have that $t_2 \notin \operatorname{dom} P$ and we know that $P = P' \uplus \{p \colon \langle n; a \rangle\}$, then (iii) $t_2 \notin \operatorname{dom} P'$. Applying Lemma 7.1.4 to (ii) $\Delta; t_1; n \vdash P'$ and (iii) $t_2 \notin \operatorname{dom} P'$, yields (iv) $\Delta'; t_2; n \vdash P'$. By inverting $N \vdash \Delta$, we have that $\leq_\Delta$ is symmetric and $t_1 \in N$, thus $t_1 \leq_\Delta t_1$ and therefore from Definition 6.1.1 (v) $\Delta(t_1, t_1) = 0$. Hence, we have the following premise (vi).

$$\frac{\dfrac{\text{(iv) } \Delta'; t_2; n \vdash P' \qquad \dfrac{\text{(v) } \Delta(t_1, t_1) = 0 \qquad ([t_1/t_2])(t_2) = t_1}{\Delta'(t_2, t_1) = 0} \text{ Definition 7.1.1}}{\Delta'; t_2; n \vdash P' \uplus \{t_1 \colon \langle n; a \rangle\}} \text{ D-L-CONS}}{\Delta'; t_2; n \vdash P} \stackrel{\text{def}}{=}$$

Therefore,

$$\dfrac{\dfrac{\Delta \vdash P \quad t_2 \notin \operatorname{dom} P}{\Delta' \vdash P}\ \text{Lemma 7.1.3} \qquad \text{(vi) } \Delta'; t_2; n \vdash P}{\Delta' \vdash P \uplus \{t_2 \colon \langle n; a \rangle\}}\ \textsc{D-ph-cons}$$

□

Lemma 7.1.6. *If* $N \vdash \Delta$, $\Delta; N \vdash M_1$, $M_2 = \operatorname{copy}(s, t, t', M_1)$, *and* $t' \notin N$, *then*

$$\operatorname{copy}_{\mathrm{D}}(t, t', \Delta); N \cup \{t'\} \vdash M_2$$

Proof. Let $\Delta' \stackrel{\text{def}}{=} \operatorname{copy}_{\mathrm{D}}(t, t', \Delta)$ and $N' = N \cup \{t'\}$. The proof follows by induction on $M' = \operatorname{copy}(s, t, t', M)$. We perform a case analysis on the derivation of the last rule applied.

- Case Cpy-nil:

  $$\operatorname{copy}(\emptyset, t, t', \emptyset) = \emptyset$$

  where M_1 is $\emptyset$. The case concludes with rule T-p-map-nil.

- Case Cpy-cons:

  $$\dfrac{\text{(i) } \operatorname{copy}(s, t, t', M) = M' \qquad \text{(ii) } P(t) = v}{\operatorname{copy}(s \uplus \{p\}, t, t', M \uplus \{p \colon P\}) = M' \uplus \big\{p \colon P \uplus \{t' \colon v\}\big\}}$$

  where M_1 is $M \uplus \{p \colon P\}$ and M_2 is $M' \uplus \big\{p \colon P \uplus \{t' \colon v\}\big\}$. Let (iii) $P' \stackrel{\text{def}}{=} P \uplus \{t' \colon v\}$. With Lemma 6.3.6, $\Delta; N \vdash M_1$, and $M_1(p) = P$, we get (iv) $\Delta \vdash P$, (v) $\operatorname{dom} P \subseteq N$, and (vi) $\Delta; N \vdash M$.

  (a) Since we have that (v) $\operatorname{dom} P \subseteq N$ and $t' \notin N$ (hypothesis), then (vii) $t' \notin \operatorname{dom} P$. From Lemma 7.1.5, $N \vdash \Delta$ (hypothesis), (iv) $\Delta \vdash P$, (ii) $P(t) = v$, and (vii) $t' \notin \operatorname{dom} P$, we get that $\Delta' \vdash P'$.

  (b) The following tree holds.

  $$\dfrac{\dfrac{\operatorname{dom} P \subseteq N}{\operatorname{dom} P \cup \{t'\} \subseteq N \cup \{t'\}} = }{\operatorname{dom} P' \subseteq N'} \stackrel{\text{def}}{=}$$

  (c) Applying the induction hypothesis to $N \vdash \Delta$ (hyp.), (vi) $\Delta; N \vdash M$, (i) $\operatorname{copy}(s, t, t', M) = M'$, (vi) $\Delta; N \vdash M$, and $t' \notin N$ (hypothesis) to obtain $\Delta'; N' \vdash M'$.

Hence,

$$\dfrac{\dfrac{\text{(a) } \Delta' \vdash P' \qquad \text{(b) } \operatorname{dom} P' \subseteq N' \qquad \text{(c) } \Delta'; N' \vdash M'}{\Delta'; N' \vdash M' \uplus \{p\colon P'\}}\ \textsc{T-p-map-cons}}{\operatorname{copy}_{\mathrm{D}}(t, t', \Delta); N \cup \{t'\} \vdash M_2}\ \overset{\text{def}}{=}$$

- Case Cpy-skip:

$$\dfrac{\operatorname{copy}(s, t, t', M) = M' \qquad p \notin s}{\operatorname{copy}(s, t, t', M \uplus \{p\colon P\}) = M' \uplus \{p\colon P\}}$$

With Lemma 6.3.6, $\Delta; N \vdash M_1$, and $M_1(p) = P$, we get (iv) $\Delta \vdash P$, (v) $\operatorname{dom} P \subseteq N$, and (vi) $\Delta; N \vdash M$.

(a) From Lemma 7.1.3, (iv) $\Delta \vdash P$, and $t_2 \notin \operatorname{dom} P$, we get that $\Delta' \vdash P$.

(b) The following tree holds.

$$\dfrac{\dfrac{\operatorname{dom} P \subseteq N}{\operatorname{dom} P \subseteq N \cup \{t'\}}\ \cup}{\operatorname{dom} P \subseteq N'}\ \overset{\text{def}}{=}$$

(c) We apply the induction hypothesis to $N \vdash \Delta$ (hypothesis), (vi) $\Delta; N \vdash M$, (i) $\operatorname{copy}(s, t, t', M) = M'$, and $t' \notin N$ (hypothesis) to obtain $\Delta'; N' \vdash M'$.

Hence,

$$\dfrac{\dfrac{\text{(a) } \Delta' \vdash P \qquad \text{(b) } \operatorname{dom} P \subseteq N' \qquad \text{(c) } \Delta'; N' \vdash M'}{\Delta'; N' \vdash M' \uplus \{p\colon P\}}\ \textsc{T-p-map-cons}}{\operatorname{copy}_{\mathrm{D}}(t, t', \Delta); N \cup \{t'\} \vdash M_2}\ \overset{\text{def}}{=}$$

□

Lemma 7.1.7. *If $N \vdash \Delta$, $\Delta; N \vdash M$, $M' = \operatorname{copy}(s, t, t', M)$, and $t' \notin N$, then there exists a Δ' such that $N \cup \{t'\} \vdash \Delta'$ and $\Delta'; N \cup \{t'\} \vdash M'$.*

Proof. Let $\Delta' = \operatorname{copy}_{\mathrm{D}}(t, t', \Delta)$. From Lemma 7.1.1 and $N \vdash \Delta$ we get

$$N \cup \{t'\} \vdash \Delta'$$

From Lemma 7.1.6, $N \vdash \Delta$ (hypothesis), $\Delta; N \vdash M$ (hypothesis), $M' = \operatorname{copy}(s, t, t', M)$ (hypothesis), and $t' \notin N$, then $\Delta'; N \cup \{t'\} \vdash M'$. □

Lemma 7.1.8. *If* $\vdash_t M : \Gamma$, $\mathrm{copy}(s, t_1, t_2, M) = M'$, *and* $t \neq t_2$, *then* $\vdash_t M' : \Gamma$.

Proof. The proof follows by induction on $\mathrm{copy}(s, t_1, t_2, M) = M'$. We perform a case analysis on the derivation of the last rule applied.

- Case Cpy-nil:
$$\mathrm{copy}(\emptyset, t_1, t_2, \emptyset) = \emptyset$$
where M is $\emptyset$, s is $\emptyset$, and M' is $\emptyset$. The case holds by hypothesis $\vdash_t \emptyset : \Gamma$.

- Case Cpy-cons:
$$\frac{\text{(i) } \mathrm{copy}(s', t_1, t_2, M_1) = M_2 \qquad \text{(ii) } P(t_1) = v}{\mathrm{copy}(s' \uplus \{p\}, t_1, t_2, M_1 \uplus \{p : P\}) = M_2 \uplus \{p : P \uplus \{t_2 : v\}\}}$$
where M is $M_1 \uplus \{p : P\}$, s is $s' \uplus \{p\}$, and M' is $M_2 \uplus \{p : P \uplus \{t_2 : v\}\}$. We test the membership of $t \in \mathrm{dom}\, P$.

  - Case $t \in \mathrm{dom}\, M(p)$. Applying Lemma 6.3.5 to $\vdash_t M : \Gamma$, and $t \in \mathrm{dom}\, M(p)$, yields that there exist a typing Γ' such that (iii) $P(t) = (n, a)$, $\Gamma = \Gamma' \uplus \{p : a\}$, and $\vdash_t M_1 : \Gamma'$. Applying the induction hypothesis to the latter, (i) $\mathrm{copy}(s', t_1, t_2, M_1) = M_2$, and $t \neq t_2$ (hypothesis), we get that (iv) $\vdash_t M_2 : \Gamma'$. Hence,
$$\frac{\text{(iv) } \vdash_t M_2 : \Gamma' \qquad \text{(iii) } P(t) = (\_, a)}{\vdash_t M_2 \uplus \{p : P\} : \Gamma' \uplus \{p : a\}}\ \textsc{T-perm-cons}$$
  - Case $t \notin M(p)$.
    Applying Lemma 6.3.4 to $\vdash_t M : \Gamma$ and $t \notin \mathrm{dom}\, M(p)$ we get that $\vdash_t M_1 : \Gamma$. With the induction hypothesis, the latter, (i), and $t \neq t_2$ (hypothesis), we get that (iv) $\vdash_t M_2 : \Gamma$. Thus,
$$\frac{\vdash_t M_2 : \Gamma \qquad t \notin \mathrm{dom}\, P}{\vdash_t M_2 \uplus \{p : P\} : \Gamma}\ \textsc{T-perm-skip}$$

- Case Cpy-skip:
$$\frac{\mathrm{copy}(s, t_1, t_2, M_1) = M_2 \qquad p \notin s}{\mathrm{copy}(s, t_1, t_2, M_1 \uplus \{p : P\}) = M_2 \uplus \{p : P\}}$$
where M is $M_1 \uplus \{p : P\}$ and M' is $M_2 \uplus \{p : P\}$. The proof has the same structure as case Cpy-cons.

□

Lemma 7.1.9. *If* $\Sigma; M \vdash T$, $t_2 \notin \operatorname{dom} T$, *and* $\operatorname{copy}(s, t_1, t_2, M) = M'$, *then* $\Sigma; M' \vdash T$.

Proof. The proof follows by induction on the relation $\Sigma; M \vdash T$. We perform a case analysis on the derivation of the last rule applied.

- Case T-TM-N:

$$\emptyset; M \vdash \emptyset$$

  where Σ is $\emptyset$ and T is $\emptyset$. The proof for this case consists of a direct application of rule T-TM-N.

- Case T-TM-C:

$$\frac{\text{(i) } \vdash_t M : \Gamma \qquad \text{(ii) } \Psi; \Gamma \vdash \tau \qquad \text{(iii) } \Sigma'; M \vdash T'}{\Sigma' \uplus \{t : \Psi\}; M \vdash T' \uplus \{t : \tau\}}$$

  where Σ is $\Sigma' \uplus \{t : \Psi\}$ and T is $T' \uplus \{t : \tau\}$. From Lemma 7.1.8, (i) $\vdash_t M : \Gamma$, $\operatorname{copy}(s, t_1, t_2, M) = M'$ (hypothesis), and $t \neq t_2$ (hypothesis), and we get that (iv) $\vdash_t M' : \Gamma$. Next, we apply the induction hypothesis to $\Sigma; M \vdash T'$, $t_2 \notin \operatorname{dom} T'$ (as $t_2 \notin \operatorname{dom} T$), and $\operatorname{copy}(s, t_1, t_2, M) = M'$ to obtain (v) $\Sigma; M' \vdash T'$. Hence,

$$\frac{\text{(iv) } \vdash_t M' : \Gamma \qquad \text{(ii) } \Psi; \Gamma \vdash \tau \qquad \text{(v) } \Sigma'; M \vdash T'}{\Sigma' \uplus \{t : \Psi\}; M \vdash T' \uplus \{t : \tau\}}\ \text{T-TM-C}$$

□

Lemma 7.1.10. *If* $\Psi; \Gamma \vdash (B, \texttt{async}(s, b'); b)$, *then* $\Psi; \Gamma \vdash (B, b)$.

Proof. By inverting the hypothesis we get the following premises.

$$\frac{\text{(i) } \Gamma \vdash B \qquad \dfrac{\cdots \qquad \text{(ii) } \Gamma \vdash b : \emptyset}{\Gamma \vdash \texttt{async}(s, b'); b : \emptyset}}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, \texttt{async}(s, b'); b)}$$

The proof concludes applying rule T-T-R to premises (i) and (ii). □

Lemma 7.1.11. *If* $\Sigma; M \vdash T \uplus \{t : (B, \texttt{async}(s, b'); b)\}$, $\operatorname{copy}(s, t, t', M) = M'$, *and* $t' \notin \operatorname{dom} T \cup \{t\}$, *then* $\Sigma; M' \vdash T \uplus \{t : (B, b)\}$.

Proof. By Lemma 6.3.2 and $\Sigma; M \vdash T \uplus \{t : (B, \texttt{async}(s, b'); b)\}$ we get that

(i) $\Sigma = \Sigma'' \uplus \{t : \Psi\}$,

(ii) $\vdash_t M : \Gamma$,

(iii) $\Psi; \Gamma \vdash (B, \texttt{async}(s, b'); b)$, and

(iv) $\Sigma''; M \vdash T$.

We know that (v) $t \neq t'$, as from $t' \notin \operatorname{dom} T \cup \{t\}$ (hypothesis). Applying Lemma 7.1.8, (ii) $\vdash_t M : \Gamma$, $\operatorname{copy}(s, t_1, t_2, M) = M'$ (hypothesis), and (v) $t \neq t'$, then (vi) $\vdash_t M' : \Gamma$. From (iii) $\Psi; \Gamma \vdash (B, \texttt{async}(s, b'); b)$ and Lemma 7.1.10 we get (vii) $\Psi; \Gamma \vdash (B, b)$. Since we have $t' \notin \operatorname{dom} T \cup \{t\}$, then (viii) $t' \notin \operatorname{dom} T$. Applying Lemma 7.1.9 to (iv) $\Sigma''; M \vdash T$, (viii) $t' \notin \operatorname{dom} T$, and $\operatorname{copy}(s, t, t', M) = M'$ (hypothesis) yields (ix) $\Sigma''; M' \vdash T$.

Hence,

$$\dfrac{\dfrac{\text{(vi) } \vdash_t M' : \Gamma \qquad \text{(vii) } \Psi; \Gamma \vdash (B, b) \qquad \text{(ix) } \Sigma''; M' \vdash T}{\Sigma'' \uplus \{t : \Psi\}; M' \vdash T \uplus \{t : (B, b)\}}\ \text{T-TM-C}}{\Sigma; M' \vdash T \uplus \{t : (B, b)\}}\ \overset{\text{def}}{=}$$

□

Definition 7.1.2. *For any maps X and Y we have that $X \subseteq Y$ if, and only if $\forall z \in \operatorname{dom} X : X(z) = Y(z)$.*

Lemma 7.1.12. *If*

1. $\vdash_t M : \Gamma$,
2. $\forall p \in \operatorname{dom} M \implies t' \notin \operatorname{dom} M(p)$, *and*
3. $\operatorname{copy}(s, t, t', M) = M'$,

then there exists a typing Γ' such that $\vdash_{t'} M' : \Gamma'$, $\Gamma' \subseteq \Gamma$, and $\operatorname{dom} \Gamma' = s$.

Proof. The proof

- Case CPY-NIL:
$$\operatorname{copy}(\emptyset, t, t', \emptyset) = \emptyset$$
where M is $\emptyset$, s is $\emptyset$, and M' is $\emptyset$. We have $\vdash_{t'} \emptyset : \emptyset$ (with rule T-PERM-NIL), $\Gamma \subseteq \emptyset$, and $\operatorname{dom} \emptyset = \emptyset$.

- Case CPY-CONS:
$$\dfrac{\text{(i) } \operatorname{copy}(s', t, t', M_1) = M_2 \qquad \text{(ii) } P(t) = v}{\operatorname{copy}(s' \uplus \{p\}, t, t', M_1 \uplus \{p : P\}) = M_1 \uplus \{p : P \uplus \{t' : v\}\}}$$

where s is $s' \uplus \{p\}$, M is $M_1 \uplus \{p\colon P\}$, and M' is $M_1 \uplus \{p\colon P \uplus \{t'\colon v\}\}$. Since we have $\vdash_t M\colon \Gamma$ (hypothesis) and (ii) $t \in \operatorname{dom} M(p)$, then there exist a typing Γ' and a flag a such that (iii) $\Gamma = \Gamma' \uplus \{p\colon a\}$, and (iv) $\vdash_t M_1\colon \Gamma'$. Applying the induction hypothesis to

- (iv) $\vdash_t M_1\colon \Gamma'$,
- $\forall p \in \operatorname{dom} M_1 \implies t' \notin \operatorname{dom} M_1(p)$, from (2), and
- $\operatorname{copy}(s', t, t', M_1) = M_2$

yields that there exists a typing Γ'' such that (v) $\vdash_{t'} M'\colon \Gamma''$, (vi) $\Gamma'' \subseteq \Gamma$, and (vii) $\operatorname{dom}\Gamma'' = s'$. Let (viii) $P' \stackrel{\text{def}}{=} P \uplus \{t'\colon v\}$. Since we have (v) $\vdash_{t'} M_1\colon \Gamma''$ and $p \notin M_1$, then by Lemma 6.4.3 $p \notin \operatorname{dom}\Gamma''$ and therefore $\Gamma'' \uplus \{p\colon a\}$ is defined. Hence,

$$\frac{\text{(v) } \vdash_{t'} M_1\colon \Gamma'' \qquad \text{(viii) } P'(t') = \langle \_; a\rangle}{\vdash_t M_1 \uplus \{p\colon P'\}\colon \Gamma'' \uplus \{p\colon a\}}\ \textsc{T-perm-cons}$$

Since (vi) $\Gamma'' \subseteq \Gamma$ and $\Gamma = \Gamma' \uplus \{p\colon a\}$, then $\Gamma'' \uplus \{p\colon a\} \subseteq \Gamma$. And, finally, from (vii) $\operatorname{dom}\Gamma'' = s'$, $s = s' \uplus \{p\}$, and $\operatorname{dom}\{\Gamma'' \uplus \{a\colon p\}\} = \operatorname{dom}\Gamma'' \cup \{p\}$, then $\operatorname{dom}\Gamma'' \cup \{p\} = s$.

- Case Cpy-skip:

$$\frac{\text{(i) } \operatorname{copy}(s, t, t', M_1) = M_2 \qquad \text{(ii) } p \notin s}{\operatorname{copy}(s, t, t', M_1 \uplus \{p\colon P\}) = M_2 \uplus \{p\colon P\}}$$

where M is $M_1 \uplus \{p\colon P\}$ and M' is $M_2 \uplus \{p\colon P\}$. We test for the membership of t in P.

  - Case $t \in M_1(p)$. Since we have $\vdash_t M\colon \Gamma$ (hypothesis) and (ii) $t \in \operatorname{dom} M(p)$, then there exist a typing Γ' and a flag a such that (iii) $\Gamma = \Gamma' \uplus \{p\colon a\}$, and (iv) $\vdash_t M_1\colon \Gamma'$. Applying the induction hypothesis to (iv) $\vdash_t M_1\colon \Gamma'$, (2), and (i) $\operatorname{copy}(s, t, t', M_1) = M_2$, we get that there exists a typing Γ'' such that $\vdash_{t'} M_2\colon \Gamma''$, $\Gamma'' \subseteq \Gamma'$, and $\operatorname{dom}\Gamma'' = s$. Thus,

$$\frac{\vdash_{t'} M_2\colon \Gamma'' \qquad \dfrac{p \in \operatorname{dom} M \implies t' \notin \operatorname{dom} M(p)}{t' \notin \operatorname{dom} P}\implies}{\vdash_t M_2 \uplus \{p\colon P\}\colon \Gamma''}\ \textsc{T-perm-skip}$$

  - Case $t \notin M_1(p)$. From $\vdash_t M\colon \Gamma$ and $t \notin \operatorname{dom} M(p)$, then $\vdash_t M_1\colon \Gamma$. Applying the induction hypothesis to (iv) $\vdash_t M_1\colon \Gamma$, (2),

and (i) $\mathrm{copy}(s, t, t', M_1) = M_2$, we get that there exists a typing Γ' such that $\vdash_{t'} M_2 \colon \Gamma'$, $\Gamma' \subseteq \Gamma$, and $\operatorname{dom}\Gamma' = s$. Thus,

$$\dfrac{\vdash_{t'} M_2 \colon \Gamma' \qquad \dfrac{p \in \operatorname{dom} M \implies t' \notin \operatorname{dom} M(p)}{t' \notin \operatorname{dom} P}}{\vdash_t M_2 \uplus \{p \colon P\} \colon \Gamma'}\ \textsc{T-perm-skip} \implies$$

□

Lemma 7.1.13. *If* $\Gamma \vdash s \colon \Gamma'$ *and* $\mathrm{bounds}(s) = B$*, then* $\Gamma' \vdash B$*.*

Proof. The proof follows by induction on the typing relation $\Gamma \vdash s \colon \Gamma'$. We perform a case analysis on the derivation tree of the last rule applied.

- Case T-A-N:
$$\Gamma \vdash \emptyset \colon \emptyset$$
where s is $\emptyset$. We have that $\mathrm{bounds}(s) = \emptyset = B$ (by Definition 5.3.10), hence the case holds with rule T-B-N.

- Case T-A-C:
$$\dfrac{\text{(i) } \Gamma(p) = a \qquad \text{(ii) } \Gamma \vdash s' \colon \Gamma''}{\Gamma \vdash s' \uplus \{p\} \colon \Gamma'' \uplus \{p \colon a\}}$$
where s is $s' \uplus \{p\}$ and Γ' is $\Gamma'' \uplus \{p \colon a\}$. Let (iii) $\mathrm{bounds}(s') = B'$. We apply the induction hypothesis to (ii) $\Gamma \vdash s' \colon \Gamma''$ and (iii) to obtain (iv) $\Gamma'' \vdash B'$. Since we have (iii) $\mathrm{bounds}(s') = B'$ and $s = s' \uplus \{p\}$, then by Definition 5.3.10 $B = B' \uplus \{p \colon 0\}$. Let Γ_1 be such that $\Gamma = \Gamma_1 \uplus \{p \colon a\}$. Thus,
$$\dfrac{\dfrac{\text{(iv) } \Gamma'' \vdash B'}{\Gamma'' \uplus \{p \colon a\} \vdash B' \uplus \{p \colon 0\}}\ \textsc{T-b-c}}{\Gamma' \vdash B} \overset{\text{def}}{=}$$

□

Lemma 7.1.14. *If* $\Psi; \Gamma \vdash (B, \mathtt{async}(s, b'); b)$ *and* $\mathrm{bounds}(s) = B'$*, then there exists a typing* Γ' *such that* $\Psi; \Gamma' \vdash (B', b')$*,* $\Gamma' \subseteq \Gamma$*, and* $\operatorname{dom}\Gamma' = s$*.*

Proof. We invert the hypothesis to obtain the following premises.

$$\dfrac{\text{(i) } \Gamma \vdash B \qquad \dfrac{\dfrac{\text{(ii) } \Gamma \vdash s \colon \Gamma' \qquad \text{(iii) } \Gamma' \vdash b' \colon \emptyset}{\Gamma \vdash \mathtt{async}(s, b') \colon \Gamma} \qquad \cdots}{\Gamma \vdash \mathtt{async}(s, b'); b \colon \emptyset}}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, \mathtt{async}(s, b'); b)}$$

where Ψ is $\langle \emptyset; \emptyset \rangle$. From Lemma 6.4.1 and $\Gamma \vdash s \colon \Gamma'$, we get that (iv) $\Gamma' \subseteq \Gamma$ and (v) $\operatorname{dom} \Gamma' = s$. Next, we apply Lemma 7.1.13 to (ii) $\Gamma \vdash s \colon \Gamma'$ and $\operatorname{bounds}(s) = B'$ (hypothesis), and obtain (vi) $\Gamma' \vdash B'$. Hence, applying rule T-T-R to (vi) and (iii) we get that (vii) $\langle \emptyset; \emptyset \rangle; \Gamma' \vdash (B', b')$. The outcome of this proof consists of premises (vii), (iv), and (v). □

Lemma 7.1.15. *If* $\vdash_t M \colon \Gamma$, $\Psi; \Gamma \vdash (B, \texttt{async}(s, b'); b)$, $\forall p \in \operatorname{dom} M \implies t' \notin \operatorname{dom} M(p)$, $\operatorname{copy}(s, t, t', M) = M'$, *and* $\operatorname{bounds}(s) = B'$, *then there exists a typing* Γ' *such that* $\vdash_{t'} M' \colon \Gamma'$ *and* $\Psi; \Gamma' \vdash (B', b')$.

