ÆMINIUM
Freeing Programmers from the Shackles of Sequentiality

Sven Stork

19 March 2013
CMU-ISR-13-102

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
&
Department of Informatics Engineering
University of Coimbra
Polo II, 3030-199 Coimbra, Portugal

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This work was partially supported by the Portuguese Research Agency – FCT, through a scholarship (SFRH / BD / 33522 / 2008), CISUC (R&D Unit 326/97), and the CMU|Portugal program.

Keywords: access permissions, data groups, concurrency, programming language, dataflow

Abstract

The aim of this doctoral thesis was to study the implications of having a concurrent-by-default programming language. This includes language design, runtime system, performance and software engineering considerations.

We conduct our study through the design of the concurrent-by-default AEminium programming language. ÆMINIUM leverages the permission flow of object and group permissions through the program to validate the program's correctness and to automatically infer a possible parallelization strategy via a dataflow graph. ÆMINIUM supports regular parallelism (such as fork-join parallelism) as well as irregular parallelism (e.g., as dataflow).

In this thesis we present a formal system, called μÆMINIUM, modeling the core concepts of ÆMINIUM. μÆMINIUM static type system is based on "Featherweight Java" with ÆMINIUM specific extensions. Besides checking for correctness ÆMINIUM 's type system it also uses the permission flow to compute a potential parallel execution strategy for the program. μÆMINIUM 's dynamic semantics use a concurrent-by-default evaluation approach. Along with the formal system we present its soundness proof.

We provide a full description of the implementation along with the description of various optimization techniques we used. We implemented AEminium as an extension of the Plaid programming language, which has first-class support for permissions built-in. The ÆMINIUM implementation and all case studies are publicly available under the General Public License.

We use various case studies to evaluate ÆMINIUM's applicability and to demonstrate that ÆMINIUM parallelized code has some performance improvements compared to its sequential counterpart. We chose to use case studies of common domains or problems that are known to benefit from parallelization, to show that ÆMINIUM is powerful enough to encode them. We demonstrate through a webserver application, which evaluates ÆMINIUM 's impact on latency-bound applications, that AEminium can achieve a 70% performance improvement over the sequential counter part. In another case study we chose to implement a dictionary function to evaluate ÆMINIUM 's capabilities to express essential data structures. Our evaluation demonstrates that ÆMINIUM can be use to express parallelism in such data-structures and that the performance benefits scale with the amount of annotation effort which is put in to the implementation. We chose an integral computation example to evaluate pure functional programming and computational intensive use cases. Our experiments show that ÆMINIUM is capable of extracting parallelism from functional code and achieving

performance improvements up to the limits of Plaid inherent performance bounds. We elaborate about our experience in developing and using ÆMINIUM and discuss potential shortcomings and potential areas of interest for future work.

Overall, we hope that the work helps to advance concurrent programming in modern programming environments.

Resumo

O objetivo desta dissertação consistiu no estudo das implicações de construir uma linguagem de programação "concorrente por omissão". Tal inclui aspetos de desenho e arquitetura da linguagem, ambiente de execução, performance, usabilidade, assim como considerações de engenharia de software.

A pedra basilar do trabalho consiste na proposta e desenho de uma linguagem de programação cujas primitivas são concorrentes por omissão chamada ÆMINIUM. A linguagem ÆMINIUM utiliza o permissões sobre objetos e grupos de objetos para validar a correção de um programa assim como para inferir possíveis estratégias de paralisação através de um grafo de fluxo de dados. A linguagem ÆMINIUM suporta paralelismo regular (e.g., do tipo fork-join) e paralelismo irregular (e.g., do tipo dataflow).

Neste trabalho é apresentado um sistema formal para a linguagem, chamado μÆMINIUM, incluindo a sua semântica estática e dinâmica, assim como uma prova formal da correção do seu sistema de tipos. O sistema de tipos da μÆMINIUM é baseado em "Featherweight Java" com extensões específicas para Aeminium. Para além de verificar a correção do sistema de tipos, o sistema é também usado para calcular uma potencial estratégia de paralelização. A semântica dinâmica do μÆMINIUM utiliza sempre uma abordagem de avaliação concorrente.