Proof. From Lemma 7.1.12, $\vdash_t M \colon \Gamma$, $\forall p \in \operatorname{dom} M \implies t' \notin \operatorname{dom} M(p)$, and $\operatorname{copy}(s, t, t', M) = M'$, we get that there exists a typing Γ' such that (i) $\vdash_{t'} M' \colon \Gamma'$, (ii) $\Gamma' \subseteq \Gamma$, and (iii) $\operatorname{dom} \Gamma' = s$. Applying Lemma 7.1.14 to $\Psi; \Gamma \vdash (B, \texttt{async}(s, b'); b)$ and $\operatorname{bounds}(s) = B'$ (hypothesis) we get that there exists a typing Γ'' such that $\Psi; \Gamma'' \vdash (\operatorname{bounds}(s), b')$, $\Gamma'' \subseteq \Gamma$, and $\operatorname{dom} \Gamma'' = s$. Thus, since we have $\Gamma' \subseteq \Gamma$, $\Gamma'' \subseteq \Gamma$, and $\operatorname{dom} \Gamma'' = \operatorname{dom} \Gamma' = s$, then $\Gamma' = \Gamma''$ and we conclude the proof. □

Lemma 7.1.16. *If* $\Delta; N \vdash M$, $t \notin N$, *and* $p \in \operatorname{dom} M$, *then* $t \notin \operatorname{dom} M(p)$.

Proof. Since we have $\Delta; N \vdash M$ and $p \in \operatorname{dom} M$, then by Lemma 6.3.6, we have that $\operatorname{dom} P \subseteq N$. But we know that $t \notin N$, hence $t \notin \operatorname{dom} P$. □

Lemma 7.1.17. *If*

1. $\Sigma; M \vdash T \uplus \{t \colon (B, \texttt{async}(s, b'); b)\}$,
2. $\Delta; N \vdash M$,
3. $\operatorname{copy}(s, t, t', M) = M'$,
4. $\operatorname{bounds}(s) = B'$,
5. $t' \notin N$, *and*
6. $N = \operatorname{dom} T \cup \{t\}$,

then then there exists a Σ' *such that*

$$\Sigma'; M' \vdash T \uplus \{t \colon (B, b)\} \uplus \{t' \colon (B', b')\}$$

Proof. By Lemma 6.3.2 and $\Sigma; M \vdash T \uplus \{t \colon (B, \texttt{async}(s, b'); b)\}$ we get that there exists a Ψ such that (i) $\vdash_t M \colon \Gamma$ and (ii) $\Psi; \Gamma \vdash (B, \texttt{async}(s, b'); b)$.

From Lemma 7.1.16, (2) $\Delta; N \vdash M$, (5) $t' \notin N$, then we have (iii) $\forall p \in \operatorname{dom} M \implies t' \notin \operatorname{dom} M(p)$.

Applying Lemma 7.1.15 to (i), (ii), (iii), (3) $\text{copy}(s, t, t', M) = M'$, and (4) $\text{bounds}(s) = B'$, yields that there exists a typing Γ' such that (iv) $\vdash_{t'} M' \colon \Gamma'$ and (v) $\Psi; \Gamma' \vdash (\text{bounds}(s), b')$. Applying Lemma 7.1.11 to

$$\Sigma; M \vdash T \uplus \{t \colon (B, \texttt{async}(s, b'); b)\}$$

, $\text{copy}(s, t, t', M) = M'$, and $t' \notin \text{dom}\, T \cup \{t\}$, then (vi) $\Sigma; M' \vdash T \uplus \{t \colon (B, b)\}$.

Let $T' \stackrel{\text{def}}{=} T \uplus \{t \colon (B, b)\}$. Therefore,

$$\dfrac{\dfrac{\text{(iv)} \vdash_{t'} M' \colon \Gamma' \quad \text{(v)}\ \Psi; \Gamma' \vdash (\text{bounds}(s), b') \quad \text{(vi)}\ \Sigma; M' \vdash T'}{\Sigma \uplus \{t' \colon \Psi\}; M' \vdash T' \uplus \{t' \colon (\text{bounds}(s), b')\}}\ \textsc{T-tm-c}}{\Sigma \uplus \{t' \colon \Psi\}; M' \vdash T \uplus \{t \colon (B, b)\} \uplus \{t' \colon (\text{bounds}(s), b')\}} \stackrel{\text{def}}{=}$$

□

7.2 Phaser creation

Definition 7.2.1 (Substitution function for typing)**.** *Formula* $\Gamma\sigma$ *is defined as:*

$$\dfrac{p_2 \notin \text{dom}\, \Gamma \qquad \sigma = [p_2/p_1]}{\Gamma\sigma = \{\sigma(p) \colon \Gamma(p) \mid \forall p \in \text{dom}\, \Gamma\}}$$

Lemma 7.2.1. *If* $\sigma(p) = q$*, then* $(\Gamma \uplus \{p \colon a\})\sigma = (\Gamma\sigma) \uplus \{q \colon a\}$.

Proof.

$$\begin{aligned} (\Gamma \uplus \{p \colon a\})\sigma &= \\ (\Gamma\sigma) \uplus (\{p \colon a\}\sigma) &= \\ \Gamma\sigma \uplus \{\sigma(p) \colon a\} &= \\ \Gamma\sigma \uplus \{q \colon a\} & \end{aligned}$$

□

Lemma 7.2.2. *If* $\Gamma\sigma = \Gamma'$ *and* $\sigma(p) = q$*, then* $\Gamma(p) = \Gamma'(q)$.

Proof. From $\Gamma(p) = a$, we have that $\Gamma = \Gamma'' \uplus \{p \colon a\}$. Applying Lemma 7.2.1 to $\sigma(p) = q$ and we get $(\Gamma'' \uplus \{p \colon a\})\sigma = (\Gamma''\sigma) \uplus \{q \colon a\}$. But we know that $\Gamma\sigma = \Gamma'$, thus $(\Gamma''\sigma) \uplus \{q \colon a\} = \Gamma'$ and therefore $\Gamma'(q) = a$. □

Lemma 7.2.3. *If* $p \notin s$*, then* $s[q/p] = s$.

Proof. Let $s = \{p_1, \ldots, p_n\}$ and $s[q/p] = \{\sigma(p_1), \ldots, \sigma(p_n)\}$. To show that $s = s\sigma$ it is enough to show that $p_i = \sigma(p_i)$. By Definition 5.3.4 since $p_i \neq p$, then $\sigma(p_i) = p_i$. □

Lemma 7.2.4 (Substitution for arguments). *If* $\Gamma_1 \vdash s\colon \Gamma_2$, $\sigma = [p_2/p_1]$, $p_2 \notin \operatorname{dom}\Gamma_1 \cap s$, *and* $p_1 \notin s$, *then* $\Gamma_1\sigma \vdash s\sigma\colon \Gamma_2\sigma$ *and* $\Gamma_2\sigma = \Gamma_2$.

Proof. The proof follows by induction on the derivation of $\Gamma_1 \vdash s\colon \Gamma_2$. We do a case inspection on the last rule applied.

- Case T-A-N:

$$\Gamma_1 \vdash \emptyset\colon \emptyset$$

where arguments s are $\emptyset$ and typing Γ_2 is $\emptyset$. We have that $\emptyset\sigma = \emptyset$ (by Definition 7.2.1) and $\emptyset = \emptyset\sigma$ (by Definition 5.3.4). Since $p_2 \notin \operatorname{dom}\Gamma_1$ and $p_2 \notin \operatorname{dom}\emptyset$, then, by Definition 7.2.1, $\Gamma_1\sigma$ and $\emptyset\sigma$ are both defined. The case holds by direct application of rule T-A-N.

$$\Gamma_1\sigma \vdash \emptyset\sigma\colon \emptyset\sigma$$

- Case T-A-C:

$$\frac{\text{(i) } \Gamma_1(p) = a \qquad \text{(ii) } \Gamma_1 \vdash s'\colon \Gamma_2'}{\Gamma_1 \vdash s' \uplus \{p\}\colon \Gamma_2' \uplus \{p\colon a\}}$$

where arguments s are $s' \uplus \{p\}$ and typing Γ_2 is $\Gamma_2' \uplus \{p\colon a\}$. From $\sigma = [p_2/p_1]$, $p_2 \notin \operatorname{dom}\Gamma_1$, and Definition 7.2.1, then (iii) $\Gamma_1\sigma = \Gamma_1'$. Let $\sigma(p) = q$. Next, since (iii) $\Gamma_1\sigma = \Gamma_1'$ and $\sigma(p) = q$, then from Lemma 7.2.2 we have that (iv) $\Gamma_1'(q) = a$. We have that the following premise holds.

$$\frac{\dfrac{p_1 \notin s \qquad s = s' \uplus \{p\}}{p_1 \notin s'}}{\text{(v) } s'\sigma = s'}\ \text{Lemma 7.2.3}$$

And so does premise (vi).

$$\frac{\dfrac{\Gamma_1 \vdash s'\colon \Gamma_2' \quad \sigma = [p_2/p_1] \quad p_2 \notin \operatorname{dom}\Gamma_1 \cap s'}{\Gamma_1' \vdash s'\sigma\colon \Gamma_2'}\ \text{I.H.} \qquad \text{(v) } s'\sigma = s'}{\text{(vi) } \Gamma_1' \vdash s'\colon \Gamma_2'}$$

Since $p_1 \notin s$, then $p_1 \neq p$ and therefore (vii) $\sigma(p) = p$ (by Definition 5.3.4). Hence,

$$\frac{\dfrac{\dfrac{\text{(iv) } \Gamma_1'(q) = a \quad \text{(vii) } p = q}{\Gamma_1'(p) = a} = \qquad \text{(vi) } \Gamma_1' \vdash s'\colon \Gamma_2'}{\Gamma_1' \vdash s' \uplus \{p\}\colon \Gamma_2' \uplus \{p\colon a\}}\ \text{T-A-C}}{\Gamma_1\sigma \vdash s\sigma\colon \Gamma_2} =$$

From $\Gamma_2 = \Gamma_2' \uplus \{p\colon a\}$ and $q = p$, then $\Gamma_2 = \Gamma_2' \uplus \{q\colon a\}$. And since we have $\sigma(p) = q$, then $\Gamma_2 = \Gamma_2' \uplus \{\sigma(p)\colon a\}$. Next, we have that $\Gamma_2 = \Gamma_2\sigma$, thus $\Gamma_2 = \Gamma_2'\sigma \uplus \{\sigma(p)\colon a\}$. Thus, we apply Lemma 7.2.1 to $\sigma(p) = q$ we get that $\Gamma_2'\sigma \uplus \{\sigma(p)\colon a\} = (\Gamma_2' \uplus \{p\colon a\})\sigma$. Hence, $\Gamma_2 = \Gamma_2\sigma$. □

Lemma 7.2.5 (Substitution for arguments)**.** *If* $\Gamma_1 \vdash s\colon \Gamma_2$, $\sigma = [p_2/p_1]$, *and* $p_2 \notin \operatorname{dom}\Gamma_1 \cap s$, *then* $\Gamma_1\sigma \vdash s\sigma\colon \Gamma_2\sigma$.

Proof. If $p_1 \in s$, then we conclude the proof with Lemma 7.2.4. Otherwise, $p_1 \in s$. By Lemma 6.3.1, $\Gamma_1 \vdash s\colon \Gamma_2$, and $p_1 \in s$, we have that there exist some arguments s' and a typing Γ_3 such that

(i) $s = s' \uplus \{p_1\}$,

(ii) $\Gamma_2 = \Gamma_3 \uplus \{p_1\colon a\}$,

(iii) $\Gamma_1(p_1) = a$,

(iv) $\Gamma \vdash s'\colon \Gamma_3$.

We apply Lemma 7.2.4 to (iv) $\Gamma \vdash s'\colon \Gamma_3$, $\sigma = [p_2/p_1]$ (hypothesis), $p_2 \notin \operatorname{dom}\Gamma_1 \cap s$ (hypothesis), and $p_1 \notin s$ (we get this from premise (i)), then (v) $\Gamma_1\sigma \vdash s'\sigma\colon \Gamma_3\sigma$ and (vi) $\Gamma_3\sigma = \Gamma_3$. But from Lemma 7.2.3 and $p_1 \notin s$, we have that (vii) $s'\sigma = s'$. Since we know that $p_2 \notin \operatorname{dom}\Gamma_1$, then, by Definition 7.2.1, $\Gamma_1' = \Gamma_1\sigma$. Since we have (iii) $\Sigma_1(p_1) = a$ and $\Gamma_1' = \Gamma_1\sigma$, then from Lemma 7.2.2 we have that $\Gamma_1(p_1) = \Gamma_1'(p_2)$. Thus,

$$\dfrac{\dfrac{\Gamma_1'(p_2) = a \qquad \Gamma_1' \vdash s'\colon \Gamma_3}{\Gamma_1' \vdash (s' \uplus \{p_2\})\colon \Gamma_3 \uplus \{p_2\colon a\}}\,\textsc{T-a-c}}{\Gamma_1\sigma \vdash s\sigma\colon (\Gamma_2)\sigma}\,=$$

□

Lemma 7.2.6. *If* $\Gamma\sigma = \Gamma'$, $\sigma(p) = q$, *and* $p \in \operatorname{dom}\Gamma$, *then* $\Gamma(p) = \Gamma'(q)$.

Proof. From Definition 7.2.1 we have that $\operatorname{dom}\Gamma' = \{\sigma(p) \mid p \in \operatorname{dom}\Gamma\}$. Thus, if $p \in \operatorname{dom}\Gamma$ and $\sigma(p) = q$, then $q \in \operatorname{dom}\Gamma'$. Since we have $q \in \operatorname{dom}\Gamma'$ and $\sigma(p) = q$, then from Definition 7.2.1 we have $\Gamma'(q) = \Gamma(p)$. □

Lemma 7.2.7. *For any* a *if* $\forall p \in \operatorname{dom}\Gamma\colon \Gamma(p) = a$ *and* $\Gamma' = \Gamma\sigma$, *then* $\forall p \in \operatorname{dom}\Gamma'\colon \Gamma'(p) = a$.

Proof. Let $q \in \operatorname{dom}\Gamma'$. We have that $q = \sigma(p)$ and $p \in \operatorname{dom}\Gamma$. Since $p \in \operatorname{dom}\Gamma$, then $\Gamma(p) = a$. From $\Gamma\sigma = \Gamma'$, $\sigma(p) = q$, $p \in \operatorname{dom}\Gamma$, and Lemma 7.2.6, then $\Gamma(p) = \Gamma'(q)$. □

Lemma 7.2.8 (Substitution for programs). *If* $\Gamma \vdash b \colon \Gamma'$, $p \notin \mathrm{bn}(b)$, $q \notin \mathrm{dom}\,\Gamma$, $\sigma = [q/p]$, *then* $\Gamma\sigma \vdash b\sigma \colon \Gamma'\sigma$.

Proof. The proof follows by induction on the typing relation over programs. We perform a case analysis on derivation of the inversion of $\Gamma \vdash b \colon \Gamma'$.

Case T-END.

$$\Gamma \vdash \mathtt{end} \colon \Gamma$$

where $b \stackrel{\text{def}}{=} \mathtt{end}$ and $\Gamma = \Gamma'$. By hypothesis we have that $q \notin \mathrm{dom}\,\Gamma$, so from Definition 7.2.1 $\Gamma\sigma$ is defined. By Definition 5.3.4 we have that $\mathtt{end} = \mathtt{end}\sigma$. Hence,

$$\frac{\Gamma = \Gamma' \qquad \dfrac{}{\Gamma\sigma \vdash \mathtt{end} \colon \Gamma\sigma}\ \textsc{T-end}}{\Gamma\sigma \vdash \mathtt{end}\sigma \colon \Gamma'\sigma}$$

Case T-CONS.

$$\frac{\text{(a) } \Gamma \vdash i \colon \Gamma'' \qquad \text{(b) } \Gamma'' \vdash b' \colon \Gamma'}{\Gamma \vdash i; b' \colon \Gamma'}$$

Next, we invert $\Gamma \vdash i \colon \Gamma''$, performing a case analysis on the derivation of the last rule applied.

- Case T-PHASER:

  $$\Gamma \vdash r = \mathtt{newPhaser}() \colon \Gamma \uplus \{r \colon \mathbf{u}\}$$

  where $i \stackrel{\text{def}}{=} r = \mathtt{newPhaser}()$ and $\Gamma'' \stackrel{\text{def}}{=} \Gamma \uplus \{p \colon \mathbf{u}\}$.

  We have that $r \in \mathrm{bn}(b)$ (from Definition 5.3.3), and that $p \notin \mathrm{bn}(b)$ (hypothesis), hence $p \neq r$ and therefore (i) $\sigma(r) = r$ (from Definition 5.3.4).

  We also know that $\mathrm{bn}(r = \mathtt{newPhaser}(); b') \stackrel{\text{def}}{=} \{r\} \cup \mathrm{bn}(b')$. From $p \neq r$ and the latter, we get that (ii) $p \notin \mathrm{bn}(b')$. Since $q \notin \mathrm{dom}\,\Gamma$ and $r \notin \mathrm{dom}\,\Gamma$, then (iii) $q \notin \mathrm{dom}\,\Gamma''$.

  Applying the induction hypothesis to (b) $\Gamma'' \vdash b \colon \Gamma'$, (ii) $p \notin \mathrm{bn}(b')$, (iii) $q \notin \mathrm{dom}\,\Gamma''$, $\sigma = [q/p]$ (hypothesis), and obtain that $\Gamma''\sigma \vdash b'\sigma \colon \Gamma'\sigma$. But, we know that $\Gamma'' = \Gamma \uplus \{r \colon \mathbf{u}\}$, hence $\Gamma''\sigma = (\Gamma \uplus \{r \colon \mathbf{u}\})\sigma$. From (i) $\sigma(r) = r$, the latter, and Lemma 7.2.1, we have $(\Gamma \uplus \{r \colon \mathbf{u}\})\sigma = (\Gamma\sigma) \uplus \{r \colon \mathbf{u}\}$. Hence, we have (iii) $(\Gamma\sigma) \uplus \{r \colon \mathbf{u}\} \vdash b'\sigma \colon \Gamma'\sigma$ and (iv) $r \notin \mathrm{dom}\,\Gamma\sigma$.

  The following premise (v) holds.

  $$\frac{\text{(iv) } r \notin \mathrm{dom}\,\Gamma\sigma}{\Gamma\sigma \vdash p = \mathtt{newPhaser}() \colon (\Gamma\sigma) \uplus \{r \colon \mathbf{u}\}}\ \textsc{T-phaser}$$

And therefore,

$$\frac{\text{(v) } \Gamma\sigma \vdash i\colon (\Gamma\sigma) \uplus \{r\colon \mathbf{u}\} \qquad \text{(iii) } (\Gamma\sigma) \uplus \{r\colon \mathbf{u}\} \vdash b'\sigma\colon \Gamma'\sigma}{\Gamma\sigma \vdash (i; b')\sigma\colon \Gamma'\sigma}\ \text{T-CONS}$$

- Case T-ADV and T-DEREG have the same proof structure, so we just show case T-ADV:

$$\Gamma_1 \uplus \{r\colon \mathbf{u}\} \vdash \mathtt{adv}(r)\colon \Gamma_1 \uplus \{r\colon \mathbf{a}\}$$

where $i \stackrel{\text{def}}{=} \mathtt{adv}(r)$ and $\Gamma \stackrel{\text{def}}{=} \Gamma_1 \uplus \{r\colon \mathbf{u}\}$ and $\Gamma'' \stackrel{\text{def}}{=} \Gamma_1 \uplus \{r\colon \mathbf{a}\}$.

From Definition 5.3.4 and $p \notin \mathrm{bn}(\mathtt{adv}(r); b')$ we get that (i) $p \notin \mathrm{bn}(b')$. Applying the induction hypothesis to (b) $\Gamma'' \vdash b\colon \Gamma'$, (i) $p \notin \mathrm{bn}(b')$, $\sigma = [q/p]$ (hypothesis), and obtain that $\Gamma''\sigma \vdash b'\sigma\colon \Gamma'\sigma$. But, we know that $\Gamma'' = \Gamma_1 \uplus \{r\colon \mathbf{a}\}$, hence $\Gamma''\sigma = (\Gamma_1 \uplus \{r\colon \mathbf{a}\})\sigma$. From (i) $\sigma(r) = r$, the latter, and Lemma 7.2.1, we have (ii) $(\Gamma_1 \uplus \{r\colon \mathbf{a}\})\sigma = (\Gamma\sigma) \uplus \{r\colon \mathbf{a}\}$. Hence, we have (iii) $(\Gamma_1\sigma) \uplus \{r\colon \mathbf{a}\} \vdash b'\sigma\colon \Gamma'\sigma$.

As we have (ii) $(\Gamma_1 \uplus \{r\colon \mathbf{a}\})\sigma = (\Gamma\sigma) \uplus \{r\colon \mathbf{a}\}$, then we also have (iii) $(\Gamma_1 \uplus \{r\colon \mathbf{u}\})\sigma = (\Gamma\sigma) \uplus \{r\colon \mathbf{u}\}$. Thus,

$$\frac{\dfrac{}{\Gamma_1\sigma \uplus \{p\colon \mathbf{u}\} \vdash \mathtt{adv}(r)\colon \Gamma_1\sigma \uplus \{p\colon \mathbf{a}\}}\ \text{T-ADV}}{\text{(iv) } \Gamma\sigma \vdash i\colon \Gamma''\sigma}$$

Hence,

$$\frac{\text{(iv) } \Gamma\sigma \vdash i\colon \Gamma''\sigma \qquad \text{(iii) } \Gamma''\sigma \vdash b'\sigma\colon \Gamma'\sigma}{\Gamma\sigma \vdash (i; b')\sigma\colon \Gamma'\sigma}\ \text{T-CONS}$$

- Case T-AWAIT:

$$\frac{\forall p \in \mathrm{dom}\,\Gamma\colon \Gamma(p) = \mathbf{a}}{\Gamma \vdash \mathtt{await}\colon \Gamma}$$

where $i \stackrel{\text{def}}{=} \mathtt{await}$ and $\Gamma = \Gamma''$.

From Definition 5.3.3 and $p \notin \mathrm{bn}(b)$, we get that (i) $p \notin \mathrm{bn}(b')$. We apply the induction hypothesis to (b) $\Gamma \vdash b'\colon \Gamma'$, (i) $p \notin \mathrm{bn}(b')$, $q \notin \mathrm{dom}\,\Gamma$ (hypothesis), $\sigma = [q/p]$ (hypothesis), then (ii) $\Gamma\sigma \vdash b'\sigma\colon \Gamma'\sigma$.

From Definition 5.3.4 we have that $\mathtt{await}; b'\sigma = \mathtt{await}; (b'\sigma)$. Since we have $\forall p \in \mathrm{dom}\,\Gamma\colon \Gamma(p) = \mathbf{a}$ and Lemma 7.2.7, then

$$\forall p \in \mathrm{dom}\,(\Gamma\sigma)\colon (\Gamma\sigma)(p) = \mathbf{a}$$

and applying T-AWAIT to the latter yields that

$$\text{(iii) } \Gamma\sigma \vdash \mathtt{await}\colon \Gamma\sigma$$

Thus,

$$\frac{\text{(iii) } \Gamma\sigma \vdash i \colon \Gamma\sigma \qquad \text{(ii) } \Gamma\sigma \vdash b'\sigma \colon \Gamma'\sigma}{\Gamma\sigma \vdash (i; b')\sigma \colon \Gamma'\sigma}\ \text{T-CONS}$$

- Case T-NEXT:

$$\{p_1 \colon \mathbf{a}, \ldots, p_n \colon \mathbf{a}\} \vdash \texttt{next} \colon \{p_1 \colon \mathbf{u}, \ldots, p_n \colon \mathbf{u}\}$$

where $i \stackrel{\text{def}}{=} \texttt{next}$, $\Gamma \stackrel{\text{def}}{=} \{p_1 \colon \mathbf{a}, \ldots, p_n \colon \mathbf{a}\}$, and $\Gamma'' \stackrel{\text{def}}{=} \{p_1 \colon \mathbf{u}, \ldots, p_n \colon \mathbf{u}\}$.

From Definition 5.3.3 and $p \notin \text{bn}(b)$, we get that (i) $p \notin \text{bn}(b')$. It is easy to see that $\text{dom}\,\Gamma = \text{dom}\,\Gamma''$, hence (ii) $q \notin \text{dom}\,\Gamma''$. We apply the induction hypothesis to (b) $\Gamma'' \vdash b' \colon \Gamma'$, (i) $p \notin \text{bn}(b')$, (ii) $q \notin \text{dom}\,\Gamma$, $\sigma = [q/p]$ (hypothesis), then (iii) $\Gamma''\sigma \vdash b'\sigma \colon \Gamma'\sigma$. But we know that $\forall p \in \text{dom}\,\Gamma'' \colon \Gamma''(p) = \mathbf{u}$, then from Lemma 7.2.7, $\forall p \in \text{dom}\,\Gamma''\sigma \colon (\Gamma''\sigma)(p) = \mathbf{u}$. Thus, $\Gamma''\sigma = \{q_1 \colon \mathbf{u}, \ldots, q_n \colon \mathbf{u}\}$ and by Definition 7.2.1

$$\Gamma\sigma = \{q_1 \colon \mathbf{a}, \ldots, q_n \colon \mathbf{a}\}$$

Therefore, premise (iv) holds with rule T-AWAIT.

$$\{q_1 \colon \mathbf{a}, \ldots, q_n \colon \mathbf{a}\} \vdash \texttt{await} \colon \{q_1 \colon \mathbf{u}, \ldots, q_n \colon \mathbf{u}\}$$

Thus,

$$\frac{\dfrac{\text{(iv) } \Gamma\sigma \vdash \texttt{await} \colon \Gamma''\sigma \qquad \text{(iii) } \Gamma''\sigma \vdash b'\sigma \colon \Gamma'\sigma}{\Gamma\sigma \vdash \texttt{await}; (b'\sigma) \colon \Gamma'\sigma}\ \text{T-CONS}}{\Gamma\sigma \vdash (\texttt{await}; b')\sigma \colon \Gamma'\sigma}\ \text{Definition 5.3.4}$$

- Case T-ASYNC:

$$\frac{\text{(i) } \Gamma \vdash s \colon \Gamma_1 \qquad \text{(ii) } \Gamma_1 \vdash b_1 \colon \emptyset}{\Gamma \vdash \texttt{async}(s, b_1) \colon \Gamma}$$

where $i \stackrel{\text{def}}{=} \texttt{async}(s, b_1)$ and $\Gamma'' = \Gamma$. Since we know that (i) $\Gamma \vdash s \colon \Gamma_1$, then, by Lemma 6.4.1, we have that $\Gamma_1 \subseteq \Gamma$ and since $q \notin \text{dom}\,\Gamma$, then (iii) $q \notin s$. And according to Definition 7.1.2 since we have that $q \notin \text{dom}\,\Gamma$ (hypothesis), then (iv) $q \notin \text{dom}\,\Gamma_1$. From (iii) and (iv) we have that $q \notin \text{dom}\,\Gamma_1 \cup s$.

The following premise (v) holds.

$$\frac{\text{(i) } \Gamma \vdash s \colon \Gamma_1 \quad \sigma = [q/p] \quad \text{(iv) } q \notin \text{dom}\,\Gamma_1 \cap s}{\Gamma_1\sigma \vdash (s\sigma) \colon (\Gamma_2\sigma)}\ \text{Lemma 7.2.5}$$

From $p \notin \mathrm{bn}(\mathtt{async}(s, b_1); b)$ and Definition 5.3.4, we have

$$p \notin s \cup \mathrm{bn}(b) \cup \mathrm{bn}(b_1)$$

Thus, (vi) $p \notin \mathrm{bn}(b')$ and (vii) $p \notin \mathrm{bn}(b_1)$. Applying the induction hypothesis to (ii) $\Gamma_1 \vdash b_1 \colon \emptyset$, (vi) $p \notin \mathrm{bn}(b_1)$, (iv) $q \notin \mathrm{dom}\,\Gamma_1$, and $\sigma = [q/p]$, yields (vii) $\Gamma_1\sigma \vdash b_1\sigma \colon \emptyset$.

Hence, the following premise (viii) holds.

$$\frac{\text{(v) } (\Gamma\sigma) \vdash (s\sigma) \colon (\Gamma_1\sigma) \qquad \text{(vii)}\Gamma_1\sigma \vdash b_1\sigma \colon \emptyset}{\Gamma\sigma \vdash \mathtt{async}(s, (b_1\sigma)) \colon \Gamma\sigma}\ \textsc{T-async}$$

Applying the induction hypothesis to (b) $\Gamma \vdash b' \colon \Gamma'$, (vi) $p \notin \mathrm{bn}(b')$, $q \notin \mathrm{dom}\,\Gamma$ (hypothesis), $\sigma = [q/p]$ (hypothesis), results in (ix) $\Gamma\sigma \vdash b'\sigma \colon \Gamma'\sigma$.

And therefore,

$$\dfrac{\dfrac{\text{(iv) } \Gamma\sigma \vdash \mathtt{async}(s, (b_1\sigma)) \colon \Gamma\sigma \qquad \text{(ix) } \Gamma\sigma \vdash b'\sigma \colon \Gamma'\sigma}{\Gamma\sigma \vdash \mathtt{async}(s, (b_1\sigma)); (b'\sigma) \colon \Gamma'\sigma}\ \textsc{T-cons}}{\Gamma\sigma \vdash (i; b')\sigma \colon \Gamma'\sigma}\ \text{Definition 5.3.4}$$

- Case T-FINISH. the proof for this case follows a similar, yet simpler, structure than case T-ASYNC.

□

Lemma 7.2.9. *If* $N \vdash \Delta$, $\Delta; N \vdash M$, $q \notin \mathrm{dom}\,M$, *and* $t \in N$, *then* $\Delta; N \vdash M \uplus \{q \colon P\}$, *where* $P = \{t \colon \langle 0; \mathbf{u}\rangle\}$.

Proof. We have that $\Delta \vdash P$:

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash \emptyset}\ \textsc{D-ph-nil} \qquad \dfrac{}{\Delta; t; 0 \vdash \emptyset}\ \textsc{D-l-nil}}{\Delta \vdash \emptyset \uplus \{t \colon \langle 0; \mathbf{u}\rangle\}}\ \textsc{D-ph-cons}}{\Delta \vdash P}\ \stackrel{\text{def}}{=}$$

Hence,

$$\dfrac{\Delta \vdash P \qquad \dfrac{\mathrm{dom}\,P = \{t\} \quad t \in N}{\mathrm{dom}\,P \subseteq N} \qquad \Delta; N \vdash M}{\Delta; N \vdash M \uplus \{p \colon P\}}\ \textsc{T-p-map-cons}$$

□

Lemma 7.2.10. *If* $p \notin \operatorname{dom}\Gamma$, *then* $\Gamma[q/p] = \Gamma$.

Proof. The proof follows by induction on the structure of Γ. We perform a case analysis on the structure of Γ.

- Case Γ is $\emptyset$. Let $\sigma = [q/p]$. The case holds by Definition 7.2.1, as $\emptyset\sigma = \emptyset$.
- Case Γ is $\Gamma' \uplus \{r\colon a\}$. Let $\sigma = [q/p]$. Applying the induction hypothesis to $p \notin \operatorname{dom}\Gamma'$ (as $p \notin \operatorname{dom}\Gamma$), we have that (i) $\Gamma'\sigma = \Gamma'$. Since $p \notin \operatorname{dom}\Gamma$, then $r \neq p$, and thus from Definition 7.2.1 (ii) $\sigma(r) = r$. Thus,

$$\begin{aligned} \Gamma &= \Gamma' \uplus \{r\colon a\} \\ &= (\Gamma'\sigma) \uplus \{\sigma(r)\colon a\} && \text{(i),(ii)} \\ &= (\Gamma' \uplus \{r\colon a\})\sigma && \text{(Definition 7.2.1)} \\ &= \Gamma\sigma && \text{(hyp.)} \end{aligned}$$

□

Lemma 7.2.11. *If*

1. $\langle\emptyset;\emptyset\rangle; \Gamma \vdash (B, p = \texttt{newPhaser}(); b)$,
2. $q \notin \operatorname{bn}(b)$, *and*
3. $q \notin \operatorname{dom}\Gamma$,

then $\langle\emptyset;\emptyset\rangle; \Gamma \uplus \{q\colon \mathbf{u}\} \vdash (B \uplus \{q\colon 0\}, b[q/p])$.

Proof. By inverting hypothesis (1), we get the following two premises.

$$\dfrac{\text{(i) } \Gamma \vdash B \qquad \dfrac{\cdots \quad \text{(ii) } \Gamma \uplus \{p\colon \mathbf{u}\} \vdash b\colon \emptyset}{\Gamma \vdash p = \texttt{newPhaser}(); b\colon \emptyset}}{\langle\emptyset;\emptyset\rangle; \Gamma \vdash (B, p = \texttt{newPhaser}(); b)}$$

Let $\sigma = [q/p]$ and $\Gamma_1 \stackrel{\text{def}}{=} \Gamma \uplus \{p\colon \mathbf{u}\}$.

We know that $q \notin \operatorname{bn}(p = \texttt{newPhaser}(); b)$ by hypothesis, hence $q \neq p$. Since $q \neq p$ and $q \notin \operatorname{dom}\Gamma$, then (iii) $q \notin \operatorname{dom}\Gamma_1$. Applying the substitution Lemma 7.2.8 for programs to (ii) $\Gamma_1 \vdash b\colon \emptyset$, (2) $q \notin \operatorname{bn}(b)$, (iii) $q \notin \operatorname{dom}\Gamma_1$, and $\sigma = [q/p]$, yields $\Gamma_1\sigma \vdash b\sigma\colon \emptyset$.

As (3) $p \notin \operatorname{dom}\Gamma$, we have that $\Gamma\sigma = \Gamma$, from Lemma 7.2.10. Hence, $\Gamma_1\sigma = \Gamma \uplus \{q\colon \mathbf{u}\}$, by Definition 7.2.1. Therefore, we have (iv): $\Gamma \uplus \{q\colon \mathbf{u}\} \vdash$

$b\sigma : \emptyset \stackrel{\text{def}}{=} \Gamma \uplus \{q : \mathbf{u}\} \vdash b[q/p] : \emptyset$. And concluding, the next tree holds.

$$\dfrac{\dfrac{\text{(i) } \Gamma \vdash B}{\Gamma \uplus \{q : \mathbf{u}\} \vdash B \uplus \{q : 0\}}\ \text{T-B-C} \qquad \text{(iv) } \Gamma \uplus \{q : \mathbf{u}\} \vdash b[q/p] : \emptyset}{\langle \emptyset ; \emptyset \rangle ; \Gamma \uplus \{q : \mathbf{u}\} \vdash (B \uplus \{q : 0\}, b[q/p])}\ \text{T-T-R}$$

□

Lemma 7.2.12. *If* $\Sigma ; M \vdash T$, $p \notin \operatorname{dom} M$, *and* $\operatorname{dom} T \cap \operatorname{dom} P = \emptyset$, *then* $\Sigma ; M \uplus \{p : P\} \vdash T$.

Proof. The proof follows by induction on the typing relation. We perform a case analysis on the derivation of the last rule applied.

- Case T-TM-C:

$$\dfrac{\vdash_t M : \Gamma \qquad \Psi ; \Gamma \vdash \tau \qquad \Sigma' ; M \vdash T'}{\Sigma' \uplus \{t : \Psi\} ; M \vdash T' \uplus \{t : \tau\}}$$

  where Σ is $\Sigma' \uplus \{t : \Psi\}$ and T is $T' \uplus \{t : \tau\}$. Since $\operatorname{dom} T \cap \operatorname{dom} P = \emptyset$, then $t \notin \operatorname{dom} P$. Applying the induction hypothesis to $\Sigma' ; M \vdash T'$, $p \notin \operatorname{dom} M$, and $\operatorname{dom} T \cap \operatorname{dom} P = \emptyset$ yields $\Sigma' ; M \uplus \{p : P\} \vdash T'$. We conclude the case with rule T-TM-C:

$$\dfrac{\dfrac{\vdash_t M : \Gamma \quad t \notin \operatorname{dom} P}{\vdash_t M : \Gamma}\ \text{T-PERM-SKIP} \qquad \Psi ; \Gamma \vdash \tau \quad \Sigma' ; M \uplus \{p : P\} \vdash T'}{\Sigma' \uplus \{t : \Psi\} ; M \uplus \{p : P\} \vdash T' \uplus \{t : \tau\}}$$

- Case T-TM-N:

$$\emptyset ; M \vdash \emptyset$$

  where Σ is $\emptyset$ and T is $\emptyset$. The case concludes by direct application of rule T-TM-N.

□

Lemma 7.2.13 (Substitution for tasks)**.** *If*

1. $\Sigma ; M \vdash T \uplus \{t : (B, p = \texttt{newPhaser}() ; b)\}$,
2. $q \notin \operatorname{dom} M$, *and*
3. $q \notin \operatorname{bn}(b)$,

then $\Sigma; M \uplus \{q\colon \{t\colon \langle 0; \mathbf{u}\rangle\}\} \vdash T \uplus \{t\colon (B \uplus \{q\colon 0\}, b[q/p])\}$.

Proof. Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon \tau_1\}$, $\tau_1 \stackrel{\text{def}}{=} (B, p = \texttt{newPhaser}(); b)$, and $M_2 \stackrel{\text{def}}{=} M \uplus \{q\colon \{t\colon \langle 0; \mathbf{u}\rangle\}\}$. Applying Lemma 6.3.2 to the hypothesis $\Sigma; M \vdash T_1$ and $T_1(t) = \tau_1$ (by definition), yields (i) $\Sigma = \Sigma_1 \uplus \{t\colon \Psi\}$, $T_1 = T \uplus \{t\colon \tau_1\}$, (ii) $\vdash_t M\colon \Gamma$, (iii) $\Psi; \Gamma \vdash \tau_1$, and (iv) $\Sigma_1; M \vdash T$.

To prove this lemma, it is enough to show the following three premises, which we use to prove that $\Sigma \uplus \{t\colon \Psi\}; M_2 \vdash T \uplus \{t\colon \tau_2\}$ holds, where $\tau_2 \stackrel{\text{def}}{=} (B \uplus \{q\colon 0\}, b[q/p])$.

(a) Show that $\vdash_t M_2\colon \Gamma \uplus \{q\colon \mathbf{u}\}$. From Lemma 6.4.2 and (ii) $\vdash_t M\colon \Gamma$, yields that $\operatorname{dom}\Gamma \subseteq \operatorname{dom} M$. Since (2) $q \notin \operatorname{dom} M$, then we have (iv) $q \notin \operatorname{dom}\Gamma$. Thus,

$$\dfrac{\dfrac{\text{(ii)}\ \vdash_t M\colon \Gamma \qquad P(t) = \langle 0; \mathbf{u}\rangle \qquad q \notin \operatorname{dom}\Gamma}{\vdash_t M \uplus \{q\colon P\}\colon \Gamma \uplus \{q\colon \mathbf{u}\}}\ \text{T-PERM-CONS} \qquad \stackrel{\text{def}}{=}}{\vdash_t M_2\colon \Gamma \uplus \{q\colon \mathbf{u}\}}$$

(b) Show that $\Psi; \Gamma \uplus \{q\colon \mathbf{u}\} \vdash \tau_2$. By inverting (iii) $\Psi; \Gamma \vdash \tau_1$, we get that $\Psi = \langle \emptyset; \emptyset\rangle$. Applying Lemma 7.2.11, (iii) $\langle \emptyset; \emptyset\rangle; \Gamma \vdash \tau_1$, (2) $q \notin \operatorname{bn}(b)$, and (iv) $q \notin \operatorname{dom}\Gamma$, then $\Psi; \Gamma \uplus \{q\colon \mathbf{u}\} \vdash \tau_2$.

(c) Show that $\Sigma_1; M_2 \vdash T$. Applying Lemma 7.2.12 to (iv) $\Sigma_1; M \vdash T$, (2) $q \notin \operatorname{dom} M$, and $\operatorname{dom} T \cap \{t\} = \emptyset$ (since $t \notin \operatorname{dom} T$), yields that $\Sigma_1; M_2 \vdash T$.

We apply rule T-TM-C to (a), (b), and (c) to conclude this proof. □

7.3 Deregistration

Lemma 7.3.1. *If* $N \vdash \Delta$, $\Delta; N \vdash M_1$, $M_1(p) = P_1$, *and* $t \in \operatorname{dom} P_1$, *then there exist a phaser map* M_2, *phaser* P_2 *such that:*

1. $M_1 = M_2 \uplus \{p\colon P_1\}$,
2. $P_1 = P_2 \uplus \{t\colon v\}$, *and*
3. $\Delta; N \vdash M_2 \uplus \{p\colon P_2\}$.

Proof. By Lemma 6.3.6, $\Delta; N \vdash M_1$, and $M_1(p) = P_1$, we get that there exists a phaser map M_2 such that (i) $M_1 = M_2 \uplus \{p\colon P_1\}$, (ii) $\Delta \vdash P_1$, (iii) $\operatorname{dom} P_1 \subseteq N$, and (iv) $\Delta; N \vdash M_2$. Let $P_1(t) = v$. Since we have $N \vdash \Delta$, $\Delta \vdash P_1$, $P_1(t) = v$, then there exists P_2 such that (2) $P_1 = P_2 \uplus \{p\colon v\}$, (v) $\Delta \vdash P_2$.

Hence,

$$\dfrac{\text{(v) } \Delta \vdash P_2 \qquad \dfrac{\dfrac{\text{(2) } P_1 = P_2 \uplus \{t\colon v\}}{\operatorname{dom} P_2 \subseteq \operatorname{dom} P_1} \qquad \text{(iii) } \operatorname{dom} P_1 \subseteq N}{\operatorname{dom} P_2 \subseteq N} \qquad \text{(iv) } \Delta; N \vdash M_2}{\text{(3) } \Delta; N \vdash M \uplus \{p\colon P_2\}}$$

□

Lemma 7.3.2. *If* $\emptyset; \Gamma \uplus \{p\colon a\} \vdash (B \uplus \{p\colon n\}, \texttt{dereg}(p); b)$*, then*

$$\emptyset; \Gamma \vdash (B, b)$$

Proof. We invert the hypothesis to obtain the following premises.

$$\dfrac{\text{(i) } \Gamma \uplus \{p\colon a\} \vdash B \uplus \{p\colon n\} \qquad \dfrac{\text{(ii) } \Gamma \vdash b\colon \emptyset}{\Gamma \uplus \{p\colon a\} \vdash \texttt{dereg}(p); b\colon \emptyset}}{\langle \emptyset; \emptyset \rangle; \Gamma \uplus \{p\colon a\} \vdash (B \uplus \{p\colon n\}, \texttt{dereg}(p); b)}$$

Further, we apply Lemma 6.3.3 to $\Gamma \uplus \{p\colon a\} \vdash B \uplus \{p\colon n\}$ to obtain (iii) $\Gamma \vdash B$. Hence,

$$\dfrac{\text{(iii) } \Gamma \vdash B \qquad \text{(ii) } \Gamma \vdash b\colon \emptyset}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, b)}$$

□

Lemma 7.3.3. *If*

$$\Sigma; M \uplus \big\{p\colon P \uplus \{t\colon v\}\big\} \vdash T \uplus \{t\colon (B \uplus \{p\colon n\}, \texttt{dereg}(p); b)\}$$

then there exists Σ' *such that*

$$\Sigma'; M \uplus \{p\colon P\} \vdash T \uplus \{t\colon (B, b)\}$$

Proof. Let $M_1 \stackrel{\text{def}}{=} M \uplus \big\{p\colon P \uplus \{t\colon v\}\big\}$, $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon \tau_1\}$, and

$$\tau_1 \stackrel{\text{def}}{=} (B \uplus \{p\colon n\}, \texttt{dereg}(p); b)$$

Applying Lemma 6.3.2 to the hypothesis $\Sigma; M_1 \vdash T_1$ and $T_1(t) = \tau_1$ (by definition), yields (i) $\Sigma = \Sigma_1 \uplus \{t\colon \Psi_1\}$, (ii) $\vdash_t M_1\colon \Gamma_1$, (iii) $\Psi_1; \Gamma_1 \vdash \tau_1$, and (iv) $\Sigma_1; M \vdash T$.

Let $M_2 \stackrel{\text{def}}{=} M \uplus \{p\colon P\}$, $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon \tau_2\}$, and $\tau_2 \stackrel{\text{def}}{=} (B, b)$.

(a) Show that $\vdash_t M_2 \colon \Gamma_2$ holds. Since $\vdash_t M_1 \colon \Gamma_1$ and $M_1(p)(t) = \langle n; a \rangle$, then by Lemma 6.3.5 we have (v) $\Gamma_1 = \Gamma_2 \uplus \{p \colon a\}$ and $\vdash_t M \colon \Gamma_2$. We have that $p \notin \operatorname{dom} P$. Hence,

$$\dfrac{\dfrac{\vdash_t M \colon \Gamma_2 \qquad p \notin \operatorname{dom} P}{\vdash_t M \uplus \{p \colon P\} \colon \Gamma_2}\,\textsc{T-perm-skip}}{\vdash_t M_2 \colon \Gamma_2}\,\stackrel{\text{def}}{=}$$

(b) Show that $\Psi; \Gamma_2 \vdash \tau_2$ holds. By inverting (iii) $\Psi_1; \Gamma_1 \vdash \tau_1$, we get that $\Psi_1 = \emptyset$. Applying Lemma 7.3.2 to $\emptyset; \Gamma_2 \uplus \{p \colon a\} \vdash \tau_1$ yields $\emptyset; \Gamma_2 \vdash \tau_2$.

(c) Show that $\Sigma_2; M_2 \vdash T$ holds. Applying Lemma 6.5.3 to (iv) $\Sigma_1; M_1 \vdash T$, $t \notin \operatorname{dom} P$ yields that $\Sigma_1; M_2 \vdash T$.

We apply rule T-TM-C to (a), (b), and (c) to conclude this proof. □

7.4 Advance phase

Lemma 7.4.1. *If $N \vdash \Delta$ and $\Delta \vdash P \uplus \{t \colon (n, \mathbf{u})\}$, then $\Delta \vdash P \uplus \{t \colon (n, \mathbf{a})\}$.*

Proof. Let $P_1 = P \uplus \{t \colon (n, \mathbf{u})\}$. Applying Lemma 6.3.8 to $N \vdash \Delta$, $\Delta \vdash P_1$, $P_1(t) = \langle n; \mathbf{u} \rangle$, then (i) $\Delta \vdash P$, and (ii) $\Delta; t; n \vdash P$. Applying rule D-PH-CONS to (i) and (ii) yields that $\Delta \vdash P \uplus \{t \colon (n, \mathbf{a})\}$ holds. □

Lemma 7.4.2. *If $N \vdash \Delta$ and $\Delta; N \vdash M \uplus \{p \colon P \uplus \{t \colon \langle n; \mathbf{u} \rangle\}\}$, then $\Delta; N \vdash M \uplus \{p \colon P \uplus \{t \colon \langle n; \mathbf{a} \rangle\}\}$.*

Proof. We use Lemma 6.3.6 to obtain

(i) $\Delta \vdash P \uplus \{t \colon (n, \mathbf{u})\}$,

(i) $\operatorname{dom} P \uplus \{t \colon (n, \mathbf{u})\} \subseteq N$, and

(i) $\Delta; N \vdash M$.

From Lemma 7.4.1, $N \vdash \Delta$, and (i)$\Delta \vdash P \uplus \{t \colon (n, \mathbf{u})\}$, we get that (iv) $\Delta \vdash P \uplus \{t \colon (n, \mathbf{a})\}$. We have that $\operatorname{dom} P \uplus \{t \colon (n, \mathbf{u})\} = \operatorname{dom} P \uplus \{t \colon (n, \mathbf{a})\}$, hence we have (v) $\operatorname{dom} P \uplus \{t \colon (n, \mathbf{a})\} \subseteq N$. Hence, we conclude this proof applying rule T-P-MAP-CONS to (iv), (v), and (iii). □

Lemma 7.4.3. *If $\vdash_t M \uplus \{p \colon P \uplus \{t \colon (n, \mathbf{u})\}\} \colon \Gamma$, then there exists a typing Γ' such that $\Gamma = \Gamma' \uplus \{p \colon \mathbf{u}\}$ and*

$$\vdash_t M \uplus \{p \colon P \uplus \{t \colon (n, \mathbf{a})\}\} \colon \Gamma' \uplus \{p \colon \mathbf{a}\}$$

Proof. Applying Lemma 6.3.5 to the hypothesis, we get that there exists a typing Γ' such that $\Gamma = \Gamma' \uplus \{p\colon \mathbf{a}\}$ and (i) $\vdash_t M\colon \Gamma'$. Let $P' \stackrel{\text{def}}{=} P \uplus \{t\colon (n, \mathbf{a})\}$. We know that (ii) $P'(t) = (n, \mathbf{a})$. Thus,

$$\frac{\text{(i) } \vdash_t M\colon \Gamma' \qquad \text{(ii) } P'(t) = (\_, \mathbf{a})}{\vdash_t M \uplus \{p\colon P'\}\colon \Gamma' \uplus \{p\colon \mathbf{a}\}}\ \textsc{T-perm-cons}$$

□

Lemma 7.4.4. *If* $\vdash_{t'} M \uplus \{p\colon P \uplus \{t\colon \langle n; \mathbf{u}\rangle\}\}\colon \Gamma$ *and* $t \neq t'$*, then*

$$\vdash_{t'} M \uplus \{p\colon P \uplus \{t\colon \langle n; \mathbf{a}\rangle\}\}\colon \Gamma$$

Proof. Let $M_1 \stackrel{\text{def}}{=} M \uplus \{p\colon P \uplus \{t\colon \langle n; \mathbf{u}\rangle\}\}$. We test if $t' \in M_1(p)$.

- Case $t' \in M_1(p)$. Applying Lemma 6.3.5 to the hypothesis, we get that there exists a typing Γ' such that $\Gamma = \Gamma' \uplus \{p\colon a\}$ and (i) $\vdash_{t'} M\colon \Gamma'$. Let $P' \stackrel{\text{def}}{=} P \uplus \{t\colon \langle n; \mathbf{a}\rangle\}$. We conclude applying rule T-perm-cons to (i) and $P'(t') = \langle n; a\rangle$ (as $t \neq t'$).
- Case $t' \notin M_1(p)$. Applying Lemma 6.3.4 to the hypothesis, we get that (i) $\vdash_{t'} M\colon \Gamma$. Let $P' \stackrel{\text{def}}{=} P \uplus \{t\colon \langle n; \mathbf{a}\rangle\}$. The case concludes with the application of rule T-perm-skip to $t' \notin \operatorname{dom} P'$ (as $t \neq t'$).

□

Lemma 7.4.5. *If* $\Psi; \Gamma \vdash (B, \mathtt{adv}(p); b)$*, then there exists a typing* Γ' *such that* $\Gamma = \Gamma' \uplus \{p\colon \mathbf{u}\}$ *and* $\Psi; \Gamma' \uplus \{p\colon \mathbf{a}\} \vdash (B, b)$.