É apresentada uma descrição completa da implementação da linguagem e do seu sistema de execução, assim como das várias optimizações que foram usadas. A linguagem ÆMINIUM foi implementada como uma extensão da linguagem Plaid que possui suporte explícito para permissões. Tanto a implementação da linguagem ÆMINIUM como de todos os casos de estudo encontram-se publicamente disponíveis.

Finalmente, são discutidos vários exemplos usados para validar a aplicabilidade da linguagem ÆMINIUM assim como para mostrar que os programas paralisados automaticamente pelo sistema possuem uma performance superior às suas versões sequenciais. Os casos de estudo foram escolhidos de domínios comuns e de domínios conhecidos como podendo beneficiar de paralelização. No caso de estudo da implementação de um servidor web a abordagem ÆMINIUM obtém melhorias de performance superiores a 70% quando comparado com o caso sequencial. Noutro caso de estudo é implementado uma estrutura de dados dicionário para avaliar a expressividade da linguagem. Os resultados mostram que os benefícios de performance obtidos ao usar a linguagem ÆMINIUM são proporcionais ao esforço despendido em anotar o programa original. É usado um caso de estudo de cálculo numérico de um integral para avaliar um modelo de programação puramente funcional e, simultaneamente, intensivo em termos de computação. As experiências mostram que o sistema

ÆMINIUM permite obter paralelismo de código funcional assim como obter ganhos de desempenho, sendo estes atualmente limitados apenas pelo ambiente de execução subjacente (Plaid). Finalmente, é discutida toda a abordagem, globalmente, assim como as grandes conclusões do desenvolvimento do sistema ÆMINIUM, as suas fraquezas e seu potencial. São ainda apresentadas áreas de possível trabalho futuro.

Com este trabalho esperamos ter contribuído para o avanço da área da programação concorrente em ambientes de desenvolvimento de software modernos.

Für meine Familie.

ACKNOWLEDGMENTS

There are many people to whom I owe gratitude and I apologize in advance to everyone I forgot.

I have to thank my advisors, Paulo Marques and Jonathan Aldrich, for their support and advice, and especially for suffering with me through what Paulo labelled the most complicated PhD he has ever seen. I need to thank Paulo in particular for helping me out when it came to dealing with the Portuguese and Coimbra administration nightmare. I would also like to thank my committee, William Scherlis, Ernesto Costa, Todd Mowry, Marco Vieira, for their support and the constructive feedback they gave me.

I would thank Kevin Bierhoff, Ciera Jaspan and Nels Beckman for passing on the wisdom of previous generations to me. It was most helpful in streamlining many aspects of my PhD and allowed me more than one shortcut. Thanks to Karl Naden and Joshua Sunshine for sufferring with me through years of Plaid development and for the endless banter during our meetings. I also thank Jeff Barnes, Vishal Dwivedi, Thomas LaToza, Greg Hartman, Marwan Abi-Antoun, Dean Sutherland and the rest of the Plaid group and CS students for their help and advice, as well as for countless lunches and discussions.

I have to thank Marcelo Mendes and Ivano Irrera for constantly reminding me that there is a life outside of the PhD and forcing me to take a break every once in a while. I owe my thanks to Koichiro Suzuki and Yulia Zhukoff for introducing me to the wonderful world of Tango and helping to keep me sane during the last year of my PhD. I also like to thank Meghan Tighe, George Mangieri, Rafael Herrera and the rest of the Pittsburgh Tango community for countless Milongas and all the fun we had. Last but not least, I have to thank my mother, Renate Stork, my sister, Daniela Stork, and my brother, Markus Stork, for their support and for understanding that I had to disappear for +5 years.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

One of the most fundamental technology shifts in the last few decades is best characterized by "The free lunch is over" [86]. Because it is becoming harder and harder to improve single-CPU performance, hardware vendors started to integrate multiple cores into single chip. This means that programmers need to develop concurrent applications if they want to achieve performance improvements on new hardware. Writing concurrent applications is notoriously complicated and error-prone, because concurrent tasks must be coordinated to avoid problems like race conditions or deadlocks.

Pure functional programming is by nature an excellent fit for concurrent programming. In functional programming there are no side effects and programs can execute concurrently to the extent permitted by data dependencies. Although functional programming can solve every problem, having explicit state, as provided by imperative languages, allows the developer to express certain problems in a more intuitive and efficient way. In an ideal world we would like to combine the concurrent execution benefits of functional programming with the expressiveness of an imperative object-oriented language.