Proof. By inverting $\Psi; \Gamma \vdash (B, \mathtt{adv}(p); b)$ we get the next premises.

$$\frac{\text{(i) } \Gamma \vdash B \qquad \dfrac{\Gamma' \uplus \{p\colon \mathbf{u}\} \vdash \mathtt{adv}(p)\colon \Gamma' \uplus \{p\colon \mathbf{a}\} \qquad \text{(ii) } \Gamma' \uplus \{p\colon \mathbf{a}\} \vdash b\colon \emptyset}{\Gamma' \uplus \{p\colon \mathbf{u}\} \vdash \mathtt{adv}(p); b\colon \emptyset}}{\langle \emptyset; \emptyset\rangle; \Gamma' \uplus \{p\colon \mathbf{u}\} \vdash (B, \mathtt{adv}(p); b)}$$

where $\Gamma = \Gamma' \uplus \{p\colon \mathbf{u}\}$. We apply Lemma 6.3.3 to (i) $\Gamma \vdash B$ and $p \in \operatorname{dom}\Gamma$, and get (iii) $B = B' \uplus \{p\colon n\}$, and (iv) $\Gamma' \vdash B'$. Hence,

$$\frac{\dfrac{\text{(iv) } \Gamma' \vdash B'}{\Gamma' \uplus \{p\colon \mathbf{a}\} \vdash B' \uplus \{p\colon n\}}\ \textsc{T-b-c} \qquad \text{(ii) } \Gamma' \uplus \{p\colon \mathbf{a}\} \vdash b\colon \emptyset}{\Gamma' \uplus \{p\colon \mathbf{a}\} \vdash B}\ = \qquad \frac{}{\langle \emptyset; \emptyset\rangle; \Gamma' \uplus \{p\colon \mathbf{a}\} \vdash (B, b)}\ \textsc{T-t-r}$$

□

Lemma 7.4.6. *If*

- $\Sigma; M \uplus \{p\colon P \uplus \{t\colon (n, \mathbf{u})\}\} \vdash T$ *and*
- $t \notin \operatorname{dom} T$*,*

then $\Sigma; M \uplus \{p\colon P \uplus \{t\colon (n, \mathbf{a})\}\} \vdash T$*.*

Proof. The proof follows by induction on the typing relation. We do a case analysis on the derivation of the last rule applied.

- Case T-TM-N:

$$\emptyset; M \vdash \emptyset$$

  where Σ is $\emptyset$ and T is $\emptyset$. The case concludes by direct application of rule T-TM-N.

- Case T-TM-C:

$$\frac{\text{(i) } \vdash_{t'} M_1\colon \Gamma \qquad \text{(ii) } \Psi; \Gamma \vdash \tau \qquad \text{(iii) } \Sigma'; M_1 \vdash T'}{\Sigma' \uplus \{t'\colon \Psi\}; M_1 \vdash T' \uplus \{t'\colon \tau\}}$$

  where Σ is $\Sigma' \uplus \{t'\colon \Psi\}$, M_1 is $M \uplus \{p\colon P \uplus \{t\colon \langle n; \mathbf{u}\rangle\}\}$, and T is $T' \uplus \{t'\colon \tau\}$. Let $M_2 \stackrel{\text{def}}{=} M \uplus \{p\colon P \uplus \{t\colon \langle n; \mathbf{a}\rangle\}\}$. From Lemma 7.4.4, $\vdash_{t'} M_1\colon \Gamma$ and $t \neq t'$ (as $t \notin \operatorname{dom} T$), we get (iv) $\vdash_{t'} M_2\colon \Gamma$. Applying the induction hypothesis to (iii) $\Sigma'; M_1 \vdash T'$ we obtain (v) $\Sigma'; M_2 \vdash T'$. We conclude the proof applying rule T-TM-C to (iv), (ii), and (v).

□

Lemma 7.4.7. *If*

$$\Sigma; M \uplus \{p\colon P \uplus \{t\colon \langle n; \mathbf{u}\rangle\}\} \vdash T \uplus \{t\colon (B, \mathtt{adv}(p); b)\}$$

then

$$\Sigma; M \uplus \{p\colon P \uplus \{t\colon \langle n; \mathbf{a}\rangle\}\} \vdash T \uplus \{t\colon (B, b)\}$$

Proof. Let $M_1 \stackrel{\text{def}}{=} M \uplus \{p\colon P \uplus \{t\colon \langle n; \mathbf{u}\rangle\}\}$. From Lemma 6.3.2 and the hypothesis we have that (i) $\Sigma = \Sigma'' \uplus \{t\colon \Psi\}$, (ii) $\vdash_t M_1\colon \Gamma$, (iii) $\Psi; \Gamma \vdash \tau$, and (iv) $\Sigma''; M_1 \vdash T$.

From Lemma 7.4.3 and $\vdash_t M_1\colon \Gamma$, we get that there exists a typing Γ'' such that (v) $\Gamma = \Gamma'' \uplus \{p\colon \mathbf{u}\}$, (vi) $\vdash_t M_2\colon \Gamma'' \uplus \{p\colon \mathbf{a}\}$. Let $\Gamma' \stackrel{\text{def}}{=} \Gamma'' \uplus \{p\colon \mathbf{a}\}$. Next, we apply Lemma 7.4.5 to (iii) $\Psi; \Gamma \vdash \tau$ and get (vii) $\Psi; \Gamma' \vdash (B, b)$.

Applying Lemma 7.4.6 to (iv) $\Sigma''; M_1 \vdash T$ and $t \notin \operatorname{dom} T$, yields that (viii) $\Sigma''; M_2 \vdash T$. Therefore,

$$\frac{\text{(vi) } \vdash_t M_2 \colon \Gamma' \qquad \text{(vii) } \Psi; \Gamma' \vdash (B, b) \qquad \text{(viii) } \Sigma''; M_2 \vdash T}{\Sigma'' \uplus \{t \colon \Psi\}; M_2 \vdash T \uplus \{t \colon (B, b)\}}$$

□

7.5 Change bound

Lemma 7.5.1. *If* $\Psi; \Gamma \vdash (B \uplus \{p \colon \_\}, \texttt{bound}(p); b)$, *then* $\Psi; \Gamma \vdash (B \uplus \{p \colon n\}, b)$.

Proof. By inverting the hypothesis we get the next premises.

$$\frac{\text{(i) } \Gamma \vdash B \uplus \{p \colon \_\} \qquad \dfrac{\cdots \qquad \text{(ii) } \Gamma \vdash b \colon \emptyset}{\Gamma \vdash \texttt{bound}(p); b \colon \emptyset}}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B \uplus \{p \colon \_\}, \texttt{bound}(p); b)}$$

We apply Lemma 6.3.3 to (i) $\Gamma \vdash B \uplus \{p \colon \_\}$, and get that there exists a typing Γ' such that (iii) $\Gamma = \Gamma' \uplus \{p \colon a\}$ and (iv) $\Gamma' \vdash B$. Hence,

$$\dfrac{\text{(iii) } \Gamma = \Gamma' \uplus \{p \colon a\} \qquad \dfrac{\text{(iv) } \Gamma' \vdash B}{\Gamma' \uplus \{p \colon a\} \vdash B \uplus \{p \colon n\}}\ \textsc{T-B-C}}{\dfrac{\Gamma \vdash B \uplus \{p \colon n\} \qquad \text{(ii) } \Gamma \vdash b \colon \emptyset}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B \uplus \{p \colon n\}, b)}\ \textsc{T-T-R}} =$$

□

Lemma 7.5.2. *If*

$$\Sigma; M \vdash T \uplus \{t \colon (B \uplus \{p \colon \_\}, \texttt{bound}(p); b)\}$$

then

$$\Sigma; M \vdash T \uplus \{t \colon (B \uplus \{p \colon n\}, b)\}$$

Proof. From Lemma 6.3.2 and the hypothesis we have that (i) $\Sigma = \Sigma'' \uplus \{t \colon \Psi\}$, (ii) $\vdash_t M \colon \Gamma$, (iii) $\Psi; \Gamma \vdash \tau$, and (iv) $\Sigma''; M \vdash T$. Next, we apply Lemma 7.5.1 to (iii) $\Psi; \Gamma \vdash \tau$ and get (v) $\Psi; \Gamma \vdash (B \uplus \{p \colon n\}, b)$. Therefore,

$$\frac{\text{(ii) } \vdash_t M \colon \Gamma \qquad \text{(v) } \Psi; \Gamma \vdash (B \uplus \{p \colon n\}, b) \qquad \text{(iv) } \Sigma''; M \vdash T}{\Sigma'' \uplus \{t \colon \Psi\}; M \vdash T \uplus \{t \colon (B \uplus \{p \colon n\}, b)\}}$$

□

7.6 Await

Lemma 7.6.1. *If* $\Psi; \Gamma \vdash (B, \texttt{await}; b)$*, then* $\Psi; \Gamma \vdash (B, b)$*.*

Proof. Inverting the hypothesis yields the following premises.

$$\frac{\text{(i) } \Gamma \vdash B \qquad \dfrac{\cdots \quad \text{(ii) } \Gamma \vdash b \colon \emptyset}{\Gamma \vdash \texttt{await}; b \colon \emptyset}}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, \texttt{await}; b)}$$

where Ψ is $\langle \emptyset; \emptyset \rangle$. We conclude the proof applying T-T-R to (i), (ii). □

Lemma 7.6.2. *If* $\Sigma; M \vdash T \uplus \{t \colon (B, \texttt{await}; b)\}$*, then*

$$\Sigma; M \vdash T \uplus \{t \colon (B, b)\}$$

Proof. Applying Lemma 6.3.2 to the hypothesis yields that (i) $\Sigma = \Sigma' \uplus \{t \colon \Psi\}$, (ii) $\vdash_t M \colon \Gamma$, (iii) $\Psi; \Gamma \vdash (B, \texttt{await}; b)$, and (iv) $\Sigma'; M \vdash T$. From (iii) $\Psi; \Gamma \vdash (B, \texttt{await}; b)$ and Lemma 7.6.1, we get that (v) $\Psi; \Gamma \vdash (B, b)$. Hence, we conclude applying rule T-TM-C to (ii), (v), and (iv). □

7.7 Next

Definition 7.7.1. *Let* $\text{commit}_{\text{D}}(\Delta, t) = \Delta'$*. Function* $\text{commit}_{\text{D}}(\Delta, t)$ *is defined by cases as follows.*

$$\Delta'(t_1, t_2) = \begin{cases} \Delta(t, t_2) + 1 & \textit{if } t_1 = t \textit{ and } t_2 \neq t \\ \Delta(t_1, t) - 1 & \textit{if } t_1 \neq t \textit{ and } t_2 = t \\ \Delta(t_1, t_2) & \textit{otherwise} \end{cases}$$

Lemma 7.7.1. *If* $N \vdash \Delta$ *and* $\Delta' = \text{commit}_{\text{D}}(\Delta, t)$*, then* $N \vdash \Delta'$*.*

Proof. By hypothesis we have

$$\frac{(N, \leq_\Delta) \text{ is a total ordering} \qquad \forall t_1, t_2 \in N. \Delta(t_1, t_2) = z \implies \Delta(t_2, t_1) = -z}{N \vdash \Delta}$$

We show that for any task names t_1 and t_2 picked from N we have that $\Delta'(t_1, t_2) = -\Delta'(t_2, t_1)$. In the first column, a cell corresponds to a sub-case where we compare the picked task names with t. In the second column, we establish that $\Delta'(t_1, t_2) = -\Delta'(t_2, t_1)$ with Definition 7.7.1.

Sub-case	$\Delta'(t_1, t_2) = -\Delta'(t_2, t_1)$
$t_1 = t \wedge t_2 = t$	$\Delta(t_1, t_2) = -\Delta(t_2, t_1)$
$t_1 \neq t \wedge t_2 = t$	$\Delta(t_1, t) - 1 = -(\Delta(t, t_1) + 1)$
$t_1 = t \wedge t_2 \neq t$	$\Delta(t, t_2) + 1 = -(\Delta(t_2, t) - 1)$
$t_1 \neq t \wedge t_2 \neq t$	$\Delta(t_1, t_2) = -\Delta(t_2, t_1)$

We now show that $(N, \leq_{\Delta'})$ is reflexive. Let $t' \in N$. By hypothesis we have $t' \leq_{\Delta} t'$, hence $\Delta(t', t') \leq_{\Delta} 0$. Given that regardless of which task name we chose $\Delta'(t', t') = \Delta(t', t')$, then we have that $t \leq_{\Delta'} t$.

Relation structure $(N, \leq_{\Delta'})$ is anti-symmetric. By definition we get that $\Delta'(t_1, t_2) \leq_{\Delta} 0$ and $\Delta'(t_1, t_2) \leq_{\Delta} 0$. Let $t_1, t_2 \in N$, $t_1 \leq_{\Delta'} t_2$, and $t_2 \leq_{\Delta'} t_1$. We need to show that $t_1 =_{\Delta'} t_2$ holds. But we have shown that $\Delta'(t_1, t_2) = -\Delta'(t_2, t_1)$, hence $\Delta'(t_1, t_2) = 0$ and therefore $t_1 \leq_{\Delta'} t_2$.

Relation structure $(N, \leq_{\Delta'})$ is compatible. Let $t_1, t_2 \in N$. We need to show that we have $t_1 \leq_{\Delta'} t_2 \vee t_2 \leq_{\Delta'} t_1$, or simply that $\Delta'(t_1, t_2) \leq_{\Delta'} 0 \vee \Delta'(t_2, t_1) \leq_{\Delta} 0$. But we have already shown that $\Delta'(t_1, t_2) = -\Delta'(t_2, t_1)$, hence $\Delta'(t_1, t_2) \leq_{\Delta'} 0 \vee \Delta'(t_2, t_1) \leq_{\Delta} 0$.

Finally, we show that $(N, \leq_{\Delta'})$ is transitive. Let $t_1, t_2, t_3 \in N$, $\Delta(t_1, t_2) = z_1$, $\Delta(t_2, t_3) = z_2$, and $\Delta(t_1, t_3) = z_3$. Relation structure $(N, \leq_{\Delta})$ is transitive, thus $z_1 + z_2 = z_3$. It is enough to show that $\Delta'(t_1, t_2) + \Delta'(t_2, t_3) = \Delta'(t_1, t_3)$. We build a table to show this result. In the first column, a cell corresponds to a sub-case, where we compare t_1, t_2, and t_3 each with t. In the second column, a cell is the proof of the sub-case, using Definition 7.7.1.

Sub-case			$\Delta'(t_1, t_2) + \Delta'(t_2, t_3) = \Delta'(t_1, t_3)$
$t_1 = t$	$t_2 = t$	$t_3 = t$	$z_1 + z_2 = z_3$
$t_1 \neq t$	$t_2 = t$	$t_3 = t$	$z_1 - 1 + z_2 = z_3 - 1$
$t_1 = t$	$t_2 \neq t$	$t_3 = t$	$z_1 + 1 + z_2 - 1 = z_3$
$t_1 = t$	$t_2 = t$	$t_3 \neq t$	$z_1 + z_2 + 1 = z_3 + 1$
$t_1 \neq t$	$t_2 \neq t$	$t_3 = t$	$z_1 + z_2 - 1 = z_3 - 1$
$t_1 \neq t$	$t_2 = t$	$t_3 \neq t$	$z_1 - 1 + z_2 + 1 = z_3$
$t_1 = t$	$t_2 \neq t$	$t_3 \neq t$	$z_1 + 1 + z_2 = z_3 + 1$
$t_1 \neq t$	$t_2 \neq t$	$t_3 \neq t$	$z_1 + z_2 = z_3$

Relation structure $(N, \leq_{\Delta'})$ is a total ordering, since it is reflexive, transitive, anti-symmetric, and compatible. □

Lemma 7.7.2. *If* $\Delta; t_1; n_1 \vdash P$, $t_2 \notin \operatorname{dom} P \cup \{t_1\}$, *and* $\Delta' = \operatorname{commit}_{\mathrm{D}}(\Delta, t_2)$, *then* $\Delta'; t_1; n_1 \vdash P$.

Proof. The proof follows by induction on the derivation of $\Delta; t_1; n_1 \vdash P$. We do a case analysis on the last rule applied.

- Case D-L-NIL:

$$\Delta; t_1; n_1 \vdash \emptyset$$

The case holds by direct application of rule D-L-NIL.

- Case D-L-CONS:

$$\frac{\text{(i) } \Delta; t_1; n_1 \vdash P_1 \qquad \text{(ii) } \Delta(t_1, t_3) = n_1 - n_3}{\Delta; t_1; n_1 \vdash P_1 \uplus \{t_3 \colon (n_3, a_3)\}}$$

where phaser P is $P_1 \uplus \{t_3 \colon (n_3, a_3)\}$. We know that (iii) $t_2 \notin \operatorname{dom} P_1 \cup \{t_1\}$, as $t_2 \notin \operatorname{dom} P_1 \cup \{t_1\}$ (hypothesis) and $P = P_1 \uplus \{t_3 \colon (n_3, a_3)\}$ (hypothesis). We apply the induction hypothesis to (i) $\Delta; t_1; n_1 \vdash P_1$, (iii) $t_2 \notin \operatorname{dom} P_1 \cup \{t_1\}$, and $\Delta' = \text{commit}_{\text{D}}(\Delta, t_2)$ (hypothesis), and get that (iv) $\Delta'; t_1; n_1 \vdash P_1$ holds. From Definition 7.7.1 since $t_1 \neq t_2$ (hypothesis) and $t_3 \neq t_2$ (hypothesis), then (v) $\Delta'(t_1, t_3) = \Delta(t_1, t_3) = n_1 - n_3$. Thus,

$$\frac{\text{(iv) } \Delta'; t_1; n_1 \vdash P_1 \qquad \text{(v) } \Delta'(t_1, t_3) = n_1 - n_3}{\Delta'; t_1; n_1 \vdash P_1 \uplus \{t_3 \colon (n_3, a_3)\}} \text{ D-L-CONS}$$

□

Lemma 7.7.3. *If* $\Delta \vdash P$, $t \notin \operatorname{dom} P$, *and* $\Delta' = \text{commit}_{\text{D}}(\Delta, t)$, *then* $\Delta' \vdash P$.

Proof. We do an induction proof on the derivation of $\Delta \vdash P$, with a case analysis on the last rule applied.

- Case D-PH-NIL:

$$\Delta \vdash \emptyset$$

where P is $\emptyset$. The case concludes with direct use of rule D-PH-NIL.

- Case D-PH-CONS:

$$\frac{\text{(i) } \Delta \vdash P_1 \qquad \text{(ii) } \Delta; t_1; n \vdash P_1}{\Delta \vdash P_1 \uplus \{t_1 \colon (n, a)\}}$$

where phaser P is $P_1 \uplus \{t_1 \colon (n, a)\}$. Since we have $t \notin \operatorname{dom} P$ (hypothesis) and P is $P_1 \uplus \{t_1 \colon (n, a)\}$ (hypothesis), then (iii) $t \notin \operatorname{dom} P_1$. We apply the induction hypothesis to (i) $\Delta \vdash P_1$, (iii) $t \notin \operatorname{dom} P_1$, and $\Delta' = \text{commit}_{\text{D}}(\Delta, t)$, and get that $\Delta' \vdash P_1$ holds. Since we have $\Delta; t_1; n \vdash P_1$, $t \notin \operatorname{dom} P_1 \cup \{t_1\}$, and $\Delta' = \text{commit}_{\text{D}}(\Delta, t_2)$, then by Lemma 7.7.2 we have that $\Delta'; t_1; n \vdash P_1$.

□

Lemma 7.7.4. *If* $\Delta; t; n \vdash P$, $t \notin \operatorname{dom} P$, *and* $\Delta' = \operatorname{commit}_{\mathrm{D}}(\Delta, t)$, *then* $\Delta'; t; (n+1) \vdash P$.

Proof. The proof is by induction on the derivation of relation $\Delta; t; n \vdash P$. We do a case analysis on the last rule applied.

- Case D-L-NIL:

$$\Delta; t; n \vdash \emptyset$$

  where P is $\emptyset$. The case holds by direct application of rule D-L-NIL.

- Case D-L-CONS:

$$\frac{\text{(i) } \Delta; t; n \vdash P_1 \qquad \text{(ii) } \Delta(t, t_1) = n - n_1}{\Delta; t; n \vdash P_1 \uplus \{t_1 \colon (n_1, a_1)\}}$$

  where phaser P is $P_1 \uplus \{t_1 \colon (n_1, a_1)\}$. Since $t \notin \operatorname{dom} P$ and P is $P_1 \uplus \{t_1 \colon (n_1, a_1)\}$, then (iii) $t \notin \operatorname{dom} P_1$. We apply the induction hypothesis to (i) $\Delta; t; n \vdash P_1$, (iii) $t \notin \operatorname{dom} P_1$, and $\Delta' = \operatorname{commit}_{\mathrm{D}}(\Delta, t)$ (hypothesis), and get that (iv) $\Delta'; t; (n+1) \vdash P_1$. From Definition 7.7.1 since $t_1 \neq t$, then $\Delta'(l, t_1) = \Delta(l, t_1) + 1 = n - n_1 + 1$. Hence,

$$\dfrac{\dfrac{\Delta'; t; (n+1) \vdash P_1 \qquad \dfrac{\Delta'(l, t_1) = n - n_1 + 1}{\Delta'(l, t_1) = (n+1) - n_1}}{\Delta'; t; (n+1) \vdash P_1 \uplus \{t_1 \colon (n_1, a_1)\}} \text{D-L-CONS}}{\Delta'; t; (n+1) \vdash P} \overset{\text{def}}{=}$$

□

Lemma 7.7.5. *If* $N \vdash \Delta$, $\Delta \vdash P \uplus \{t \colon (n, \mathbf{a})\}$ *and* $\Delta' = \operatorname{commit}_{\mathrm{D}}(\Delta, t)$, *then* $\Delta' \vdash P \uplus \{t \colon (n+1, \mathbf{u})\}$.

Proof. Let $P_1 = P \uplus \{t \colon (n, \mathbf{a})\}$. We have that (i) $P_1(t) = (n, \mathbf{a})$. Applying $N \vdash \Delta$ (hypothesis), $\Delta \vdash P_1$ (hypothesis), $P_1(t) = (n, \mathbf{a})$, then (ii) $\Delta \vdash P$, and (iii) $\Delta; t; n \vdash P$. By hypothesis we have that (iv) $t \notin \operatorname{dom} P$. From (ii) $\Delta \vdash P$, (iv) $t \notin \operatorname{dom} P$, $\Delta' = \operatorname{commit}_{\mathrm{D}}(\Delta, t)$ (hypothesis), and Lemma 7.7.3 yields (v) $\Delta' \vdash P$. Applying Lemma 7.7.4 to (iii) $\Delta; t; n \vdash P$, (iv) $t \notin \operatorname{dom} P$, and $\Delta' = \operatorname{commit}_{\mathrm{D}}(\Delta, t)$ (hypothesis), then (vi) $\Delta'; t; (n+1) \vdash P$. Hence,

$$\frac{\text{(v) } \Delta' \vdash P \qquad \text{(vi) } \Delta'; t; (n+1) \vdash P}{\Delta \vdash P \uplus \{t \colon (n+1, \mathbf{u})\}} \text{D-PH-CONS}$$

□

Lemma 7.7.6. *If* $N \vdash \Delta, \Delta; N \vdash M, \Delta' = \text{commit}_{\text{D}}(\Delta, t)$, *and* $M' = \text{commit}(M, t)$, *then* $\Delta'; N \vdash M'$.

Proof. The proof follows by induction on the structure of $M' = \text{commit}(M, t)$. We perform a case analysis.

- Case Com-n:

$$\text{commit}(\emptyset, t) = \emptyset$$

The case holds with rule T-p-map-nil.

- Case Com-s:

$$\frac{\text{(i) } \text{commit}(M_1, t) = M_2 \qquad \text{(ii) } t \notin \text{dom}\, P}{\text{commit}(M_1 \uplus \{p\colon P\}, t) = M_2 \uplus \{p\colon P\}}$$

where $M \stackrel{\text{def}}{=} M_1 \uplus \{p\colon P\}$ and $M' \stackrel{\text{def}}{=} M_2 \uplus \{p\colon P\}$.

We know that (iii) $M(p) = P$. From Lemma 6.3.6, $\Delta; N \vdash M$ (hypothesis), and (iii) $M(p) = P$, we have (iv) $\Delta \vdash P$, (v) $\text{dom}\, P \subseteq N$, and (vi) $\Delta; N \vdash M_1$. From (iv) $\Delta \vdash P$, (ii) $t \notin \text{dom}\, P$, $\Delta' = \text{commit}_{\text{D}}(\Delta, t)$, and Lemma 7.7.3, then (vii) $\Delta' \vdash P$. Next, we apply the induction hypothesis to $N \vdash \Delta$ (hypothesis), (vi) $\Delta; N \vdash M_1$, $\Delta' = \text{commit}_{\text{D}}(\Delta, t)$ (hypothesis), and (i) $\text{commit}(M_1, t) = M_2$, to obtain (viii) $\Delta'; N \vdash M_2$. Thus,

$$\frac{\text{(vii) } \Delta' \vdash P \qquad \text{(v) } \text{dom}\, P \subseteq N \qquad \text{(viii) } \Delta'; N \vdash M_2}{\Delta; N \vdash M_2 \uplus \{p\colon P\}}\ \text{T-p-map-cons}$$

- Case Com-c:

$$\frac{\text{commit}(M_1, t) = M_2}{\begin{array}{l}\text{commit}(M_1 \uplus \{p\colon P \uplus \{t\colon (n, \mathbf{a})\}\}, t) \\ = M_2 \uplus \{p\colon P \uplus \{t\colon (n+1, \mathbf{u})\}\}\end{array}}$$

where $M \stackrel{\text{def}}{=} M_1 \uplus \{p\colon P_1\}$, $M' \stackrel{\text{def}}{=} M_2 \uplus \{p\colon P_2\}$, $P_1 \stackrel{\text{def}}{=} P \uplus \{t\colon (n, \mathbf{a})\}$, and $P_2 \stackrel{\text{def}}{=} P \uplus \{t\colon (n+1, \mathbf{u})\}$.

We know that (iii) $M(p) = P_1$. From Lemma 6.3.6, $\Delta; N \vdash M$ (hypothesis), and (iii) $M(p) = P_1$, we have (iv) $\Delta \vdash P_1$, (v) $\text{dom}\, P_1 \subseteq N$, and (vi) $\Delta; N \vdash M_1$. From $N \vdash \Delta$ (hypothesis), (iv) $\Delta \vdash P \uplus \{t\colon (n, \mathbf{a})\}$, $\Delta' = \text{commit}_{\text{D}}(\Delta, t)$, and Lemma 7.7.5, then (vii) $\Delta' \vdash P_2$. Next, we apply the induction hypothesis to $N \vdash \Delta$ (hypothesis), (vi) $\Delta; N \vdash M_1$,

$\Delta' = \text{commit}_{\text{D}}(\Delta, t)$ (hypothesis), and (i) $\text{commit}(M_1, t) = M_2$, to obtain (viii) $\Delta'; N \vdash M_2$. Thus,

$$\frac{\text{(vii) } \Delta' \vdash P_2 \qquad \text{(v) } \operatorname{dom} P \subseteq N \qquad \text{(viii) } \Delta'; N \vdash M_2}{\Delta; N \vdash M_2 \uplus \{p \colon P_2\}} \text{T-P-MAP-CONS}$$

□

Lemma 7.7.7. *If* $\vdash_t M \colon \Gamma$ *and* $M' = \text{commit}(M, t)$*, then there exists a* Γ' *such that* $\operatorname{dom}\Gamma = \operatorname{dom}\Gamma'$, $\forall p \in \operatorname{dom}\Gamma' \colon \Gamma'(p) = \mathbf{u}$*, and* $\vdash_t M' \colon \Gamma'$.

Proof. The proof follows by induction on the structure of $M' = \text{commit}(M, t)$. We perform a case analysis on the definition of $\text{commit}(M, t)$.

- Case COM-N:

  $$\text{commit}(\emptyset, t) = \emptyset$$

  where M is $\emptyset$ and M' is $\emptyset$. By inverting $\vdash_t M \colon \Gamma$, we have that $\Gamma = \emptyset$. Hence, the case holds by hypothesis

  $$\vdash_t M' \colon \Gamma \quad \stackrel{\text{def}}{=} \quad \vdash_t \emptyset \colon \emptyset \quad \stackrel{\text{def}}{=} \quad \vdash_t M \colon \Gamma$$

  and $\operatorname{dom}\Gamma = \operatorname{dom}\Gamma' = \emptyset$ and $\forall p \in \operatorname{dom}\Gamma' \colon \Gamma'(p) = \mathbf{u}$.

- Case COM-C:

  $$\frac{\text{(i) } \text{commit}(M_1, t) = M_2}{\text{commit}(M_1 \uplus \{p \colon P_1\}, t) = M_2 \uplus \{p \colon P_2\}}$$

  where $P_1 \stackrel{\text{def}}{=} P \uplus \{t \colon (n, \mathbf{a})\}$, $P_2 \stackrel{\text{def}}{=} P \uplus \{t \colon (n+1, \mathbf{u})\}$, $M \stackrel{\text{def}}{=} M_1 \uplus \{p \colon P_1\}$, and $M' \stackrel{\text{def}}{=} M_2 \uplus \{p \colon P_2\}$.

  Since we have $\vdash_t M \colon \Gamma$ (hypothesis) and $t \in \operatorname{dom} M(p)$ (hypothesis), then by Lemma 6.3.5 there exists a Γ_1 such that (ii) $\Gamma = \Gamma_1 \uplus \{p \colon \mathbf{a}\}$, and (iii) $\vdash_t M_1 \colon \Gamma_1$. Applying the induction hypothesis to (iii) $\vdash_t M_1 \colon \Gamma_1$ and (i) $\text{commit}(M_1, t) = M_2$, yields that there exists a typing Γ_2 such that (iv) $\operatorname{dom}\Gamma_1 = \operatorname{dom}\Gamma_2$, (v) $\forall p \in \operatorname{dom}\Gamma_2 \colon \Gamma_2(p) = \mathbf{u}$, and (vi) $\vdash_t M_2 \colon \Gamma_2$. Let $\Gamma' \stackrel{\text{def}}{=} \Gamma_2 \uplus \{p \colon \mathbf{u}\}$. From (iv) $\operatorname{dom}\Gamma_1 = \operatorname{dom}\Gamma_2$ and $\operatorname{dom}\Gamma = \operatorname{dom}\Gamma_1 \cup \{p\}$, then (1) $\operatorname{dom}\Gamma = \operatorname{dom}\Gamma'$. Since we have (v) $\forall p \in \operatorname{dom}\Gamma_2 \colon \Gamma_2(p) = \mathbf{u}$, $\operatorname{dom}\Gamma' = \operatorname{dom}\Gamma_2 \cup \{p\}$, and $\Gamma'(p) = \mathbf{u}$, then (2) $\forall p \in \operatorname{dom}\Gamma' \colon \Gamma'(p) = \mathbf{u}$. Finally, the following derivation tree holds.

  $$\cfrac{\cfrac{\text{(vi) } \vdash_t M_2 \colon \Gamma_2 \qquad P_2(t) = (\_, \mathbf{u})}{\vdash_t M_2 \uplus \{p \colon P_2\} \colon \Gamma_2 \uplus \{p \colon \mathbf{u}\}} \text{T-PERM-CONS}}{\vdash_t M' \colon \Gamma'} \stackrel{\text{def}}{=}$$

- Case COM-S:

$$\frac{\text{(i) } \mathrm{commit}(M_1, t) = M_2 \qquad \text{(ii) } t \notin \mathrm{dom}\, P}{\mathrm{commit}(M_1 \uplus \{p\colon P\}, t) = M_2 \uplus \{p\colon P\}}$$

where $M \stackrel{\text{def}}{=} M_1 \uplus \{p\colon P\}$ and $M' \stackrel{\text{def}}{=} M_2 \uplus \{p\colon P\}$. Since we have $\vdash_t M\colon \Gamma$ (hypothesis) and $t \notin \mathrm{dom}\, M(p)$ (hypothesis), then by Lemma 6.3.4 we have (iii) $\vdash_t M_1\colon \Gamma$. Applying the induction hypothesis to (iii) $\vdash_t M_1\colon \Gamma$ and (i) $\mathrm{commit}(M_1, t) = M_2$, yields that there exists a Γ' such that (1) $\mathrm{dom}\,\Gamma = \mathrm{dom}\,\Gamma'$, (2) $\forall p \in \mathrm{dom}\,\Gamma'\colon \Gamma'(p) = \mathbf{u}$, and (iv) $\vdash_t M_2\colon \Gamma'$. Thus,

$$\dfrac{\dfrac{\text{(iv) } \vdash_t M_2\colon \Gamma' \qquad \text{(ii) } t \notin \mathrm{dom}\, P}{\vdash_t M_2 \uplus \{p\colon P\}\colon \Gamma'}\ \textsc{T-perm-skip}}{\vdash_t M'\colon \Gamma'}$$

□

Lemma 7.7.8. *If* $\Psi; \Gamma \vdash (B, \mathtt{next}; b)$*, then there exists a typing* Γ' *such that* $\mathrm{dom}\,\Gamma = \mathrm{dom}\,\Gamma'$*,* $\forall p \in \mathrm{dom}\,\Gamma'\colon \Gamma'(p) = \mathbf{u}$*, and* $\Psi; \Gamma' \vdash (B, b)$*.*

Proof. By inverting the hypothesis we get the following premises.

$$\dfrac{\text{(i) } \Gamma \vdash B \qquad \dfrac{\cdots \qquad \text{(ii) } \{p_1\colon \mathbf{u}, \ldots, p_n\colon \mathbf{u}\} \vdash b\colon \emptyset}{\{p_1\colon \mathbf{a}, \ldots, p_n\colon \mathbf{a}\} \vdash \mathtt{next}; b\colon \emptyset}}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, \mathtt{next}; b)}$$

where $\Gamma \stackrel{\text{def}}{=} \{p_1\colon \mathbf{a}, \ldots, p_n\colon \mathbf{a}\}$. Let $\Gamma' \stackrel{\text{def}}{=} \{p_1\colon \mathbf{u}, \ldots, p_n\colon \mathbf{u}\}$. By definition we have that $\mathrm{dom}\,\Gamma = \mathrm{dom}\,\Gamma'$ and that $\forall p \in \mathrm{dom}\,\Gamma'\colon \Gamma'(p) = \mathbf{u}$. Hence,

$$\frac{\text{(i) } \Gamma \vdash B \qquad \text{(ii) } \Gamma' \vdash b\colon \emptyset}{\langle \emptyset; \emptyset \rangle; \Gamma' \vdash (B, b)}\ \textsc{T-t-r}$$

□

Lemma 7.7.9. *If* $\vdash_{t'} M\colon \Gamma$*,* $t \neq t'$*, and* $M' = \mathrm{commit}(M, t)$*, then* $\vdash_{t'} M'\colon \Gamma$*.*

Proof. The proof follows by induction on the definition of $M' = \mathrm{commit}(M, t)$. We perform a case analysis on the derivation of the last rule applied.

- Case COM-N:

$$\mathrm{commit}(\emptyset, t) = \emptyset$$

where $M \stackrel{\text{def}}{=} M' \stackrel{\text{def}}{=} \emptyset$. The case holds by hypothesis.

- Case Com-s:

$$\frac{\text{(i) } \mathsf{commit}(M_1, t) = M_2 \qquad \text{(ii) } t \notin \mathsf{dom}\, P}{\mathsf{commit}(M_1 \uplus \{p\colon P\}, t) = M_2 \uplus \{p\colon P\}}$$

where $M \stackrel{\text{def}}{=} M_1 \uplus \{p\colon P\}$ and $M' \stackrel{\text{def}}{=} M_2 \uplus \{p\colon P\}$. We now test if $t' \in \mathsf{dom}\, P$.

  - Case $t' \in \mathsf{dom}\, P$. From Lemma 6.3.5, $\vdash_{t'} M\colon \Gamma$ (hypothesis), $t' \in \mathsf{dom}\, M_1(p)$, then (i) $P(t') = (n, a)$, (ii) $\Gamma = \Gamma' \uplus \{p\colon a\}$, and (iii) $\vdash_{t'} M_1\colon \Gamma'$. Applying the induction hypothesis to (iii) $\vdash_{t'} M_1\colon \Gamma_1, t \neq t'$ (hypothesis), and $M_2 = \mathsf{commit}(M_1, t)$ (hypothesis), then (iv) $\vdash_{t'} M_2\colon \Gamma'$. Thus,

$$\frac{\text{(iv) } \vdash_{t'} M_2\colon \Gamma' \qquad \text{(i) } P(t') = (n, a)}{\vdash_t M \uplus \{p\colon P\}\colon \Gamma \uplus \{p\colon a\}}\ \textsc{T-perm-cons}$$

  - Case $t' \notin \mathsf{dom}\, P$. Applying Lemma 6.3.4 to $\vdash_{t'} M\colon \Gamma$ (hypothesis) and $t' \notin \mathsf{dom}\, M(p)$, then (i) $\vdash_{t'} M_1\colon \Gamma$. Next, we apply the induction hypothesis to (i) $\vdash_{t'} M_1\colon \Gamma$, $t \neq t'$ (hypothesis), and $M_2 = \mathsf{commit}(M_1, t)$ (hypothesis), then (ii) $\vdash_{t'} M_2\colon \Gamma$. Thus,

$$\frac{\dfrac{\text{(ii) } \vdash_{t'} M_2\colon \Gamma \qquad t \notin \mathsf{dom}\, P}{\vdash_t M_2 \uplus \{p\colon P\}\colon \Gamma}\ \textsc{T-perm-skip} \quad \stackrel{\text{def}}{=}}{\vdash_t M'\colon \Gamma}$$

- Case Com-c:

$$\frac{\mathsf{commit}(M_1, t) = M_2}{\mathsf{commit}(M_1 \uplus \{p\colon P_1\}, t) = M_2 \uplus \{p\colon P_2\}}$$

where $P_1 \stackrel{\text{def}}{=} P \uplus \{t\colon (n, \mathbf{a})\}$, $P_2 \stackrel{\text{def}}{=} P \uplus \{t\colon (n+1, \mathbf{u})\}$, $M \stackrel{\text{def}}{=} M_1 \uplus \{p\colon P_1\}$, and $M' \stackrel{\text{def}}{=} M_2 \uplus \{p\colon P_2\}$. The proof has the same structure as in case Com-s.

□

Lemma 7.7.10. *If* $\Sigma; M \vdash T$, $t \notin \mathsf{dom}\, T$, *and* $M' = \mathsf{commit}(M, t)$, *then* $\Sigma; M' \vdash T$.

Proof. The proof follows by induction on the typing relation. We perform a case analysis on the derivation of the last rule applied.

- Case T-TM-N:

$$\emptyset; M \vdash \emptyset$$

where $T \stackrel{\text{def}}{=} \emptyset$. The case holds by direct application of T-TM-N.

- Case T-TM-C:

$$\frac{\text{(i) } \vdash_{t'} M : \Gamma \qquad \text{(ii) } \Psi; \Gamma \vdash \tau \qquad \text{(iii) } \Sigma'; M \vdash T'}{\Sigma' \uplus \{t' : \Psi\}; M \vdash T' \uplus \{t' : \tau\}}$$

where $\Sigma \stackrel{\text{def}}{=} \Sigma' \uplus \{t' : \Psi\}$ and $T \stackrel{\text{def}}{=} T' \uplus \{t' : \tau\}$. From $t \notin \operatorname{dom} T$ we have that (iv) $t \neq t'$. Applying Lemma 7.7.9 to (i) $\vdash_{t'} M : \Gamma$, (iv) $t \neq t'$, and $M' = \text{commit}(M, t)$ results in (v) $\vdash_{t'} M' : \Gamma$. We also know that (vi) $t \notin \operatorname{dom} T'$, as $t \notin \operatorname{dom} T$ (hypothesis) and $T \stackrel{\text{def}}{=} T' \uplus \{t' : \tau\}$ (hypothesis). Next, from the induction hypothesis, (iii) $\Sigma'; M \vdash T'$, (vi) $t \notin \operatorname{dom} T'$, and $M' = \text{commit}(M, t)$, and we have that (vii) $\Sigma'; M' \vdash T'$. Hence,

$$\frac{\text{(v) } \vdash_{t'} M' : \Gamma \qquad \text{(ii) } \Psi; \Gamma \vdash \tau \qquad \text{(vii) } \Sigma; M' \vdash T'}{\Sigma \uplus \{t' : \Psi\}; M' \vdash T' \uplus \{t' : \tau\}}\ \text{T-TM-C}$$

□

Lemma 7.7.11. *If* $\Sigma; M \vdash T \uplus \{t : (B, \texttt{next}; b)\}$ *and* $M' = \text{commit}(M, t)$*, then there exists a* Σ' *such that* $\Sigma'; M' \vdash T \uplus \{t : (B, b)\}$*.*

Proof. Let $\tau_1 = (B, \texttt{next}; b)$ and $T_1 \stackrel{\text{def}}{=} T \uplus \{t : \tau_1\}$. We know that (i) $T_1(t) = \tau_1$. We apply Lemma 6.3.2 to $\Sigma; M \vdash T_1$ (hypothesis) and (i) $T_1(t) = \tau_1$, and get (ii) $\Sigma = \Sigma_1 \uplus \{t : \Psi\}$, (iii) $\vdash_t M : \Gamma$, (iv) $\Psi; \Gamma \vdash \tau_1$, and (v) $\Sigma_1; M \vdash T$.

From (iii) $\vdash_t M : \Gamma$, $M' = \text{commit}(M, t)$ (hypothesis), and Lemma 7.7.7 we have that there exists a typing Γ' such that (vi) $\operatorname{dom}\Gamma = \operatorname{dom}\Gamma'$, (vii) $\forall p \in \operatorname{dom}\Gamma' : \Gamma'(p) = \mathbf{u}$, and (viii) $\vdash_t M' : \Gamma'$.

But we also know that from (iv) $\Psi; \Gamma \vdash \tau_1$ and Lemma 7.7.8, we have a typing Γ'' such that (ix) $\operatorname{dom}\Gamma = \operatorname{dom}\Gamma''$, (x) $\forall p \in \operatorname{dom}\Gamma'' : \Gamma''(p) = \mathbf{u}$, (xi) $\Psi; \Gamma'' \vdash (B, b)$. We know that $\operatorname{dom}\Gamma = \operatorname{dom}\Gamma'' = \operatorname{dom}\Gamma'$ and that $\forall p \in \operatorname{dom}\Gamma : \Gamma''(p) = \Gamma'(p) = \mathbf{u}$, hence (xii) $\Gamma'' = \Gamma'$.

Finally, since we have (v) $\Sigma_1; M \vdash T$, $t \notin \operatorname{dom} T$, and $M' = \text{commit}(M, t)$ (hypothesis), then from Lemma 7.7.10 (vii) $\Sigma_1; M' \vdash T$ holds.

Let $\tau_2 \stackrel{\text{def}}{=} (B, b)$. Hence,

$$\frac{\text{(v) } \vdash_t M' : \Gamma \qquad \text{(vi) } \Psi; \Gamma \vdash \tau_2 \qquad \text{(vii) } \Sigma_1; M' \vdash T}{\Sigma \uplus \{t : \Psi\}; M \vdash T \uplus \{t : \tau_2\}}\ \text{T-TM-C}$$

□

7.8 Finish

Lemma 7.8.1. *For any t if $\emptyset \vdash b : \emptyset$, then $\langle \emptyset; \emptyset \rangle \vdash \big(\emptyset, \{t : (\emptyset, b)\}\big)$.*

Proof. The following premise (i) holds.

$$\dfrac{\dfrac{}{\vdash_t \emptyset : \emptyset}\,\textsc{T-perm-nil} \quad \dfrac{\dfrac{}{\emptyset \vdash \emptyset}\,\textsc{T-b-n} \qquad \emptyset \vdash b : \emptyset}{\langle \emptyset; \emptyset \rangle; \emptyset \vdash (B, b)}\,\textsc{T-t-r} \quad \dfrac{}{\emptyset; \emptyset \vdash \emptyset}\,\textsc{T-tm-n}}{\{t : \langle \emptyset; \emptyset \rangle\}; \emptyset \vdash \{t : (B, b)\}}\,\textsc{T-tm-c}$$

Let $\Sigma \stackrel{\text{def}}{=} \{t : \langle \emptyset; \emptyset \rangle\}$, $T \stackrel{\text{def}}{=} \{t : (B, b)\}$, $N \stackrel{\text{def}}{=} \{t\}$, and Δ be such that $\Delta(t, t) = 0$ and $\operatorname{dom} \Delta = \{(t, t)\}$. It is easy to see that $(N, \leq_\Delta)$ is a total ordering and that

$$\forall t_1, t_2 \in N : \Delta(t_1, t_2) = z \implies \Delta(t_2, t_1) = -z$$

Thus, from rule D-WF we have that (ii) $N \vdash \Delta$. Hence,

$$\dfrac{\text{(ii) } N \vdash \Delta \qquad \dfrac{}{\Delta; N \vdash \emptyset}\,\textsc{T-p-map-nil} \qquad \text{(i) } \Sigma; \emptyset \vdash T}{\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)}$$

□

Lemma 7.8.2. *If $\Psi; \Gamma \vdash (B, \texttt{finish}(b_2); b_1)$ and $S = \big(\emptyset, \{t_2 : (\emptyset, b_2)\}\big)$, then there exists a Ψ' such that $\Psi'; \Gamma \vdash S \triangleright (B, b_1)$.*

Proof. Inverting (ii) yields the following premises.

$$\dfrac{\text{(i) } \Gamma \vdash B \qquad \dfrac{\dfrac{\text{(ii) } \emptyset \vdash b_2 : \emptyset}{\Gamma \vdash \texttt{finish}(b_2) : \Gamma} \qquad \text{(iii) } \Gamma \vdash b_1 : \emptyset}{\Gamma \vdash \texttt{finish}(b_2); b_1 : \emptyset}}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, \texttt{finish}(b_2); b_1)}$$

where $\Psi = \langle \emptyset; \emptyset \rangle$. With Lemma 7.8.1 and $\emptyset \vdash b_2 : \emptyset$, we have that there exists a Ψ' such that (v) $\Psi' \vdash S$. Therefore,

$$\dfrac{\text{(v) } \Psi' \vdash S \qquad \dfrac{\text{(i) } \Gamma \vdash B \qquad \text{(iii) } \Gamma \vdash b_1 : \emptyset}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, b_1)}\,\textsc{T-t-r}}{\Psi'; \Gamma \vdash S \triangleright (B, b_1)}\,\textsc{T-t-f}$$

□

Lemma 7.8.3. *If* $\Sigma; M \vdash T \uplus \{t_1{:}(B, \texttt{finish}(b_2); b_1)\}$ *and* $S = \big(\emptyset, \{t_2 : (\emptyset, b_2)\}\big)$, *then there exists a* Σ' *such that* $\Sigma'; M \vdash T \uplus \{t_1 \colon S \triangleright (B, b_1)\}$.

Proof. Let $\tau_1 \stackrel{\text{def}}{=} (B, \texttt{finish}(b_2); b_1)$ and $T_1 \stackrel{\text{def}}{=} T \uplus \{t_1 \colon (B, \texttt{finish}(b_2); b_1)\}$. From Lemma 6.3.2, $\Sigma; M \vdash T_1$, and $T_1(t_1) = \tau_1$, we get that (i) $\Sigma = \Sigma_1 \uplus \{t_1 \colon \Psi\}$, (ii) $\vdash_{t_1} M \colon \Gamma$, (iii) $\Psi; \Gamma \vdash \tau_1$, and (iv) $\Sigma_1; M \vdash T$. Let $\tau_2 \stackrel{\text{def}}{=} S \triangleright (B, b_1)$. We have that from Lemma 7.8.2, $\Psi; \Gamma \vdash \tau_1$, and $S = \big(\emptyset, \{t_2 \colon (\emptyset, b_2)\}\big)$ (hypothesis), we get that there exists a Ψ' such that (v) $\Psi'; \Gamma \vdash \tau_2$. Thus,

$$\frac{\text{(ii)} \vdash_{t_1} M \colon \Gamma \qquad \text{(v)}\ \Psi'; \Gamma \vdash \tau_2 \qquad \text{(iv)}\ \Sigma_1; M \vdash T}{\Sigma_1 \uplus \{t \colon \Psi'\}; M \vdash T \uplus \{t \colon \tau\}}\ \text{T-TM-C}$$

□

7.9 Join

Lemma 7.9.1. *If* $\Sigma; M \vdash T \uplus \{t \colon S \triangleright (B, b)\}$, *then there exists a* Σ' *such that* $\Sigma'; M \vdash T \uplus \{t \colon (B, b)\}$.

Proof. Let $\tau_1 \stackrel{\text{def}}{=} S \triangleright (B, b)$ and $T_1 \stackrel{\text{def}}{=} T \uplus \{t \colon \tau_1\}$. Applying Lemma 6.3.2 to the hypothesis and to $T_1(t) = \tau_1$, we get that (i) $\Sigma = \Sigma_1 \uplus \{t \colon \Psi\}$, (ii) $\vdash_t M \colon \Gamma$, (iii) $\Psi; \Gamma \vdash \tau_1$, and (iv) $\Sigma_1; M \vdash T$. By inverting (iii) we get that (v) $\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, b)$. We conclude the proof applying rule T-TM-C to (ii), (v), and (iv). □

7.10 Control flow

Lemma 7.10.1. *If* $\Gamma \vdash b \colon \Gamma''$ *and* $\Gamma'' \vdash b' \colon \Gamma'$, *then* $\Gamma \vdash b \cdot b' \colon \Gamma'$.

Proof. We know that the program concatenation is total, let $b_0 = b \cdot b'$. The proof follows by induction on the definition of $b \cdot b'$. Next, we perform a case analysis on the derivation of the last rule applied.

- Case $(i; b'') \cdot b' \stackrel{\text{def}}{=} i; (b'' \cdot b')$, where $b = i; b''$. We invert the hypothesis to obtain the next premises.

$$\frac{\text{(i)}\ \Gamma \vdash i \colon \Gamma_i \qquad \text{(ii)}\ \Gamma_i \vdash b'' \colon \Gamma''}{\Gamma \vdash i; b'' \colon \Gamma''}$$

Applying the induction hypothesis to (ii) $\Gamma_i \vdash b\colon \Gamma''$ and $\Gamma'' \vdash b'\colon \Gamma'$ (hypothesis) yields that (iii) $\Gamma_i \vdash b'' \cdot b'\colon \Gamma'$. Thus,

$$\dfrac{\dfrac{\text{(i) } \Gamma \vdash i\colon \Gamma_i \qquad \text{(iii) } \Gamma_i \vdash b'' \cdot b'\colon \Gamma'}{\Gamma \vdash i; (b'' \cdot b')\colon \Gamma'}}{\Gamma \vdash b \cdot b'\colon \Gamma'}\ \stackrel{\text{def}}{=}$$

- Case $\texttt{end} \cdot b' \stackrel{\text{def}}{=} b'$, where $b = \texttt{end}$. By inversion of $\Gamma \vdash b\colon \Gamma''$, we get that $\Gamma'' = \Gamma$, hence we have $\Gamma \vdash b'\colon \Gamma'$. Thus, from $\texttt{end} \cdot b' \stackrel{\text{def}}{=} b'$ and the latter, we get that $\Gamma \vdash \texttt{end} \cdot b'\colon \Gamma'$

□

Lemma 7.10.2. *If* $c; b \to b'$ *and* $\Gamma \vdash c; b\colon \emptyset$*, then* $\Gamma \vdash b'\colon \emptyset$*.*

Proof. We invert hypothesis $c; b \to b'$ and obtain three cases.