Sharing state between concurrent tasks immediately raises questions like: *'In which order should those accesses occur?'* and *'How can one coordinate those accesses to maintain a program invariants?'* The reason why those questions are hard to answer is because there are *implicit* dependencies between code and state. Methods can arbitrarily change any accessible state without revealing this information to the caller. This means that two methods can be dependent on the same state, without the caller knowing about it. Because of this lack of information, current programming languages use the order in which code is written as a proxy to express those implicit dependencies. Therefore the compiler mostly has to follow the given order and cannot exploit potential concurrency automatically. When developer add concurrency manually, it is easy for her to miss important dependencies, introducing race conditions and other defects.

To overcome this situation, we propose to transform *implicit* dependencies into *explicit* dependencies and then infer the ordering constraints automatically. In our proposed system, by default, everything is concurrent unless explicit dependencies imply a specific ordering. By using a concurrent by default approach, we eliminate explicit, notoriously complicated and error prone, reasoning about sequential and parallel ordering. Instead of specifying when and where which operations should be executed, in our approach the programmer specifies which stateful effects[1] each operation performs. The system uses that dependency information to perform the operations in an non-interfering manner. The system will not only use the dependency information to perform concurrent execution but also validate that the dependency information is consistent.

[1]E.g. reading a certain memory region, updating a specific memory region, etc.

1.1 Approach

We propose to use *access permissions* [21] to specify explicit dependencies between stateful operations. Access permissions are abstract capabilities that grant or prohibit certain kinds of accesses to specific state. Our approach requires each method to specify a permission to all directly accessed state. Transitively accessed state is handled automatically through embedded permission information. Looked at from a slightly different perspective, our system ensures that every method only accesses state for which it has explicit permissions. The way we use access permissions to specify state dependencies resembles the way Haskell [58] uses its I/O monad[2] to specify access to global state. But unlike the I/O monad, which provides just one permission to all the state in the system, access permissions allow greater flexibility by supporting fine-grained specifications, describing the exact state and permitted operations on it.

To enhance support for shared data we propose to use the combination of *data groups* [62] and *data group permissions* to enable user-defined granularity control and the avoidance of unnecessary synchronization. Data groups represent an abstract collection of objects which form a disjoint partitioning of the heap. Data group permissions represent aliasing and protection information about data groups.

The goal of this dissertation is to show that the concurrent-by-default paradigm is an feasible and useful approach. Therefore we are going to design an new programming language, called ÆMINIUM, and runtime system, which takes concurrency-by-default as one of its main design principles. Achieving this goal implies language design, development of the runtime system, performance evaluations and case studies.

1.2 Example: Proposed Approach

To illustrate these concepts, consider the `transfer` function shown below, which transfers a specific amount between two bank accounts. It first withdraws the specified amount of money from the 'from' account and then deposits the same amount into the 'to' account.

Listing 1.1: Transfer Code

```
public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
    withdraw(from, amount)
    deposit(to, amount);
}
```

[2]Think of it as one global permission, which grants the right to access or change all state in the system.

For this example we assume that the order in which we perform the withdraw and deposit operations does not matter. In particular, they could be executed concurrently because both the withdraw and deposit operations should only affect the specified bank account and no other. To encode this extra information ÆMINIUM uses permission annotations. Permissions [32] specify aliasing and access information for objects. The `transfer` method specifies that it requires a *unique* permission to both bank accounts and an *immutable* permission to the amount parameter. The unique permission means that there is only one valid reference to the specified object in the whole system at the moment of a function call, and modifications to the object within the function are possible. The immutable permission specifies that there might be multiple aliases to this object but none of them can be used to change the object.