- Case R-SKIP:

$$\texttt{skip}; b \to b$$

  where $c; b \stackrel{\text{def}}{=} \texttt{skip}; b$ and $b' \stackrel{\text{def}}{=} b$. We invert hypothesis $\Gamma \vdash \texttt{skip}; b\colon \emptyset$ to conclude our case.

$$\dfrac{\cdots \qquad \Gamma \vdash b\colon \emptyset}{\Gamma \vdash \texttt{skip}; b\colon \emptyset}$$

- Case R-ITER:

$$\texttt{loop}(b''); b \to b'' \cdot (\texttt{loop}(b''); b)$$

  where $c; b \stackrel{\text{def}}{=} \texttt{loop}(b''); b$ and $b' \stackrel{\text{def}}{=} b'' \cdot (\texttt{loop}(b''); b)$. We invert hypothesis $\Gamma \vdash \texttt{loop}(b''); b\colon \emptyset$ and premise (i).

$$\dfrac{\dfrac{\text{(i) } \Gamma \vdash b''\colon \Gamma}{\Gamma \vdash \texttt{loop}(b'')\colon \Gamma} \qquad \cdots}{\Gamma \vdash \texttt{loop}(b''); b'\colon \emptyset}$$

  From Lemma 7.10.1, $\Gamma \vdash b''\colon \Gamma$ and $\Gamma \vdash \texttt{loop}(b''); b'\colon \emptyset$ (hypothesis), we get that $\Gamma \vdash b'' \cdot (\texttt{loop}(b''); b)\colon \emptyset$.

- Case R-ELIDE:

$$\texttt{loop}(b''); b' \to b'$$

  where $c; b \stackrel{\text{def}}{=} \texttt{loop}(b''); b'$. We invert hypothesis $\Gamma \vdash \texttt{loop}(b''); b'\colon \emptyset$ to conclude our case.

$$\dfrac{\cdots \qquad \Gamma \vdash b'\colon \emptyset}{\Gamma \vdash \texttt{loop}(b''); b'\colon \emptyset}$$

□

Lemma 7.10.3. *If* $c; b \to b'$ *and* $\Psi; \Gamma \vdash (B, c; b)$*, then* $\Psi; \Gamma \vdash (B, b')$*.*

Proof. By inverting the hypothesis we get the next premises.

$$\frac{\text{(i) } \Gamma \vdash B \qquad \text{(ii) } \Gamma \vdash c; b \colon \emptyset}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, c; b)}$$

From Lemma 7.10.2, $c; b \to b'$ (hypothesis), and $\Gamma \vdash c; b \colon \emptyset$, we know that $\Gamma \vdash b' \colon \emptyset$. Hence,

$$\frac{\text{(i) } \Gamma \vdash B \qquad \text{(ii) } \Gamma \vdash b' \colon \emptyset}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, b')} \text{T-T-R}$$

□

Lemma 7.10.4. *If* $c; b \to b'$ *and* $\Sigma; M \vdash T \uplus \{t \colon (B, c; b)\}$ *then* $\Sigma; M \vdash T \uplus \{t \colon (B, b')\}$*.*

Proof. From Lemma 6.3.2 and the hypothesis we have that (i) $\Sigma = \Sigma'' \uplus \{t \colon \Psi\}$, (ii) $\vdash_t M \colon \Gamma$, (iii) $\Psi; \Gamma \vdash \tau$, and (iv) $\Sigma''; M \vdash T$. Next, we apply Lemma 7.10.3 to $c; b \to b'$ (hypothesis) and (iii) $\Psi; \Gamma \vdash \tau$ to obtain (v) $\Psi; \Gamma \vdash (B, b')$. Therefore,

$$\frac{\text{(ii) } \vdash_t M \colon \Gamma \qquad \text{(v) } \Psi; \Gamma \vdash (B \uplus \{p \colon n\}, b) \qquad \text{(iv) } \Sigma''; M \vdash T}{\Sigma'' \uplus \{t \colon \Psi\}; M \vdash T \uplus \{t \colon (B \uplus \{p \colon n\}, b)\}}$$

□

7.11 Main result

Theorem 7.11.1 (Subject reduction)**.** *If* $\Psi_1 \vdash S_1$ *and* $S_1 \to S_2$*, then there exists a* Ψ_2 *such that* $\Psi_2 \vdash S_2$*.*

Proof. By induction on the derivation of the reduction relation between abstract machines ($\to$), analysing the last rule applied.

Case R-ASYNC.

$$\begin{aligned} &\big(M_1, T \uplus \{t \colon (B, \texttt{async}(s, b'); b)\}\big) \\ \to &\big(\text{copy}(s, t, t', M_1), T \uplus \{t \colon (B, b)\} \uplus \{t' \colon (\text{bounds}(s), b')\}\big) \end{aligned}$$

where S_1 is $\big(M, T \uplus \{t \colon (B, \texttt{async}(s, b'); b)\}\big)$ and S_2 is

$$\big(\text{copy}(s, t, t', M), T \uplus \{t \colon (B, b)\} \uplus \{t' \colon (\text{bounds}(s), b')\}\big)$$

Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, \texttt{async}(s, b'); b)\}$. By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\text{(i) } \operatorname{dom} T_1 \vdash \Delta \qquad \text{(ii) } \Delta; \operatorname{dom} T_1 \vdash M_1 \qquad \text{(iii) } \Sigma; M_1 \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M_1, T_1\big)}$$

Let $M_2 \stackrel{\text{def}}{=} M \uplus \big\{p\colon P \uplus \{t\colon \langle n; \mathbf{a} \rangle\}\big\}$, $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, b)\}$, and $N \stackrel{\text{def}}{=} \operatorname{dom} T_1$. To show that $\langle \Delta; \Sigma \rangle \vdash \big(M_2, T_2\big)$ holds, we need to establish the following. Since we have (i) $N \vdash \Delta$, (ii) $\Delta; N \vdash M_1$, $\operatorname{copy}(s, t, t', M_1) = M_2$ (hypothesis), and $t' \notin N$ (hypothesis), then from Lemma 7.1.7 there exists a Δ' such that (1) $\operatorname{dom} T_2 \vdash \Delta'$ and (2) $\Delta'; \operatorname{dom} T_2 \vdash M_2$.

We apply Lemma 7.1.17 to

$$\Sigma; M_1 \vdash T \uplus \{t\colon (B, \texttt{async}(s, b'); b)\}$$

$\operatorname{copy}(s, t, t', M_1) = M_2$ (hypothesis), $\operatorname{bounds}(s) = B'$ (hypothesis), and $t' \notin \operatorname{dom} T \cup \{t\}$ (hypothesis), to obtain a Σ' such that (3) $\Sigma'; M_2 \vdash T_2$. We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-PHASER.

$$\frac{\text{(i) } q \notin \operatorname{bn}(b) \qquad M_2 \stackrel{\text{def}}{=} M_1 \uplus \big\{q\colon \{t\colon \langle 0; \mathbf{u} \rangle\}\big\} \qquad B_2 \stackrel{\text{def}}{=} B_1 \uplus \{q\colon 0\}}{\big(M_1, T \uplus \{t\colon (B_1, p = \texttt{newPhaser}(); b)\}\big) \to \big(M_2, T \uplus \{t\colon (B_2, b[q/p])\}\big)}$$

Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon (B_1, p = \texttt{newPhaser}(); b)\}$ and $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon (B_2, b[q/p])\}$. By inversion of the hypothesis that the state is well typed we get the following premises.

$$\frac{\operatorname{dom} T_1 \vdash \Delta \qquad \Delta; \operatorname{dom} T_1 \vdash M_1 \qquad \Sigma; M_1 \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M_1, T_1\big)}$$

To show that $\langle \Delta; \Sigma \rangle \vdash \big(M_2, T_2\big)$ holds, we need to establish the following.

1. We have that $\operatorname{dom} T_1 = \operatorname{dom} T_2$, hence $\operatorname{dom} T_2 \vdash \Delta$.
2. From Lemma 7.2.9, $\operatorname{dom} T_1 \vdash \Delta$, $\Delta; \operatorname{dom} T_1 \vdash M_1$, $q \notin \operatorname{dom} M_1$, and $t \in \operatorname{dom} T_1$, we get that $\Delta; \operatorname{dom} T_2 \vdash M_2$.
3. Applying Lemma 7.2.13 to $\Sigma; M \vdash T \uplus \{t\colon (B, p = \texttt{newPhaser}(); b)\}$, $q \notin \operatorname{dom} M_1$ (since we have M_2), and (i) $q \notin \operatorname{bn}(b)$, to obtain $\Sigma; M_2 \vdash T_2$.

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-DEREG.

$$\begin{aligned}&\big(M \uplus \big\{p\colon P \uplus \{t\colon v\}\big\}, T \uplus \big\{t\colon (B \uplus \{p\colon n\}, \mathtt{dereg}(p); b)\big\}\big)\\ \to&\big(M \uplus \big\{p\colon P\big\}, T \uplus \big\{t\colon (B, b)\big\}\big)\end{aligned}$$

where S_1 is $\big(M \uplus \big\{p\colon P \uplus \{t\colon v\}\big\}, T \uplus \big\{t\colon (B \uplus \{p\colon n\}, \mathtt{dereg}(p); b)\big\}\big)$ and S_2 is $(M \uplus \big\{p\colon P\big\}, T \uplus \big\{t\colon (B, b)\big\})$. Let $M_1 \stackrel{\text{def}}{=} M \uplus \big\{p\colon P \uplus \{t\colon v\}\big\}$ and $T_1 \stackrel{\text{def}}{=} T \uplus \big\{t\colon (B \uplus \{p\colon n\}, \mathtt{dereg}(p); b)\big\}$. By inversion of the hypothesis we get the following premises.

$$\frac{\operatorname{dom} T_1 \vdash \Delta \qquad \Delta; \operatorname{dom} T_1 \vdash M_1 \qquad \Sigma; M_1 \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M_1, T_1\big)}$$

Let $M_2 \stackrel{\text{def}}{=} M \uplus \big\{p\colon P\big\}$ and $T_2 \stackrel{\text{def}}{=} T \uplus \big\{t\colon (B, b)\big\}$. To show that $\langle \Delta; \Sigma \rangle \vdash \big(M_2, T_2\big)$ holds, we need to establish the following.

1. We have that $\operatorname{dom} T_1 = \operatorname{dom} T_2$, hence $\operatorname{dom} T_2 \vdash \Delta$.
2. From Lemma 7.3.1, $\operatorname{dom} T_1 \vdash \Delta$, $\Delta; N \vdash M_1$, $M_1(p) = P_1$, and $t \in \operatorname{dom} P_1$, then $M_1 = M \uplus \{p\colon P\}$, $P_1 = P \uplus \{t\colon v\}$, and $\Delta; \operatorname{dom} T_2 \vdash M \uplus \big\{p\colon P\big\}$.
3. Applying Lemma 7.3.3 to $\Sigma; M \vdash T \uplus \{t\colon (B, p = \mathtt{newPhaser}(); b)\}$, $q \notin \operatorname{dom} M_1$ (since we have M_2), and (i) $q \notin \operatorname{bn}(b)$, to obtain $\Sigma; M_2 \vdash T_2$.

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-ADVANCE.

$$\begin{aligned}&\big(M \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{u})\}\big\}, T \uplus \{t\colon (B, \mathtt{adv}(p); b)\}\big)\\ \to&\big(M \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{a})\}\big\}, T \uplus \{t\colon (B, b)\}\big)\end{aligned}$$

where S_1 is $\big(M \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{u})\}\big\}, T \uplus \{t\colon (B, \mathtt{adv}(p); b)\}\big)$ and S_2 is

$$\big(M \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{a})\}\big\}, T \uplus \{t\colon (B, b)\}\big)$$

Let $M_1 \stackrel{\text{def}}{=} M \uplus \big\{p\colon P \uplus \{t\colon \langle n; \mathbf{u} \rangle\}\big\}$ and $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, \mathtt{adv}(p); b)\}$.

By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\operatorname{dom} T_1 \vdash \Delta \qquad \Delta; \operatorname{dom} T_1 \vdash M_1 \qquad \Sigma; M_1 \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M_1, T_1\big)}$$

Let $M_2 \stackrel{\text{def}}{=} M \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{a})\}\big\}$ and $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, b)\}$. To show that $\langle \Delta; \Sigma \rangle \vdash \big(M_2, T_2\big)$ holds, we need to establish the following.

1. We have that $\operatorname{dom} T_1 = \operatorname{dom} T_2$, hence $\operatorname{dom} T_2 \vdash \Delta$.
2. From Lemma 7.4.2, $\operatorname{dom} T_1 \vdash \Delta$, and $\Delta; N \vdash M_1$, then $\Delta; \operatorname{dom} T_2 \vdash M_2$.
3. We apply Lemma 7.4.7 to

$$\Sigma; M \uplus \left\{p\colon P \uplus \{t\colon \langle n; \mathbf{u}\rangle\}\right\} \vdash T \uplus \{t\colon (B, \mathtt{adv}(p); b)\}$$

to obtain $\Sigma; M_2 \vdash T_2$.

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-BOUND.

$$\frac{n \in \mathcal{N}}{\left(M, T \uplus \{t\colon (B \uplus \{p\colon \_\}, \mathtt{bound}(p); b)\}\right) \to \left(M, T \uplus \{t\colon (B \uplus \{p\colon n\}, b)\}\right)}$$

where S_1 is $\left(M, T \uplus \{t\colon (B \uplus \{p\colon \_\}, \mathtt{bound}(p); b)\}\right)$ and S_2 is

$$\left(M, T \uplus \{t\colon (B \uplus \{p\colon n\}, b)\}\right)$$

Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon (B \uplus \{p\colon \_\}, \mathtt{bound}(p); b)\}$. By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\operatorname{dom} T_1 \vdash \Delta \qquad \Delta; \operatorname{dom} T_1 \vdash M \qquad \Sigma; M \vdash T_1}{\langle \Delta; \Sigma\rangle \vdash \left(M, T_1\right)}$$

Let $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon (B \uplus \{p\colon n\}, b)\}$. To show that $\langle \Delta; \Sigma\rangle \vdash \left(M, T_2\right)$ holds, we need to establish the following.

1. We have that $\operatorname{dom} T_1 = \operatorname{dom} T_2$, hence $\operatorname{dom} T_2 \vdash \Delta$.
2. Since $\operatorname{dom} T_1 = \operatorname{dom} T_2$, then we have that $\Delta; \operatorname{dom} T_2 \vdash M$.
3. We apply Lemma 7.5.2 to

$$\Sigma; M \vdash T \uplus \{t\colon (B \uplus \{p\colon \_\}, \mathtt{bound}(p); b)\}$$

to obtain $\Sigma; M \vdash T_2$.

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-AWAIT.

$$\frac{\text{awaitAll}(M, t, B)}{\big(M, T \uplus \{t\colon (B, \texttt{await}; b)\}\big) \to \big(M, T \uplus \{t\colon (B, b)\}\big)}$$

where S_1 is $\big(M, T \uplus \{t\colon (B, \texttt{await}; b)\}\big)$ and S_2 is

$$\big(M, T \uplus \{t\colon (B, b)\}\big)$$

Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, \texttt{await}; b)\}$.

By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\text{dom}\, T_1 \vdash \Delta \qquad \Delta; \text{dom}\, T_1 \vdash M \qquad \Sigma; M \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M, T_1\big)}$$

Let $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, b)\}$. To show that $\langle \Delta; \Sigma \rangle \vdash \big(M, T_2\big)$ holds, we need to establish the following.

1. We have that $\text{dom}\, T_1 = \text{dom}\, T_2$, hence $\text{dom}\, T_2 \vdash \Delta$.
2. Again, since $\text{dom}\, T_1 = \text{dom}\, T_2$, then we have that $\Delta; \text{dom}\, T_2 \vdash M$.
3. We apply Lemma 7.6.2 to $\Sigma; M \vdash T \uplus \{t\colon (B, \texttt{next}; b)\}$ to obtain:

$$\Sigma; M \vdash T \uplus \{t\colon (B, b)\}$$

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-NEXT.

$$\big(M, T \uplus \{t\colon (B, \texttt{next}; b)\}\big) \to \big(\text{commit}(M, t), T \uplus \{t\colon (B, b)\}\big)$$

where S_1 is at the right-hand side and S_2 is at the left-hand side of the conclusion. Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, \texttt{next}; b)\}$.

By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\text{(i) dom}\, T_1 \vdash \Delta \qquad \text{(ii) } \Delta; \text{dom}\, T_1 \vdash M \qquad \text{(iii) } \Sigma; M \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M, T_1\big)}$$

Let $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, b)\}$, (iv) $M' = \text{commit}(M, t)$, (v) $\Delta' = \text{commit}_{\text{D}}(\Delta, t)$ (from Definition 7.7.1), and $N = \text{dom}\, T_1 = \text{dom}\, T_2$. To show that $\langle \Delta'; \Sigma \rangle \vdash \big(M', T_2\big)$ holds, we need to establish the following.

1. From (i) $N \vdash \Delta$, (v) $\Delta' = \mathrm{commit}_{\mathrm{D}}(\Delta, t)$, and Lemma 7.7.1, we also have that $N \vdash \Delta'$.

2. Since we have (i) $N \vdash \Delta$, (ii) $\Delta; N \vdash M$, (v), (iv), and Lemma 7.7.6, then we have $\Delta'; N \vdash M'$.

3. From Lemma 7.7.11, $\Sigma; M \vdash T_1$ (hypothesis), and (iv) $M' = \mathrm{commit}(M, t)$, we obtain $\Sigma'; M' \vdash T_2$.

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-FINISH.

$$\frac{S \stackrel{\text{def}}{=} \big(\emptyset, \{t_2 \colon (\emptyset, b_2)\}\big)}{\big(M, T \uplus \{t_1 \colon (B, \mathtt{finish}(b_2); b_1)\}\big) \to \big(M, T \uplus \{t_1 \colon S \triangleright (B, b_1)\}\big)}$$

where S_1 is at the right-hand side and S_2 is at the left-hand side of the conclusion. Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t_1 \colon (B, \mathtt{finish}(b_2); b_1)\}$.

By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\mathrm{dom}\, T_1 \vdash \Delta \qquad \Delta; \mathrm{dom}\, T_1 \vdash M \qquad \Sigma; M \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M, T_1\big)}$$

Let $T_2 \stackrel{\text{def}}{=} T \uplus \{t_1 \colon S \triangleright (B, b_1)\}$. To show that $\langle \Delta; \Sigma \rangle \vdash \big(M, T_2\big)$ holds, we need to establish the following.

1. We have that $\mathrm{dom}\, T_1 = \mathrm{dom}\, T_2$, hence $\mathrm{dom}\, T_2 \vdash \Delta$.

2. Again, since $\mathrm{dom}\, T_1 = \mathrm{dom}\, T_2$, then we have that $\Delta; \mathrm{dom}\, T_2 \vdash M$.

3. We apply Lemma 7.8.3 to $\Sigma; M \vdash T_1$ to obtain $\Sigma'; M \vdash T_2$.

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-RUN.

$$\frac{S_3 \to S_4}{\big(M, T \uplus \{t \colon S_3 \triangleright (B, b)\}\big) \to \big(M, T \uplus \{t \colon S_4 \triangleright (B, b)\}\big)}$$

where S_1 is at the right-hand side and S_2 is at the left-hand side of the conclusion. Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t \colon S_3 \triangleright (B, b)\}$.

By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\text{(i) } \operatorname{dom} T_1 \vdash \Delta \qquad \text{(ii) } \Delta; \operatorname{dom} T_1 \vdash M \qquad \text{(iii) } \Sigma; M \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M, T_1\big)}$$

Let $T_2 \stackrel{\text{def}}{=} T \uplus \{t \colon S_4 \triangleright (B, b)\}$. To show that $\langle \Delta; \Sigma \rangle \vdash \big(M, T_2\big)$ holds, we need to establish the following.

1. We have that $\operatorname{dom} T_1 = \operatorname{dom} T_2$, hence $\operatorname{dom} T_2 \vdash \Delta$.
2. Since $\operatorname{dom} T_1 = \operatorname{dom} T_2$, then we have that $\Delta; \operatorname{dom} T_2 \vdash M$.
3. Let $\tau_1 \stackrel{\text{def}}{=} S_3 \triangleright (B, b)$ We know that $T_1(t) = \tau_1$. From $\Sigma; M \vdash T_1$ (hypothesis) and $T_1(t) = \tau_1$, we get that (iv) $\Sigma' \stackrel{\text{def}}{=} \Sigma \uplus \{t \colon \Psi\}$, (v) $\vdash_t M : \Gamma$, (vi) $\Psi; \Gamma \vdash \tau_1$, and (vii) $\Sigma'; M \vdash T$.

   Inverting premise (vi) yields the following.

$$\frac{\text{(viii) } \Psi \vdash S_3 \qquad \text{(ix) } \langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, b)}{\Psi; \Gamma \vdash S_3 \triangleright (B, b)}$$

   Next, we apply the induction hypothesis to $\Psi \vdash S_3$ and $S_3 \to S_4$, and get that there exists a Ψ' such that (x) $\Psi' \vdash S_4$. Hence, we have premise (xi)

$$\frac{\text{(x) } \Psi' \vdash S_4 \qquad \text{(ix) } \langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, b)}{\Psi'; \Gamma \vdash S_4 \triangleright (B, b)}\ \textsc{T-T-F}$$

   Thus,

$$\frac{\text{(v) } \vdash_t M : \Gamma \qquad \text{(xi) } \Psi'; \Gamma \vdash \tau_2 \qquad \text{(vii) } \Sigma'; M \vdash T}{\Sigma' \uplus \{t \colon \Psi'\}; M \vdash T \uplus \{t \colon \tau_2\}}$$

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-JOIN.

$$\frac{S \text{ is halted}}{\big(M, T \uplus \{t \colon S \triangleright (B, b)\}\big) \to \big(M, T \uplus \{t \colon (B, b)\}\big)}$$

Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t \colon S \triangleright (B, b)\}$.

By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\operatorname{dom} T_1 \vdash \Delta \qquad \Delta; \operatorname{dom} T_1 \vdash M \qquad \Sigma; M \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M, T_1\big)}$$

Let $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, b)\}$. To show that $\langle \Delta; \Sigma \rangle \vdash \big(M, T_2\big)$ holds, we need to establish the following.

1. We have that $\operatorname{dom} T_1 = \operatorname{dom} T_2$, hence $\operatorname{dom} T_2 \vdash \Delta$.
2. Again, since $\operatorname{dom} T_1 = \operatorname{dom} T_2$, then we have that $\Delta; \operatorname{dom} T_2 \vdash M$.
3. We apply Lemma 7.9.1 to $c; b \to b'$ and $\Sigma; M \vdash T_1$ to obtain $\Sigma'; M \vdash T_2$.

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

Case R-FLOW.

$$\frac{c; b \to b'}{\big(M, T \uplus \{t\colon (B, c; b)\}\big) \to \big(M, T \uplus \{t\colon (B, b')\}\big)}$$

where S_1 is $\big(M, T \uplus \{t\colon (B, c; b)\}\big)$ and S_2 is

$$\big(M, T \uplus \{t\colon (B, b')\}\big)$$

Let $T_1 \stackrel{\text{def}}{=} T \uplus \{t\colon (B, c; b)\}$. By inversion of the hypothesis that abstract machine is well typed we get the following premises.

$$\frac{\operatorname{dom} T_1 \vdash \Delta \qquad \Delta; \operatorname{dom} T_1 \vdash M \qquad \Sigma; M \vdash T_1}{\langle \Delta; \Sigma \rangle \vdash \big(M, T_1\big)}$$

Let $T_2 \stackrel{\text{def}}{=} T \uplus \{t\colon (B \uplus \{p\colon n\}, b)\}$. To show that $\langle \Delta; \Sigma \rangle \vdash \big(M, T_2\big)$ holds, we need to establish the following.

1. We have that $\operatorname{dom} T_1 = \operatorname{dom} T_2$, hence $\operatorname{dom} T_2 \vdash \Delta$.
2. Since $\operatorname{dom} T_1 = \operatorname{dom} T_2$, then we have that $\Delta; \operatorname{dom} T_2 \vdash M$.
3. We apply Lemma 7.10.4 to

   $$\Sigma; M \vdash T \uplus \{t\colon (B, c; b)\}$$

   to obtain $\Sigma; M \vdash T_2$.

We apply rule T-AMACH to (1), (2), and (3) and conclude this case.

□

Chapter Eight

Progress

A type system that enjoys progress states that any typable term can reduce or the term is in its elementary form. The "elementary form" depends on the language. For example, in a numeric- and expression-based language the most elementary terms can be numbers and variables. In SBrenner, the elementary states are halted, only composed by tasks that terminated.

The property of progress implies *deadlock freedom.* As we capture the notion of "execution" with the reduction relation, then we can consider that a deadlocked state S is such that S cannot reduce, *i.e.,* for any state S' the relation $S \to S'$ does not hold. If S is rejected by the type system, then we are done. But by absurd, assume that the deadlocked state S is well typed. Then, by the property of progress, state S must reduce, and we reach a contradiction.

The main result of this section is Theorem 8.0.2. The intuition behind the proof follows. With Lemma 8.0.2 we order the task names in the state with $\leq_\Delta$ and pick the smallest task name t. We show that the predicate for the await holds for task t, thus the task addressed by t reduces.

Lemma 8.0.1. *If* $\operatorname{dom} P \neq \emptyset$*, then there exists a task name* $t \in \operatorname{dom} P$ *such that* $\text{await}(P, \text{localPhase}\, P(t))$.

Proof. Let X be $\{t\colon \text{localPhase}\, P(t) \mid \forall t \in \operatorname{dom} P\}$. It is easy to see that $\operatorname{dom} X = \operatorname{dom} P$. Let t_1 be such that $\forall t \in \operatorname{dom} X\colon X(t_1) \leq X(t)$.