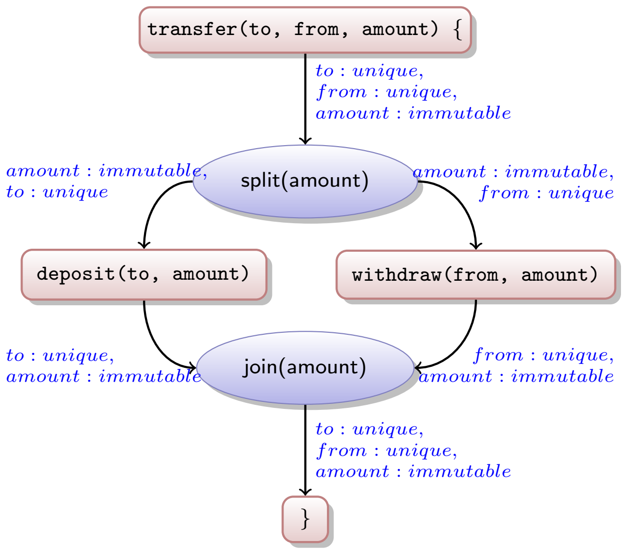
Assuming the method declarations for the `deposit` and `withdraw` methods given below, ÆMINIUM is now able to compute the permission flow within the `transfer` method. The unique permission of the 'to' parameter flows to the `deposit` method while the unique permission of the 'from' parameter flows to the `withdraw`. But we only have an immutable permission to the 'amount' object while both `withdraw` and `deposit` require one each. Because immutable permissions explicitly allow aliasing ÆMINIUM automatically *splits* the one immutable permission into two permissions, which are then passed to the two method calls:

Listing 1.2: Prototypes for `withdraw` and `deposit` functions

```
public void withdraw(unique Account account,
                     immutable Amount amount) {...}

public void deposit(unique Account account,
                    immutable Amount amount) {...}
```

The permission flow of the `transfer` method is shown in Figure 1.1. After the split operation the *unique* 'to' and *immutable* 'amount' permissions are passed to `deposit` method while the unique 'from' permission and immutable 'amount' permission flow to the `withdraw` method. After those methods complete ÆMINIUM will automatically *join* the previously split immutable permissions. The permission flow graph corresponds to the data flow graph which is used to execute the `transfer` methods. Although this example illustrates only unique and immutable data, we will later show how ÆMINIUM supports shared mutable data with *shared* permissions and an `atomic` synchronization primitive.

Figure 1.1: Permission Flow in the Transfer Example. We use the notation $var : perm$ to indicate that we have permission 'perm' for variable 'var'.


transfer(to, from, amount) {
to : unique, from : unique, amount : immutable
split(amount)
amount : immutable, to : unique
amount : immutable, from : unique
deposit(to, amount)
withdraw(from, amount)
to : unique, amount : immutable
from : unique, amount : immutable
join(amount)
to : unique, from : unique, amount : immutable
}

1.3 Thesis Statement

Following a concurrent-by-default philosophy we leverage the flow of access permissions (static alias descriptions of program references, cf Section 3) and data group permissions (static access privileges to data groups, cf Section 3) through a program to automatically parallelize its execution in a safe way at runtime. This leads us to define the following thesis statement:

The flow of access- and group-permissions provides a powerful abstraction to capture common programming idioms while simultaneous enabling the safe extraction of efficient concurrency.

We break the thesis statement down into more concrete and measurable hypotheses:

1.3.1 Hypothesis: Safety

To ensure safety we can develop and formalize an analysis that uses flow of access permission and data group permission throughout a program to compute data dependencies, which allow the concurrent execution of the program while guaranteeing the absence of race conditions.

Validation

This hypothesis will be validated by developing and formalizing a type system and operational semantics based on our permission system and formally proving the type system sound with respect to its semantics. The proof essentially says that if we parallelize the program based on the flow of permissions and our rules (e.g., simultaneous access to immutable data is safe) then there cannot get a race condition at runtime. We do not include deadlock freedom in our definition of safety because prior work already demonstrated possible solutions for either a static [59] or dynamic [60] deadlock detection and for simplicity reasons we want to focus on the concurrent-by-default aspect of our system.

1.3.2 Hypothesis: Efficiency

Programs written in our concurrent-by-default approach can be parallelized automatically, achieving a better performance than their sequential counterparts (provided sufficient availability of parallelism in the application itself).

Validation

We validate this hypothesis by performing several case studies showing that applications written in our concurrent-by-default style can achieve performance improvements through automatically parallelization over their sequential counterparts.

1.3.3 Hypothesis: Practicality

We claim that access permissions and data group permissions provide a powerful abstraction for concurrent-by-default applications, which allows common program idioms to be captured.