Since $\operatorname{dom} P \neq \emptyset$ and $\operatorname{dom} X = \operatorname{dom} P$, then $\operatorname{dom} X \neq \emptyset$. We have that $\operatorname{dom} X \neq \emptyset$, then t_1 exists. Let $n = X(t_1) = \text{localPhase}\, P(t_1)$. Thus,

$$
\begin{aligned}
&\forall t \in \operatorname{dom} X\colon X(t) \geq X(t_1)\\
\equiv &\forall t \in \operatorname{dom} P\colon \text{localPhase}(P(t)) \geq n\\
\equiv &\,\text{await}(P, n)\\
\equiv &\,\text{await}(P, \text{localPhase}\, P(t_1))
\end{aligned}
$$

□

Definition 8.0.1 (Wait phase)**.**

$$\text{waitPhase}((n, a)) \stackrel{\text{def}}{=} n$$

Lemma 8.0.2. *If* $\neg\, \text{awaitAll}(M, t, B)$*, then there exists phaser name* p *such that*

1. $M(p) = P$
2. $\text{waitPhase}\, P(t) = n$
3. $\neg\, \text{await}(P, n)$

Proof. If predicate $\text{awaitAll}(M, t, B)$ does not hold, then by Definition 5.3.13 there exists a $p \in \text{dom}\, M$ where $M(p) = P$ and $t \in \text{dom}\, P$ such that

(i) $P = M(p)$,

(ii) $n = \text{localPhase}(P(t)) - B(p)$, and

(iii) $\neg\, \text{await}(P, n)$.

By Definition 5.3.12 we have that there exists a task name t' such that

$$\text{localPhase}\,(P(t')) < \text{localPhase}(P(t)) - B(p)$$

Given that $B(p) \geq 0$, then $\text{localPhase}\,(P(t')) < \text{localPhase}(P(t))$. Hence, $\neg\, \text{await}(P, \text{localPhase}(P(t')))$. □

Lemma 8.0.3. *If* $T(t) = (B, \texttt{await}; b)$*,* $M(p)(t) = v$*, and* $\Sigma; M \vdash T$*, then* $v = (n, \mathbf{a})$*.*

Proof. Applying Lemma 6.3.2 to $\Sigma; M \vdash T$ and $T(t) = (B, \texttt{await}; b)$, we get that there exist Γ and Ψ such that $\vdash_t M \colon \Gamma$ and $\Psi; \Gamma \vdash (B, \texttt{await}; b)$. By inverting the latter, we get that

$$\dfrac{\Gamma \vdash B \qquad \dfrac{\text{(i) } \Gamma \vdash \texttt{await}; b \colon \Gamma}{\Gamma \vdash \texttt{await}; b \colon \emptyset}}{\langle \emptyset; \emptyset \rangle; \Gamma \vdash (B, \texttt{await}; b)}$$

where Ψ is $\langle \emptyset; \emptyset \rangle$. By inverting (i) $\Gamma \vdash \texttt{await}; b \colon \Gamma$ we have that (ii) $\forall p \in \text{dom}\, \Gamma \implies \Gamma(p) = \mathbf{a}$. From $\vdash_t M \colon \Gamma$, $M(p)(t) = v$, and Lemma 6.4.3, we get that $v = (n, \mathbf{a})$. □

Lemma 8.0.4. *If* $\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)$*,* $M(p) = P$*,* $P(t_1) = \langle n_1; a_1 \rangle$*, and* $P(t_2) = \langle n_2; a_2 \rangle$*, then* $\Delta(t_1, t_2) = n_1 - n_2$*.*

Proof. Inverting $\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)$, yields $\Delta; N \vdash M$ where $N = \operatorname{dom} T$. Since $\Delta; N \vdash M$ and $M(p) = P$, then by Lemma 6.3.6 we have that $\Delta \vdash P$. Applying $N \vdash \Delta$, $\Delta \vdash P$, $P(t_1) = \langle n_1; a_1 \rangle$ to Lemma 6.3.8 and we get that there exists a phaser P' such that $P = P' \uplus \{t_1 \colon \langle n_1; a_1 \rangle\}$ and $\Delta; t_1; n_1 \vdash P'$. We know that if $P(t_2) = \langle n_2; a_2 \rangle$, then $P'(t_2) = \langle n_2; a_2 \rangle$. Finally, since $\Delta; t_1; n_1 \vdash P'$, $P'(t_2) = \langle n_2; a_2 \rangle$, then by Lemma 6.3.7 we have that $\Delta(t_1, t_2) = n_1 - n_2$. □

Lemma 8.0.5. *If* $\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)$, $M(p) = P$, $P(t_1) = \langle n_1; \_ \rangle$, $P(t_2) = \langle n_2; \_ \rangle$, *and* $\Delta(t_1, t_2) = z$, *then* $n_1 - n_2 = z$.

Proof. We apply $\Delta; N \vdash M$, $M(p) = P$, $P(t_1) = \langle n_1; a_1 \rangle$, and $P(t_2) = \langle n_2; a_2 \rangle$ to Lemma 8.0.4 and get that $\Delta(t_1, t_2) = n_1 - n_2$. But by hypothesis $\Delta(t_1, t_2) = z$, therefore $z = n_1 - n_2$. □

Corollary 8.0.1. *If* $\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)$, $M(p) = P$, $t_1 \operatorname{dom} P$, *and* $t_2 \in \operatorname{dom} P$, *then* $t_1 \leq_\Delta t_2 \iff \text{waitPhase}\, P(t_1) \leq \text{waitPhase}\, P(t_2)$.

Proof. ($\Longrightarrow$) Since we have $t_1 \leq_\Delta t_2$, then we have $\Delta(t_1, t_2) = n$ and $n \leq 0$. Applying Lemma 8.0.5 $\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)$, $M(p) = P$, $t_1 \operatorname{dom} P$, $t_2 \in \operatorname{dom} P$, and $\Delta(t_1, t_2) = z$, then $n_1 - n_2 = z$. Thus, by Definition 8.0.1 waitPhase $P(t_1) \leq$ waitPhase $P(t_2)$.

($\Longleftarrow$) From waitPhase $P(t_1) \leq$ waitPhase $P(t_2)$ and Definition 8.0.1 we get that
$P(t_1) = \langle n_1; a_1 \rangle$, $P(t_2) = \langle n_2; a_2 \rangle$, and $n_1 \leq n_2$. Thus, $n_1 - n_2 = z$ and $z \leq 0$. Since $\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)$, $M(p) = P$, $t_1 \operatorname{dom} P$, $t_2 \in \operatorname{dom} P$, $P(t_1) = \langle n_1; a_1 \rangle$, $P(t_2) = \langle n_2; a_2 \rangle$, then $\Delta(t_1, t_2) = z$. Hence, $t_1 \leq_\Delta t_2$. □

Lemma 8.0.6. *If* $\Psi \vdash S$, S *is not halted, then* $S = \big(M, T\big)$ *and there exists a task name* $t \in \operatorname{dom} T$ *such that*

$$T(t) = (B, \texttt{await}; b) \implies \text{awaitAll}(M, t, B)$$

Proof. The proof develops by contradiction. We have that for all $t \in \operatorname{dom} T$

$$\begin{gathered} \neg(T(t) = (B, \texttt{await}; b) \implies \text{awaitAll}(M, t, B)) \\ \equiv \\ T(t) = (B, \texttt{await}; b) \wedge \neg\, \text{awaitAll}(M, t, B) \end{gathered}$$

We show how to reach a contradiction where there exists a name t such that

$$\neg\, \text{awaitAll}(M, t, B) \qquad \text{awaitAll}(M, t, B)$$

Let $\Psi = \langle \Delta; \Sigma \rangle$. From inverting $\Psi \vdash S$ we get $\operatorname{dom} T \vdash \Delta$, which by inversion yields that $(\leq_\Delta, \operatorname{dom} T)$ is a total ordering. Given that $\operatorname{dom} T$ is finite and nonempty, then there exists a label $t \in \operatorname{dom} T$ such that

$$\forall t' \in \operatorname{dom} T \colon t \leq_\Delta t'$$

By Lemma 8.0.2, since $\neg\,\text{awaitAll}(M, t, B)$, then there exists a phaser name p such that (i) $M(p) = P$, (ii) localPhase $P(t) = n$, and (iii) $\neg\,\text{await}(P, n)$.

By Lemma 8.0.1, since $\text{dom}\,P \neq \emptyset$ (as $t \in \text{dom}\,P$), then there must exist a task name t' such that (iv) $\text{await}(P, \text{localPhase}\,P(t'))$. If task name t' is t, then we reach a contradiction because we have (iii) $\neg\,\text{await}(P, \text{localPhase}\,P(t))$ and (iv) $\text{await}(P, \text{localPhase}\,P(t))$. Otherwise, $t \neq t'$. Let localPhase $P(t') = n'$. Since (iv) $\text{await}(P, n')$, then $\forall t \in \text{dom}\,P$ s.t. localPhase $(P(t)) \geq n'$ and therefore $n \geq n'$.

But, if $n = n'$, then (iii) $\neg\,\text{await}(P, \text{localPhase}\,P(t))$ would not hold and we would reach a contradiction. Hence, localPhase $(P(t)) >$ localPhase $(P(t'))$.

Recall that $t \leq_\Delta t'$, hence since $\langle\Delta; \Sigma\rangle \vdash \big(M, T\big)$, $M(p) = P$, $t_1\,\text{dom}\,P$, $t_2 \in \text{dom}\,P$, and $t_1 \leq_\Delta t_2$, then by Corollary 8.0.1

$$\text{waitPhase}\,P(t_1) \leq \text{waitPhase}\,P(t_2)$$

Applying Lemma 8.0.3 to $T(t) = (B, \texttt{await}; b)$, $M(p)(t) = v$, and $\Sigma; M \vdash T$, then $a_1 = \mathbf{a}$. Similarly, $a_2 = \mathbf{a}$. Thus, localPhase $(P(t)) =$ waitPhase $(P(t)) + 1$ and localPhase $(P(t')) =$ waitPhase $(P(t')) + 1$, which means that

$$\text{localPhase}\,(P(t)) \leq \text{localPhase}\,(P(t'))$$

□

Lemma 8.0.7. *If* $\vdash_t M\colon \Gamma$, $\Psi; \Gamma \vdash \tau$, $\Sigma; M \vdash T$, $T(t) = \tau = (B, i; b)$, *and* $i = \texttt{await} \implies \text{awaitAll}(M, t, B)$, *then there exists a state* S *such that*

$$\big(M, T \uplus \{t\colon (B, i; b)\}\big) \to S$$

Proof. The proof follows by inspection of the last typing rule applied. Inverting $\Psi; \Gamma \vdash (B, i; b)$ yields the next two premises.

$$\dfrac{\text{(i) } \Gamma \vdash B \qquad \dfrac{\text{(ii) } \Gamma \vdash i\colon \Gamma' \qquad \cdots}{\Gamma \vdash i; b\colon \emptyset}}{\langle\emptyset; \emptyset\rangle; \Gamma \vdash (B, i; b)}$$

Case T-PHASER:

$$\dfrac{p \notin \text{dom}\,\Gamma}{\Gamma \vdash p = \texttt{newPhaser}()\colon \Gamma \uplus \{p\colon \mathbf{u}\}}$$

where $i \stackrel{\text{def}}{=} p = \texttt{newPhaser}()$ and $\Gamma' \stackrel{\text{def}}{=} \Gamma \uplus \{p\colon \mathbf{u}\}$. Let q be such that $q \notin \operatorname{dom} M$ and $q \notin \operatorname{bn}(b)$. The case concludes with rule R-PHASER:

$$\frac{q \notin \operatorname{bn}(b) \qquad M' \stackrel{\text{def}}{=} M \uplus \big\{q\colon \{t\colon \langle 0; \mathbf{u}\rangle\}\big\} \qquad B' \stackrel{\text{def}}{=} B \uplus \{q\colon 0\}}{\big(M, T \uplus \{t\colon (B, p = \texttt{newPhaser}(); b)\}\big) \to \big(M', T \uplus \{t\colon (B', b[q/p])\}\big)}$$

Case T-DEREG:

$$\Gamma' \uplus \{p\colon a\} \vdash \texttt{dereg}(p)\colon \Gamma'$$

where $\Gamma \stackrel{\text{def}}{=} \Gamma' \uplus \{p\colon a\}$ and $i \stackrel{\text{def}}{=} \texttt{dereg}(p)$. We know that (iii) $\Gamma(p) = a$. With Lemma 6.4.3 and (iii) $\Gamma(p) = a$, we get that $M(p)(t) = (n, a)$ Thus, (iv) $M = M' \uplus \big\{p\colon P \uplus \{t\colon \langle n; a\rangle\}\big\}$. From (i) $\Gamma \vdash B$, $p \in \operatorname{dom}\Gamma$, and Lemma 6.3.3, then there exist Γ'' and B' such that $B = B' \uplus \{p\colon n\}$.

Hence,

$$\begin{aligned}&\big(M' \uplus \big\{p\colon P \uplus \{t\colon v\}\big\}, T \uplus \big\{t\colon (B' \uplus \{p\colon n\}, \texttt{dereg}(p); b)\big\}\big)\\ \to&\big(M' \uplus \big\{p\colon P\big\}, T \uplus \big\{t\colon (B', b)\big\}\big)\end{aligned}$$

Case T-ADV:

$$\Gamma'' \uplus \{p\colon \mathbf{u}\} \vdash \texttt{adv}(p)\colon \Gamma'' \uplus \{p\colon \mathbf{a}\}$$

where Γ is $\Gamma' \uplus \{p\colon \mathbf{u}\}$ and i is $\texttt{adv}(p)$. We have that (iii) $\Gamma(p) = \mathbf{u}$. With Lemma 6.4.3, $\vdash_t M\colon \Gamma$ (hypothesis), and (iii) $\Gamma(p) = \mathbf{u}$, we get that $M(p)(t) = (n, \mathbf{u})$. Thus, (iv) $M = M' \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{u})\}\big\}$. Hence,

$$\begin{aligned}&\big(M' \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{u})\}\big\}, T \uplus \{t\colon (B, \texttt{adv}(p); b)\}\big)\\ \to&\big(M' \uplus \big\{p\colon P \uplus \{t\colon (n, \mathbf{a})\}\big\}, T \uplus \{t\colon (B, b)\}\big)\end{aligned}$$

Case T-AWAIT:

$$\frac{\forall p \in \operatorname{dom}\Gamma\colon \Gamma(p) = \mathbf{a}}{\Gamma \vdash \texttt{await}\colon \Gamma}$$

where $i = \texttt{await}$ and $\Gamma' \stackrel{\text{def}}{=} \Gamma$.

From the hypothesis, we have that $\operatorname{awaitAll}(M, t, B)$. Thus, the state reduces with rule R-AWAIT.

Case T-NEXT:

$$\{p_1\colon \mathbf{a}, \ldots, p_n\colon \mathbf{a}\} \vdash \texttt{next}\colon \{p_1\colon \mathbf{u}, \ldots, p_n\colon \mathbf{u}\}$$

where $\Gamma \stackrel{\text{def}}{=} \{p_1\colon \mathbf{a}, \ldots, p_n\colon \mathbf{a}\}$, $i \stackrel{\text{def}}{=} \texttt{next}$, and $\Gamma' \stackrel{\text{def}}{=} \{p_1\colon \mathbf{u}, \ldots, p_n\colon \mathbf{u}\}$.

Since we have $\vdash_t M\colon \Gamma$, $\forall p \in \operatorname{dom}\Gamma\colon \Gamma(t) = \mathbf{a}$, and Lemma 8.0.11 we get that $\operatorname{commit}(M, t) = M'$ and the state reduces with R-NEXT.

Case T-ASYNC:

$$\frac{\Gamma \vdash s : \Gamma_1 \qquad \Gamma_1 \vdash b : {}_1\emptyset}{\Gamma \vdash \texttt{async}(s, b_1) : \Gamma}$$

where $i \stackrel{\text{def}}{=} \texttt{async}(s, b_1)$ and $\Gamma \stackrel{\text{def}}{=} \Gamma'$.

From Lemma 8.0.10, $\vdash_t M : \Gamma$, and $\Gamma \vdash s : \Gamma_1$, then we have that

$$\text{copy}(s, t, t', M) = M'$$

Hence, we conclude the case with rule R-ASYNC.

Case T-FINISH:

$$\frac{\emptyset \vdash b' : \emptyset}{\Gamma \vdash \texttt{finish}(b) : \Gamma}$$

where $i \stackrel{\text{def}}{=} \texttt{finish}(b')$ and $\Gamma' \stackrel{\text{def}}{=} \Gamma$. The case holds by direct application of rule R-FINISH.

□

Lemma 8.0.8. *If* $\big(M_1, T_1\big) \to \big(M_2, T_2\big)$*, then*

$$\big(M_1, T_1 \uplus \{t : (B, \texttt{end})\}\big) \to \big(M_2, T_2 \uplus \{t : (B, \texttt{end})\}\big)$$

Proof. By inspection of each reduction rule. □

Lemma 8.0.9. *If* $\Gamma_1 \vdash s : \Gamma_2$*, then* $p \in s \iff p \in \operatorname{dom}\Gamma_1 \cap \operatorname{dom}\Gamma_2$.

Proof. The proof follows by induction on the typing relation. We do a case analysis on the derivation of the last rule applied.

- Case T-A-C:

  $$\frac{\text{(i) } \Gamma_1(q) = a \qquad \text{(ii) } \Gamma_1 \vdash s' : \Gamma}{\Gamma_1 \vdash s' \uplus \{q\} : \Gamma \uplus \{q : a\}}$$

  where Γ_2 is $\Gamma \uplus \{q : a\}$ and s is $s' \uplus \{q\}$. If $p = q$, then we are done. Otherwise, $p \neq q$. Applying the induction hypothesis to (ii) $\Gamma_1 \vdash s' : \Gamma$ we get that $p \in s' \iff p \in \operatorname{dom}\Gamma_1 \cap \operatorname{dom}\Gamma$. We have that $p \neq q$ and that $q \notin \operatorname{dom}\Gamma_1 \cap \operatorname{dom}\Gamma$, hence $p \in s \iff p \in \operatorname{dom}\Gamma_1 \cap \operatorname{dom}\Gamma_2$.

- Case T-A-N:

  $$\Gamma \vdash \emptyset : \emptyset$$

  where Γ_2 is $\emptyset$. We end in a contradiction as we have that $p \in \emptyset$.

□

Lemma 8.0.10. *If* $\vdash_t M_1 : \Gamma_1$ *and* $\Gamma_1 \vdash s : \Gamma_2$, *then* $\mathrm{copy}(s, t_1, t_2, M_1) = M_2$.

Proof. The proof follows by induction on the typing relation. We do a case analysis on the derivation of the last rule applied.

- Case T-PERM-NIL:

$$\vdash_t \emptyset : \emptyset$$

  where $M_1 = \emptyset$ and Γ_1 is $\emptyset$. Since $\Gamma_1 = \emptyset$, then we can invert $\emptyset \vdash s : \Gamma_2$ and obtain that $s = \emptyset$. We conclude this case with rule CPY-NIL.

- Case T-PERM-SKIP:

$$\frac{\text{(i) } \vdash_t M : \Gamma_1 \qquad \text{(ii) } t \notin \mathrm{dom}\, P}{\vdash_t M \uplus \{p : P\} : \Gamma_1}$$

  where $M_1 = M \uplus \{p : P\}$. We apply $\vdash_t M : \Gamma_1$ to the induction hypothesis and get that (iii) $\mathrm{copy}(s, t_1, t_2, M) = M'$. Since we have $t \notin \mathrm{dom}\, M_1(p)$, then from Lemma 6.4.3 we know that $p \notin \mathrm{dom}\, \Gamma_1$, thus from Lemma 8.0.9 and $\Gamma_1 \vdash s : \Gamma_2$, we have that (iv) $p \notin s$. Hence,

$$\frac{\text{(iii) } \mathrm{copy}(s, t, t', M) = M' \qquad \text{(iv) } p \notin s}{\mathrm{copy}(s, t, t', M \uplus \{p : P\}) = M' \uplus \{p : P\}}\ \textsc{Cpy-skip}$$

- Case T-PERM-CONS:

$$\frac{\text{(i) } \vdash_t M : \Gamma \qquad \text{(ii) } P(t) = \langle n; a \rangle}{\vdash_t M \uplus \{p : P\} : \Gamma \uplus \{p : a\}}$$

  where $M_1 = M \uplus \{p : P\}$ and Γ_2 is $\Gamma \uplus \{p : a\}$. We test the membership of $p \in s$:

  - Case (iv) $p \in s$, then $s = s' \uplus \{p\}$. From Lemma 6.3.1 and $\Gamma_1 \vdash s : \Gamma_2$ then there exists a typing Γ_3 such that (iii) $\Gamma_2 = \Gamma_3 \uplus \{p : a\}$, $\Gamma_1(p) = a$, (iv) $\Gamma_1 \vdash s' : \Gamma_2$. Since we have (iii) $\Gamma_2 = \Gamma_3 \uplus \{p : a\}$ and $\Gamma_2 = \Gamma \uplus \{p : a\}$, then $\Gamma_3 = \Gamma$ and therefore (v) $\Gamma_1 \vdash s' : \Gamma$. Applying the induction hypothesis to (i) $\vdash_t M : \Gamma$, and (v) $\Gamma_1 \vdash s' : \Gamma$, yields (vi) $\mathrm{copy}(s', t_1, t_2, M) = M'$. Let $v \stackrel{\text{def}}{=} \langle n; a \rangle$ and $P' \stackrel{\text{def}}{=} P \uplus \{t' : v\}$.

$$\frac{\text{(vi) } \mathrm{copy}(s', t_1, t_2, M) = M' \qquad \text{(ii) } P(t) = v}{\mathrm{copy}(s' \uplus \{p\}, t_1, t_2, M \uplus \{p : P\}) = M' \uplus \{p : P'\}}\ \textsc{Cpy-cons}$$

– Case (iv) $p \notin s$. Since $\Gamma \uplus \{p\colon a\} \vdash s\colon \Gamma_2$ and $p \notin s$, then from Lemma 6.5.1 $\Gamma \vdash s\colon \Gamma_2$. We apply the induction hypothesis to $\vdash_t M\colon \Gamma$ and $\Gamma \vdash s\colon \Gamma_2$ and obtain (iii) $\mathrm{copy}(s, t_1, t_2, M) = M'$. Therefore,

$$\frac{\text{(iii) } \mathrm{copy}(s, t_1, t_2, M) = M' \qquad \text{(iv) } p \notin s}{\mathrm{copy}(s, t_1, t_2, M \uplus \{p\colon P\}) = M' \uplus \{p\colon P\}}\ \textsc{Cpy-skip}$$

□

Lemma 8.0.11. *If* $\vdash_t M\colon \Gamma$ *and* $\forall p \in \mathrm{dom}\,\Gamma\colon \Gamma(t) = \mathbf{a}$*, then* $\mathrm{commit}(M, t) = M'$.

Proof. The proof follows by induction on the structure of the typing relation. We proceed with a case analysis on the derivation of the last rule applied.

- Case T-perm-nil:

$$\vdash_t \emptyset\colon \emptyset$$

  The case holds with rule Com-n.

- Case T-perm-skip:

$$\frac{\text{(i) } \vdash_t M_1\colon \Gamma \qquad \text{(ii) } t \notin \mathrm{dom}\, P}{\vdash_t M' \uplus \{p\colon P\}\colon \Gamma}$$

  where M is $M_1 \uplus \{p\colon P\}$. Applying the induction hypothesis to $\vdash_t M_1\colon \Gamma$ and $\forall p \in \mathrm{dom}\,\Gamma\colon \Gamma(t) = \mathbf{a}$, we get that $\mathrm{commit}(M_1, t) = M_2$. Thus, we conclude this case by applying rule Com-s to the latter and to (i).

- Case T-perm-cons:

$$\frac{\text{(i) } \vdash_t M_1\colon \Gamma_1 \qquad \text{(ii) } P(t) = \langle \_; a\rangle}{\vdash_t M_1 \uplus \{p\colon P\}\colon \Gamma_1 \uplus \{p\colon a\}}$$

  where M is $M_1 \uplus \{p\colon P\}$ and Γ is $\Gamma_1 \uplus \{p\colon a\}$. From $\forall p \in \mathrm{dom}\,\Gamma\colon \Gamma(t) = \mathbf{a}$ (hypothesis) we get that $a = \mathbf{a}$ and $\Gamma = \Gamma_1 \uplus \{p\colon \mathbf{a}\}$. Thus, (iii) $\forall p \in \mathrm{dom}\,\Gamma_1\colon \Gamma_1(t) = \mathbf{a}$. Applying the induction hypothesis to (i) and (iii) yields that (iv) $\mathrm{commit}(M_1, t) = M_2$. We conclude this case applying rule Com-c.

□

Theorem 8.0.2 (Progress)**.** *If* $\Psi \vdash S_1$*, then* S_1 *is halted or there exists a state* S_2 *such that* $S_1 \to S_2$.

Proof. If S_1 is halted, then we are done. For the remainder of the proof we know that S_1 is not halted, so we must show that there exists an abstract machine S_2 such that $S_1 \to S_2$; the proof follows by induction on the reduction relation.

To show that S_1 reduces we perform a structural induction on S_1. From $\Psi \vdash S_1$, S_1 not halted, and Lemma 8.0.6

$$\begin{gathered} S_1 = \big(M, T\big) \\ \text{(i)}\ \exists t\colon T(t) = \tau = (B, \texttt{await}; b) \implies \text{awaitAll}(M, t, B) \end{gathered}$$

Let $N = \operatorname{dom} T$. Inverting hypothesis $\Psi \vdash S_1$ yields:

$$\frac{\text{(ii)}\ N \vdash \Delta \qquad \Delta; N \vdash M \qquad \Sigma; M \vdash T}{\langle \Delta; \Sigma \rangle \vdash \big(M, T\big)}\ \textsc{T-amach}$$

where $\Psi = \langle \Delta; \Sigma \rangle$. From $\Sigma; M \vdash T$, $T(t) = \tau$, and Lemma 6.3.2, we get there exists a phaser map T_t such that (iii) $\Sigma = \Sigma_t \uplus \{t\colon \Psi_t\}$, (iv) $\vdash_{\Psi_t} M\colon \Gamma_t$, (v) $\Psi_t; \Gamma_t \vdash \tau$, and (vi) $\Sigma_t; M \vdash T_t$. By inverting (v) $\Psi_t; \Gamma_t \vdash \tau$ there are two cases to consider: τ is either a regular task or a finish task.