Validation

This hypothesis is validated by performing several case studies modeling common program patterns can be expressed in the concurrent-by-default ÆMINIUM approach while achieving performance improvements compared to the sequential counterparts. Furthermore we will report the difficulties and issues encountered while developing those case studies. While the current system not fully practical for every day development, we have have shown that the approach is doable with a small number of annotations written by the programmer, being this just a first step towards practicality.

1.4 Contributions

The overall contribution of this dissertation is the examination of using access permission and data groups to automatic parallelize programs for general purpose programming languages. The main contributions are:

Core Calculus The first contribution consists of the development of μÆMINIUM, a sound core-calculus modeling a concurrent-by-default programming languages based on permissions (cf. 4). μÆMINIUM allowed us to study the core principles of ÆMINIUM and consists of a static type checking rules and a concurrent-by-default small-step evaluation semantics.

Proof of Soundness We proofed the soundness of our μÆMINIUM core-calculus stating that all programs which type check are free of data races (cf. Appendix A).

Implementation We developed a prototype implementation of ÆMINIUM, including the static type checking, the dependency computation and the concurrent-by-default execution (cf. Chapter 5)

Evaluation We evaluated our ÆMINIUM implementation through conducting several case studies to validate our claims. We also report on our experience use ÆMINIUM to develop those case studies (cf. Chapter 6).

2 State of the Art

This chapter provides an overview of the current state of the art in the area of concurrent programming. There are many different and often orthogonal concepts and principles in the area of concurrent programming. Since our system is focused on implicit concurrency this dimension is followed when presenting related work. We first present explicit concurrency approaches for concurrent programming along with their advantages and disadvantages. Then we present implicit concurrency approaches for concurrent programming, again with their advantages and disadvantage. Given the huge number of sometimes just marginally-different approaches and systems, we are focusing on general concepts and the most closely-related research. Also there is no strict borderline between implicit and explicit concurrency, but we are going to use the following definitions:

explicit In an explicit concurrency system, the programmer is actively involved in the creation and management of concurrent execution. This means in particular the programmer writes *explicit* code[1] to create or manage concurrent tasks (e.g creating threads, task pools, . . .) and coordinate synchronisation (e.g., locks, conditional variables, etc).

implicit An implicit concurrency system does not require the user to actively write code for concurrent execution. In an implicit system the semantics of the language or library interfaces imply that certain operations can be performed concurrently without user intervention.

2.1 Explicit Concurrency

Explicit concurrency is all about the manual management of different threads of execution. The most simple and most coarse grain form of explicit concurrency is separate, sequential processes which exchange data via a communication channel. A minimalistic formal description of those *communicating sequential processes* (CSP) is given by [33]. It is the common case for CSP to communicate via message passing. In message passing all necessary synchronization is implicitly handled by the message-passing abstraction and relieves the programmer from explicitly managing synchronization via low-level primitives. Because processes have strong isolation between each other, data needs to be copied from one process to another. This leads to message passing systems being in general free of low-level race-conditions (i.e., when the output of a system unexpectedly depends on the timing of other events), but also contributes to its inefficiency when a large amount of data needs to be exchanged. In general the support for spawning new processes is not directly

[1]Often code that uses low-level abstractions of operating or hardware features.

included in mainstream programming languages and is rather provided via libraries. Those libraries range from simple wrappers that call straight into the operating system (e.g., fork), to highly-sophisticated libraries that support complex communication operations and provide infrastructure management support (e.g., MPI [67, 68]).

The *Message Passing Interface* (MPI) [67, 68] is one example of such an sophisticated library. MPI is the established de-facto standard for developing high-performance distributed memory applications. MPI implementations provide, besides the library itself, several tools for starting and managing multiple jobs. The latest MPI standard [68] also added support for dynamically creating processes. The MPI standard defines a rich set of communication abstractions, ranging from synchronous and asynchronous point-to-point operations to complex collective operations (e.g., a collective reduce operation with user-defined data types and operator functions).

Erlang [16] is one of the few programming languages that has built-in support for process creation and communication channels between processes. Erlang processes do not map directly to operating system processes. Erlang provides a strong isolation guarantee between processes and a high-level communication abstraction, which allows processes to run either on the same virtual machine or on different virtual machines on different nodes. Therefore we consider Erlang a member of the CSP family and it inherits all the corresponding features and shortcomings.

Threads are concurrent entities inside a process that share the address space with their host process. Therefore, threads allow fast and easy shared memory communication. Instead of sending data between processes, and eventually duplicating shared data, data can be uniformly accessed by all threads in the system. One side effect is that all accesses to shared data needs to be coordinated to avoid *race conditions*. Similar to process management, many older programming languages (like, for instance, C, C++[2], . . .) support threads via external libraries, providing simple wrappers around operating system functionality. As shown in [29], if threads are not part of the programming language, the compiler can generate incorrect code while optimizing the program. Therefore many modern programming languages support threads at a language level and provide explicit descriptions of the memory model used [64].

Mutexes and *semaphores* are the most commonly used and supported synchronization primitives when it comes to protecting access to shared resources. The usage of those low-level primitives is notoriously complicated and error-prone. Several different static verification mechanism have been proposed to verify the correct usage of those locking primitives. We present two example systems which relate closest to our research. Terauchi [87] describes a type-system for generating a linear equation problem based on its input program. The linear equations are constructed in a way that if the linear equations are solvable if and only if the corresponding program is guaranteed to be free of race-conditions. In [89] a concurrent extension of

[2]The new C++11 standard has a thread and memory model defined in: `http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/`

typestate [84] is described to detect race-conditions. While protected through the correct lock[3] the system will follow the normal typestate protocol and, as soon as the lock is released, all typestate information is immediately forgotten. Therefore if the critical zone is not wide enough to protect all critical accesses, the system will not be able to establish the typestate conditions and will trigger an error. All those systems either require a whole program analysis or extensive annotations, and therefore have limitations with regard to practical usage.

Transaction Memory (TM) [60], which avoids the explicit reasoning about which lock protects which shared state, became a very active research area during the past few years. In TM the programmer indicates, by using an atomic-block, which code should run as if it would be one atomic-instruction. Like an atomic instruction, either the whole execution successfully completes and the rest of the system can see the changes or no changes are performed at all. The underlying runtime-system will automatically take care of protecting access to shared resources, detecting possible conflicts and resolving them. We consider TM more an implicit than an explicit concurrency control mechanism because the programmer does not specify which lock protects which data, he rather declares which piece of code should required to be run under atomic conditions.

Cilk [26] is a programming language which includes higher-level, but still explicit, concurrency abstractions (i.e., language constructs which do not necessarily represent hardware or OS entities). Cilk extends the C programming language with three new keywords: `cilk` (to mark spawn-able functions), `spawn` (to spawn function calls asynchronously) and `sync` (to wait for completion of previously started asynchronous functions). Figure 2.1 shows a simple Cilk program for concurrently computing a Fibonacci number. Although Cilk simplifies the management of concurrent tasks, it still relies on the programmer to explicitly specify where and how to extract concurrency and how to correctly synchronize access to shared resources. Cheng [38] describes a mechanism to check for race freedom when locks are used for protecting access to shared resources. The proposed approach is not a general verification tool, but rather debugging tool for checking the absence of race conditions for one specific input by sequentializing the execution of a Cilk program. Besides language-based, higher-level abstractions, one of the major contributions of Cilk is its very efficient runtime-system [45], which employs a work-stealing approach for load-balancing.

Kilim [82] is an actor-based programming language for shared memory systems. In Kilim actors run concurrently inside the same process and communicate via message passing. Similar to the Microsoft Singularity operating system [55], Kilim uses statically verified ownership transfer between actors to avoid expensive data copy operations. Therefore Kilim merges the implicit synchronization of message passing with the performance associated with shared memory communication. But the programmer is still in charge of specifying the concurrency and needs to map the concrete problem to an actor model [52].

[3]If the correct lock is not manually specified by the user the system employs heuristics to guess the correct lock automatically.