Case τ is $S \triangleright (B, b)$. Inverting (v) $\Psi_t; \Gamma_t \vdash S \triangleright (B, b)$ results in the following premises.

$$\frac{\text{(i)}\ \Psi_t \vdash S \qquad \langle \emptyset; \emptyset \rangle; \Gamma_t \vdash (B, b)}{\Psi_t; \Gamma_t \vdash S \triangleright (B, b)}$$

Applying the induction hypothesis to $\Psi_t \vdash S$ yields two sub-cases to consider.

- Sub-case S is halted. Thus, S_1 can reduce with rule R-JOIN.
- Sub-case there exists a S' such that $S \to S'$. Thus, S_1 is ready to reduce with R-RUN.

Case τ is (B, b). We do a case analysis on the structure of b:

- Case b is end. Applying Lemma 6.5.6 to $\Psi \vdash S_1$, yields that $\langle \Delta; \Sigma \rangle \vdash \big(M, T_t\big)$ Applying the induction hypothesis to $\langle \Delta; \Sigma \rangle \vdash \big(M, T_t\big)$, we get that $\big(M, T_t\big) \to \big(M', T_t'\big)$. Hence, from Lemma 8.0.8 the case holds.
- Case b is $i; b'$. We conclude this case by applying Lemma 8.0.7 to (iv) $\vdash_t M\colon \Gamma_t$, (v) $\Psi_t; \Gamma_t \vdash \tau$, and (vi) $\Sigma_t; M \vdash T_t$.

□

Chapter Nine

Conclusion

This thesis proposes two comprehensive solutions for the problem of barrier deadlocks. First, a general runtime verification technique with an implementation that is language-agnostic, distributed, and fault-tolerant. Second, a programming model that is deadlock-free by construction.

Section 9.1 summarises the thesis and highlights our technical contributions. Section 9.3 discusses about future directions of our work that include integrating the programming model of SBrenner and Armus in a single language, mechanising the theory presented in this thesis, and extending Armus to verify MPI and HJ applications.

9.1 Contributions

This thesis presents a theoretical framework for reasoning about general barrier synchronisation patterns in the form of a minimal language called Brenner. Programs written in Brenner use a single abstraction to perform any of the synchronisation patterns surveyed in Section 2.1. We introduce two complementary verification techniques that handle barrier deadlocks: a runtime technique for existing programs, and a novel parallel programming model that is deadlock free by design. The correctness of our runtime verification technique is established against the semantics of Brenner. To define our deadlock free programming model we introduce SBrenner, an extension of Brenner that restricts certain behaviours that may deadlock.

Brenner [31] Our minimal language is the cornerstone of this thesis, as it provides the definitions used in our two main contributions. Two factors shape Brenner. The first factor is a comprehensive survey of language abstractions that perform barrier synchronisation to gives us confidence of the generality of our own abstraction. Our survey categorises the origin of various properties that influence the barrier synchronisation mechanism throughout the history of parallel computing, going as far back as the first parallel computers in the 1960s. The second factor is a search for the fundamental concepts behind phasers [94],

the unifying abstraction we use in Brenner. We define phasers as a data structure that consists of multiple event counters [90], a classic synchronisation mechanism used in operating system design. Our version of phasers has fewer and simpler primitives, yet it exhibits more synchronisation patterns than in [94].

Event counters, that inspire our version of phasers, can be seen as a synchronisation mechanism that allows for tasks to await a certain logical time in the sense of Lamport's clocks [67]. We leverage the connection between phasers and logical clocks in the design of Armus. Departing from state-of-the-art techniques, we propose a novel representation of concurrency constraints based on logical time that dramatically improves the scalability of distributed deadlock detection.

Technical contributions:

- We introduce Brenner, a minimal language for reasoning about general barrier synchronisation and task parallelism.
- Brenner is mechanised in Coq along with some reduction examples. The source code is available online.[1]

Armus [31] We put forward Armus, a runtime verification tool for barrier deadlocks that features distributed deadlock detection and a scalable graph analysis technique that automatically switches between two graph models. The graph-based deadlock verification of Armus is formalised and shown to be sound and complete against Brenner. We establish an equivalence theorem between utilising two graph models (WFG and SG) for deadlock detection; this result enables us to use the WFG to prove our results, and choose automatically between the WFG and the SG during verification. Our adaptive model selection dramatically increases the performance against a fixed model selection. The runtime overhead of the deadlock detection is low for up to 64 tasks, in most cases negligible, even when considering distributed benchmarks. We present two applications: Armus-X10 monitors any unchanged X10 program for deadlocks; JArmus is a library to verify Java programs. To the best of our knowledge, our work is the first dynamic verification tool that can correctly detect Java and X10 barrier deadlocks.

Technical contributions:

- The graph-based deadlock verification of Armus is formalised and shown to be sound and complete against Brenner.
- We establish an equivalence theorem between utilising two graph models (WFG and SG) for deadlock detection. Such result enables us to prove

[1] `https://bitbucket.org/cogumbreiro/brenner-coq/`

our results with the WFG, and let the tool automatically choose from the two models at run time. Our model selection technique increases the performance of deadlock checking, against the usual approach of having a fixed graph model.

- To the best of our knowledge, our work is the first dynamic verification tool that can correctly detect Java and X10 barrier deadlocks.

SBrenner [77, 32] We introduce SBrenner a minimal deadlock-free language for fork/join and cyclic barrier synchronisation, by extending Brenner. Chapter 5 defines an operational semantics and a type system that unifies a deadlock-free semantics of clocks, regular phasers, and phaser beams, but goes further by allowing tasks to be ahead of others by a bounded number of phases, available per task and per phaser. A novelty of SBrenner is to present a deadlock-free version of two synchronisation patterns available in HJ: bounded phaser synchronisation, and tasks can advance their phases without waiting for others. SBrenner can be used as a blueprint to develop deadlock-free parallel programming libraries, so we make available a Java prototype of this programming model.[2]

Technical contributions:

- The property of subject-reduction, described in Chapter 7, ensures programs retain their validity (*i.e.,* well typedness) as they execute, which means that a program deemed valid does not become invalid by executing.
- The property of progress, described in Chapter 8, shows that valid programs can always execute, which implies deadlock freedom.
- The deadlock-free programming model of SBrenner subsumes those of Habanero-Java and of X10. Our work is the first to establish the property of deadlock freedom stated in [92].
- A Java prototype of the deadlock-free programming model put forward by SBrenner.

9.2 Summary of personal publications

[77] Francisco Martins, Vasco T. Vasconcelos, and Tiago Cogumbreiro. Types for X10 Clocks. In *Post-proceedings of PLACES'10*, volume 69 of *EPTCS*, pages 111–129, 2011

[2] `https://bitbucket.org/cogumbreiro/brenner-java/`

[32] Tiago Cogumbreiro, Francisco Martins, and Vasco Thudichum Vasconcelos. Coordinating phased activities while maintaining progress. In *Proceedings of COORDINATION'13*, volume 7890, pages 31–44, 2013

[31] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. Dynamic deadlock verification for general barrier synchronisation. In *Proceedings of PPoPP'15*, 2015. To appear

9.3 Future work

We envision three future directions for our work. First, the integration of our techniques of static and runtime verification. Second, the mechanisation our theory. And third, extending Armus to verify HJ and MPI.

Two-stage deadlock verification The overarching goal of this thesis is to improve the productivity of multicore programmers, by resorting to software verification. We envision integrating our main contributions in a compiler. The outcome is a programming language that is *aware* of all barrier deadlocks. Our idealised verification technique comprises two steps:

1. The compiler checks if the given program complies with the programming model in SBrenner. Compliant programs are safe from deadlocks by constructions, so they can run without runtime checks.

2. Programs that fail to comply with the programming model of SBrenner are instrumented and verified by Armus at run time.

There are some specific tasks that can improve our contributions in static and runtime verification.

Certified verification For static verification we would like to have: a machine checked version of our theory, and a certified implementation of phasers. A certified algorithm is an algorithm that is mechanically checked to comply with a given formal specification, *i.e.,* the algorithm is accompanied by the proof of its correction. Some of this work started already. We mechanised in Why3 [38] the proofs for the invariant of the phase difference (*cf.* Chapter 8), adapted to primitives of HJ. The source code is available online.[3]

Why3 is helpful for prototyping formal results. The tool includes a language called Why to define functions, syntactic terms, inference rules, and lemmas. For example, a phaser map M can be defined as a function `pm` that accepts two

[3] `https://bitbucket.org/cogumbreiro/hj-why3`

parameters, `phid` for p and `tid` for t, and yields an optional `taskview` for type v, which means that the function returns either an undefined value or a task view.

```
function pm phid tid : option taskview
```

Similarly, we can define Δ as follows.

```
function diff tid tid: option int
```

We can define axioms for results that are already shown in pen and paper proofs. For instance, in our formalisation we assume Lemma 8.0.4 with axiom `diff_def`, that states that any phase difference in phaser map `pm` is also in `diff`.

```
axiom diff_def:
  forall t1 t2 i n1 n2 tv1 tv2 p.
  pm p t1 = Some tv1 -> pm p t2 = Some tv2 ->
  wait_phase tv1 n1 -> wait_phase tv2 n2 ->
  i = (i1 - i2)
  ->
  diff t1 t2 = Some i
```

This can be informally written as follows.

Lemma 9.3.1. *If* $M(p)(t_1) = (n_1, \_)$, $M(p)(t_2) = (n_2, \_)$, *then* $\Delta(t_1, t_2) = n_1 - n_2$.

A benefit of Why3 is its integration with automatic theorem provers. In our mechanisation, lemma `total_for_wait_tids` establishes that relation $\leq_\Delta$ is total. Why3 uses an off-the-shelf theorem prover to automatically prove this result for us.

```
lemma total_for_wait_tids:
  forall x y.
  wait_tid x -> wait_tid y
  ->
  diff_le x y \/ diff_le y x
```

Lemmas that cannot be discharged automatically are handled by the user with a proof assistant like Coq [78].

Finally, there is some work to be done in formalisation of Armus. While our work in Chapter 4 pushes the state-of-the-art of formal runtime verification of barrier deadlocks, we still lack a mathematical description of our distributed algorithm. Since our version is not too different than the original [65], the opportunity is ripe to produce a certified verification algorithm using a tool like Coq and Why3.

Runtime verification We intend to verify HJ programs, as it will exercise the expressiveness of Armus. HJ features abstractions with complex synchronisation patterns, such as the bounded producer-consumer. Armus currently simplifies the graph generation process by ignoring the ordering of events from the same phaser. While this simplification of Brenner increases the performance of Armus to check all barriers abstractions in X10 and Java, it limits its use in the context of HJ.

Another direction is the verification of MPI programs, which introduces point-to-point synchronisation and enable a direct comparison with state-of-the-art in barrier deadlock detection. One of the biggest difficulties is verifying of a form of non-deterministic point-to-point synchronisation where a receiver task selects non-deterministically one message from possible multiple senders. Hilbrich *et al.* extended the WFG to support this non-deterministic point-to-point synchronisation in [51]. We need to investigate the impact of this specific graph model in our model selection technique.

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] José L. Abellán, Juan Fernández, and Manuel E. Acacio. Efficient hardware barrier synchronization in many-core CMPs. *Transactions on Parallel and Distributed Systems*, 23(8):1453–1466, 2012.

[3] Shail Aditya, Joseph E. Stoy, and Arvind. Semantics of barriers in a non-strict, implicitly-parallel language. In *Proceedings of FPCA'95*, pages 204–215. ACM, 1995.

[4] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of PADTAD'06*, pages 51–60. ACM, 2006.

[5] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of PPoPP'10*, pages 183–193. ACM, 2007.

[6] Shivali Agarwal, Saurabh Joshi, and Rudrapatna K. Shyamasundar. Distributed generalized dynamic barrier synchronization. In *Proceedings of ICDCN'11*, pages 143–154. Springer, 2011.

[7] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of POPL'98*, pages 342–354. ACM, 1998.

[8] Memory Alpha. Brenner. `en.memory-alpha.org/wiki/Brenner`, 2011. Accessed in October 2014.

[9] Norbert S. Arenstorf and Harry F. Jordan. Comparing barrier algorithms. *Parallel Computing*, 12(2):157–170, 1989.

[10] Arvind, Jan-Willem Maessen, Rishiyur S. Nikhil, and Joseph E. Stoy. λ_s: an implicitly parallel λ-calculus with letrec, synchronization and side-effects. *Electronic Notes Theoretical Computer Science*, 16(3):265–290, 1998.

[11] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[12] Daniel Atkins, Alex Potanin, and Lindsay Groves. The design and implementation of clocked variables in X10. In *Proceedings of ACSC'13*, pages 87–95. Australian Computer Society, 2013.

[13] Daniel Atkins, Alex Potanin, and Lindsay Groves. The design and implementation of clocked variables in X10. In *Proceedings of ACSC'13*, pages 87–95. Australian Computer Society, 2013.

[14] David A. Bader and Kamesh Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings of HiPC'05*, volume 3769 of *LNCS*, pages 465–476. Springer, 2005.

[15] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2nd edition, 2009.

[16] Hendrik P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[17] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The illiac iv computer. *IEEE Transactions on Computing*, 17(8):746–757, 1968.

[18] Maurice Bataille. Something old: the Gamma 60 the computer that was ahead of its time. *SIGARCH Computer Architecture News*, 1:10–15, 1972.

[19] Andrew D. Birrell. Implementing condition variables with semaphores. In Andrew Herbert, Karen Spärck Jones, David Gries, and Fred B. Schneider, editors, *Computer Systems*, Monographs in Computer Science, pages 29–37. Springer, 2004.

[20] Stefan Blom, Joseph Kiniry, and Marieke Huisman. How do developers use APIs? a case study in concurrency. In *Proceedings of ICECCS'13*, pages 212–221, 2013.

[21] Gérard Boudol. A deadlock-free semantics for shared memory concurrency. In *Proceedings of ICTAC'09*, pages 140–154. Springer, 2009.

[22] Eugene D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, 15:295–307, 1986.

[23] Luca Cardelli. Type Systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[24] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of PPPJ'11*, pages 51–61. ACM, 2011.

[25] Soumen Chakrabarti, Manish Gupta, and Jong-Deok Choi. Global communication analysis and optimization. *SIGPLAN Notices*, 31(5):68–78, 1996.

[26] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[27] Mani Chandy, lan Foster, Ken Kennedy, Charles Koelbel, and Chau-Wen Tseng. Integrated support for task and data parallelism. *International Journal of High Performance Computing Applications*, 8(2):80–98, 1994.

[28] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA'05*, pages 519–538. ACM, 2005.

[29] Sung-Eun Choi and Lawrence Snyder. Quantifying the effects of communication optimizations. In *Proceedings of ICPP'97*, pages 218–222, 1997.

[30] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):pp. 345–363, 1936.

[31] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. Dynamic deadlock verification for general barrier synchronisation. In *Proceedings of PPoPP'15*, 2015. To appear.

[32] Tiago Cogumbreiro, Francisco Martins, and Vasco Thudichum Vasconcelos. Coordinating phased activities while maintaining progress. In *Proceedings of COORDINATION'13*, volume 7890, pages 31–44, 2013.

[33] Melvin E. Conway. A multiprocessor system design. In *Proceedings of AFIPS '63 (Fall)*, pages 139–146. ACM, 1963.

[34] James Coyle, Indranil Roy, Marina Kraeva, and Glenn R. Luecke. UPC-CHECK: a scalable tool for detecting run-time errors in Unified Parallel C. *Computer Science*, 28(2-3):203–209, 2013.

[35] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. Resilient X10: Efficient failure-aware programming. In *Proceedings of PPoPP'14*, pages 67–80. ACM, 2014.

[36] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computing in Science and Engineering*, 5(1):46–55, 1998.

[37] Tarek El-Ghazawi and Lauren Smith. UPC: Unified Parallel C. In *Proceedings of SC'06*. ACM, 2006.

[38] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Proceedings of ESOP'13*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[39] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of DSN'10*, pages 221–230, 2010.

[40] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High-Performance Computing*, 8, 1994.

[41] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of PLDI'98*, pages 212–223. ACM, 1998.

[42] Michael A. Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan. Performance and scalability of the NAS Parallel Benchmarks in Java. In *Proceedings of IPDPS'03*. IEEE, 2003.

[43] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of OOPSLA'07*, pages 57–76. ACM, 2007.

[44] Milos Gligoric, Peter C. Mehlitz, and Darko Marinov. X10X: Model checking a new programming language with an "old" model checker. In *Proceedings of ICST'12*, pages 11–20. IEEE, 2012.

[45] Ganesh L. Gopalakrishnan and Robert M. Kirby. Top ten ways to make formal methods for HPC practical. In *Proceedings of FoSER'10*, pages 137–142. ACM, 2010.

[46] John A. Gosden. Explicit parallel processing description and control in programs for multi- and uni-processor computers. In *Proceedings of AFIPS '66 (Fall)*, pages 651–660, New York, NY, USA, 1966. ACM.

[47] Rajiv Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. *SIGARCH Computer Architecture News*, 17(2):54–63, 1989.

[48] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of LFP'84*, pages 9–17. ACM, 1984.

[49] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17:1–17, 1988.

[50] Tobias Hilbrich, Bronis R. de Supinski, Fabian Hänsel, Matthias S. Müller, Martin Schulz, and Wolfgang E. Nagel. Runtime MPI collective checking with tree-based overlay networks. In *Proceedings of EuroMPI'13*, pages 129–134. ACM, 2013.

[51] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. A graph based approach for MPI deadlock detection. In *Proceedings of ICS'09*, pages 296–305. ACM, 2009.

[52] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MPI runtime error detection with MUST: advances in deadlock detection. In *Proceedings of SC'12*, pages 1–11. IEEE, 2012.

[53] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[54] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. Fast barrier synchronization for infiniband. In *Proceedings of IPDPS'06*, pages 272–272. IEEE, 2006.

[55] Harry F. Jordan. A special purpose architecture for finite element analysis. In *Proceedings of ICPP'78*, pages 263–266, 1978.

[56] Harry F. Jordan, Muhammad S. Benten, Gita Alaghband, and Rüdiger Jakob. The FORCE: A highly portable parallel programming language. In *Proceedings of ICPP'89*, volume 2, pages 112–117, 1989.

[57] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of FSE'10*, pages 327–336. ACM, 2010.

[58] Edward G. Coffman Jr., M. J. Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[59] Inbum Jung, Jongwoong Hyun, Joonwon Lee, and Joongsoo Ma. Two-phase barrier: A synchronization primitive for improving the processor utilization. *International Journal of Parallel Programming*, 29:607–627, 2001.

[60] Amir Kamil and Katherine Yelick. Enforcing textual alignment of collectives using dynamic checks. In *Proceedings of LCPC'09*, volume 5898 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2009.

[61] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP'01*, volume 2072, pages 327–353. Springer, 2001.

[62] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Survey*, 19(4):303–328, 1987.

[63] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[64] Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and Matthias S. Müller. MPI correctness checking with Marmot. In *Proceedings of PTW'08*, pages 61–78. Springer, 2008.

[65] Ajay D. Kshemkalyani and Mukesh Singhal. Correct two-phase and one-phase deadlock detection algorithms for distributed systems. In *Proceedings of SPDP'90*, pages 126–129, 1990.

[66] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974.

[67] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[68] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[69] Duy-Khanh Le, Wei-Ngan Chin, and Yong-Meng Teo. Verification of static and dynamic barrier synchronization using bounded permissions. In *ICFEM'13*, volume 8144, pages 231–248, 2013.

[70] Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of PPoPP'10*, pages 25–36. ACM, 2010.

[71] Eugene Loh. The Ideal HPC Programming Language. *ACM Queue*, 8:30:30–30:38, 2010.

[72] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of ASPLOS'08*, pages 329–339. ACM, 2008.

[73] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[74] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Mi-Young Park, Elizabeth Kleiman, Olga Weiss, Andre Wehe, and Melissa Yahya. The importance of run-time error detection. In *Proceedings of Parallel Tools'09*, pages 145–155. Springer, 2010.

[75] Piotr R. Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of SC'06*. ACM, 2006.

[76] Eduardo R. B. Marques, Francisco Martins, Vasco Vasconcelos, Nicholas Ng, and Nuno Martins. Towards deductive verification of MPI programs against session types. In *Proceedings of PLACES'13*, volume 137 of *EPTCS*, pages 103–113. Open Publishing Association, 2013.

[77] Francisco Martins, Vasco T. Vasconcelos, and Tiago Cogumbreiro. Types for X10 Clocks. In *Post-proceedings of PLACES'10*, volume 69 of *EPTCS*, pages 111–129, 2011.

[78] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2014. Version 8.4pl4.

[79] John McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[80] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I/II. *Journal of Information and Computation*, 100:1–77, 1992.

[81] Matthew T. O'Keefe and Henry G. Dietz. Hardware barrier synchronization: Dynamic barrier MIMD (DBM). In *Proceedings of ICPP'90*, pages 43–46. Pennsylvania State University, 1990.

[82] Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of FSE'12*, pages 1–11. ACM, 2012.

[83] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, 2005.

[84] Ascher Opler. Procedure-oriented language statements to facilitate parallel processing. *Communications of the ACM*, 8(5):306–307, 1965.

[85] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Robert Palmer, Rajeev Thakur, and William Gropp. Practical model-checking method for verifying correctness of MPI programs. In *Proceedings of PVM/MPI'07*, volume 4757, pages 344–353. Springer, 2007.

[86] Benjamin C. Pierce. *Types and Programming Languages*. MIT, 2002.

[87] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–25, 2013.

[88] Swaroop Pophale, Oscar Hernandez, Stephen Poole, and Barbara Chapman. Static analysis for unaligned collective synchronization matching for OpenSHMEM. In *Proceedings of PGAS'13*, pages 231–236. The University of Edinburgh, 2013.

[89] Terrence W. Pratt. Pisces: An environment for parallel scientific computation. *IEEE Software*, 2(4):7–20, 1985.

[90] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, 1979.

[91] Indranil Roy, Glenn R. Luecke, James Coyle, and Marina Kraeva. A scalable deadlock detection algorithm for UPC collective operations. In *Proceedings of PGAS'13*, pages 2–15. The University of Edinburgh, 2013.

[92] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *Proceedings of CONCUR'05*, volume 3653 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2005.

[93] John Sartori and Rakesh Kumar. Low-overhead, high-speed multi-core barrier synchronization. In *Proceedings of HiPEAC'10*, pages 18–34. Springer, 2010.

[94] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of ICS'08*, pages 277–288. ACM, 2008.

[95] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *Proceedings of IPDPS'09*, pages 1–12. IEEE, 2009.

[96] Jun Shirako, David M. Peixotto, Dragoş-Dumitru Sbîrlea, and Vivek Sarkar. Phaser beams: Integrating stream parallelism with task parallelism. Presented at the X10'11, 2011.

[97] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free MPI programs for verification. In *Proceedings of PPoPP'05*, pages 95–106. ACM, 2005.

[98] Kenneth C. Smith, Alice Wang, and Laura C. Fujino. Through the looking glass II – part 1 of 2: Trend tracking for isscc 2013. *ISSC Magazine*, 5(1):71–89, 2013.

[99] Lorna A. Smith, J. Mark Bull, and Jan Obdrzálek. A parallel Java Grande benchmark suite. In *Proceedings of SC'01*. ACM, 2001.

[100] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[101] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[102] Franklyn Turbak. First-class synchronization barriers. In *Proceedings of ICFP'96*, pages 157–168. ACM, 1996.

[103] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[104] Nalini Vasudevan, Olivier Tardieu, Julian Dolby, and Stephen A. Edwards. Compile-time analysis and specialization of clocks in concurrent programs. In *Proceedings of CC'09*, pages 48–62. Springer, 2009.

[105] Anh Vo. *Scalable Formal Dynamic Verification of MPI Programs Through Distributed Causality Tracking*. PhD thesis, University of Utah, 2011. AAI3454168.

[106] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical MPI programs. In *Proceedings of PPoPP'09*, pages 261–270. ACM, 2009.

[107] Haitao Wei, Hong Tan, Xiaoxian Liu, and Junqing Yu. StreamX10: A stream programming framework on X10. In *Proceedings of X10'12*, pages 1–6. ACM, 2012.

[108] Michael Wolfe. Multiprocessor synchronization for concurrent loops. *IEEE Software*, 5(1):34–42, 1988.

[109] Course materials of *principles and practice of parallel programming*. `www.cs.columbia.edu/~martha/courses/4130/au13/`, 2013.

[110] Katherine Yelick, Paul Hilfinger, Susan Graham, Dan Bonachea, Jimmy Su, Amir Kamil, Kaushik Datta, Phillip Colella, and Tong Wen. Parallel languages and compilers: Perspective from the Titanium experience. *International Journal of High Performance Computing Applications*, 21(3):266–290, 2007.

[111] Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Proceedings of PPoPP'07*, pages 194–204. ACM, 2007.

[112] Yuan Zhang, Evelyn Duesterwald, and Guang R. Gao. Concurrency analysis for shared memory programs with textually unaligned barriers. In *Proceedings of LCPC'08*, volume 5234 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2008.

[113] Yingchun Zhu and Laurie J. Hendren. Communication optimizations for parallel C programs. *SIGPLAN Notices*, 33(5):199–211, 1998.

[114] Dieter Zöbel. The deadlock problem: a classifying bibliography. *SIGOPS Operating Systems Review*, 17(4):6–15, 1983.