Listing 2.1: A Cilk example program to concurrently compute a Fibonacci number. The `main` and all spawn-able functions must be marked with the `cilk` specifier. In line 21 an asynchronous computation is started, indicated by the `spwan` keyword. With the `sync` keyword the program waits for the completion of this task. While asynchronously executing the `fib` function, it recursively spawns off new asynch. tasks (line 11 and 12), and waits in the following line via the `sync` keyword for their completion.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <cilk.h>
4
5  cilk int fib (int n)
6  {
7      if (n<2) {
8        return n;
9      } else {
10         int x, y;
11         x = spawn fib (n1);
12         y = spawn fib (n2);
13         sync;
14         return (x+y);
15     }
16 }
17
18 cilk int main (int argc, char *argv ])
19 {
20     int result;
21     result = spawn fib(atoi(argv[1]));
22     sync;
23     printf (”Result:␣%d\n”, result);
24     return 0;
25 }
```

Axum[4] [69], similar to Kilim, is an actor-based programming language. But unlike Kilim, which mainly supports mail boxes as a communication primitive, Axum provides an extensive set of operators to build dataflow graphs. To achieve strong isolation between actors, all data that is sent must be serialized before the send operation and de-serialized after its reception. Axum supports the specification of communication protocols for specific channels, which allows the detection of certain deadlock scenarios by detecting a protocol violation. To avoid the expensive data copy operations that are involved in the message passing, Axum supports shared state via ’domains’. Domains are groups of data/objects that are shared between several actors. Each actor has to specify if is just reading or also writing to the mutable state of the corresponding domain. This allows the system to order accesses to mutable state and avoid race conditions. This approach can be seen as a very simple form of group permissions to domains (to be discussed later in Section 3).

[4]Formerly known as Maestro.

Dryad [57] is a runtime-system for execution of dataflow graphs. Dryad allows the execution of programs on platforms ranging from multi-core CPUs to big computer clusters. Dryad is mainly used as backend execution engine for high-level concurrent programming languages. For instance DryadLINQ [90], a LINQ[5] implementation, and SCOPE [35], a high-level data processing language, use Dryad as (one of) their backend execution engines.

X10 [37] is an new programming language that has been developed in a DARPA-funded supercomputing initiative. It aims at the creation of a next generation programming language for high-performance computing. One of the major design-goals of X10 was the support for distributed computing. X10 uses a global partitioned address space, where the distinct partitions are called places, to take the *Non-Uniform Memory Access* (NUMA) of current systems into account. X10 allows the programmer to start asynchronous activities in other places to modify or fetch data from places. Asynchronous activities can be coordinated via barrier-like objects, called clocks, which allow the execution of activities based on phases. X10 also provides an atomic-block for protecting access to shared resources.

MapReduce [42] is a simple computational model in which the input data is *mapped* into smaller intermediate result data which is then *reduced* to the final result. The mapping and (partially) the reduction can be parallelized if the mapping function supports mapping of a partial set of input data to a partial set of intermediate result data. MapReduce work very well for embarrassingly parallel problems (such as counting occurrences of specific data in the input data).

SharC [15] is a data race checker for *C* programs. SharC uses lightweight type annotation system which bares some resemblance to ÆMINIUM's permission and data group approach. SharC has *private* and *readonly* annotations which compare to ÆMINIUM's *unique* and *immtable* permissions. In SharC, all shared data accesses need to be marked with an *locked(lock)* indicating which lock needs to be held before accessing the corresponding data. This resembles ÆMINIUM's shared permissions associated with data groups. To allow for more flexibility, SharC uses on top of a static typesystem additionally a dynamic runtime checks. Unlike ÆMINIUM, SharC is a checker only and can only check that a user parallelized program is accessing its state in a safe manner.

SvS (Synchronization via Scheduling) [20] optimizes task graph executions by avoiding synchronization operations on shared data. SvS utilizes a static analysis for finding dependencies between tasks. To overcome the conservative approximations of the static analysis SvS uses additionally a dynamic analysis to refine those task dependences and only execute task in parallel which read/write sets do not overlap.

[5]Discussed in section 2.2.

2.2 Implicit Concurrency

One of the main characteristics of implicit concurrency systems is their *declarative* nature. Instead of specifying *how* to do something, the programmer rather specifies *what* should be done and lets the system decide how to do it. Therefore implicit concurrency systems relieve the programmer from the complex specification of and reasoning about concurrent execution.

Pure functional programming [58] provides a good match for implicit concurrency. The lack of side-effects and the explicit dependencies inside the code allow the runtime-system/compiler to extract high levels of concurrency. As previously mentioned, pure functional programming is not suitable for all cases (e.g., high productivity and ease of use). Therefore functional programming languages increase their features by allowing mutable state and side effects. When it comes to mutual state and side effects, *Haskell* [58] has one of the most interesting approaches in dealing with those. In Haskell all side effects, namely changes to mutable state, must be explicitly mentioned in the function signature. For instance, a function that needs to perform I/O must declare that it requires an I/O monad. The I/O monad is a permission to change the 'world' and everything in it. The flow of the I/O monad is used by Haskell to sequentialize the execution of all methods that change mutable state and therefore avoid race-conditions. Having just one permission for the whole system is rather limiting and leads to a major bottleneck in highly-concurrent systems.

HPF [63], Nesl [22, 23] and ZPL [43] are examples of data-parallel programming languages. In these languages the programmer works mainly with arrays and the application of functions to the all or just part of the array elements. Those programming languages naturally fit to scientific computing, which mainly involves computation on huge datasets. General purpose programs, like for instance a web server, are hard to realize in such programming languages.

OpenMP [36, 75] is an industry standard for shared-memory programming in C and Fortran. OpenMP specifies transparent annotations that allow the compiler to automatically parallelize the program. OpenMP supports parallelization of non-regular code via parallel sections and tasks, but the main focus of OpenMP is the parallelization of regular problems that are expressed via loops. The goal of OpenMP annotations is not to tell the compiler how to parallelize the code, but rather to point the compiler to the code-fragments that make most sense to parallelize. Having said that, the programmer still has a fair amount of liberty on controlling how it's done. In Figure 2.2 a simple OpenMP program for the concurrent computation of matrix-multiplication is shown. Similar to data-parallel programming languages, OpenMP is a natural fit for scientific computing but has several shortcomings when it comes general purpose programming (e.g., limited support for expressing parallelism in irregular structures). Intel developed an OpenMP version for parallelizing programs across multiple machines called *Cluster OpenMP* [54]. Given the higher overhead of the underlying *distributed shared memory* (DSM) model, this approach seems rather inefficient for most cases.

Listing 2.2: A OpenMP example program that performs a parallel matrix multiplication. After allocating memory and initializing the matrices (line 28, code omitted for brevity) the program calls the `matrix_mult` function to perform the matrix multiplication. The `matrix_mult` implements a standard matrix multiplication algorithm, consisting of three nested loops. The pragma in line 7 tells an OpenMP-capable compiler, that the following (the most outer) for loop should be automatically parallelized.

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <assert.h>
 4
 5  static void matrix_mult(double *A, double *B, double *C, int size)
 6  {
 7  #pragma omp parallel for
 8      for ( int i = 0; i < size; i++ ) {
 9          for ( int k = 0 ; k < size ; k++ ) {
10              for ( int j = 0; j < size ; j++ ) {
11                  C[i+j*size] += A[i+k*size] + B[k+j*size];
12              }
13          }
14      }
15  }
16
17  int main(int argc, char *argv[])
18  {
19      int size = 0;
20      double *A = NULL, *B = NULL, *C = NULL;
21
22      size = atoi(argv[1]);
23
24      A = (double*)malloc(size*size*sizeof(double));
25      B = (double*)malloc(size*size*sizeof(double));
26      C = (double*)malloc(size*size*sizeof(double));
27
28      ...
29
30      matrix_mult(A, B, C, size);
31
32      free(A);free(B);free(C);
33      return 0;
34  }
```

Language Integrated Query (LINQ, [66]) is an extension to the C# programming language, which allows *Structured Query Language* (SQL, [3])-like operations on data objects. Any object that implements IEnumerable$\langle T \rangle$ can the used as source in an LINQ query. This allows LINQ to work with a variety of data objects, ranging from simple arrays, over complex collection objects to objects that represent remote databases. The high-level declarative nature of SQL is used inside databases for various optimizations,

Listing 2.3: A simple PLINQ program for finding all people of the given name that are over 21. Note that the only difference to a normal (sequential) LINQ programs is in line 26, where the program uses the `AsParallel` method to retrieve a parallel collection.

```
1  using System.Collections.Generic;
2  using System.Linq;
3
4  class Person {
5      public int id;
6      public string name;
7      public int age;
8
9      public override string ToString() {
10         return ”[” + id + ”]␣>␣” + name + ”(” + age + ”)”;
11     }
12 };
13
14 public class Simple
15 {
16     public static int Main(string[] args) {
17
18         var objs = new List<Person> {
19             new Person {id=1, name=”Hans”, age=12},
20             new Person {id=2, name=”Willi”, age=45},
21             new Person {id=3, name=”Gustav”, age=34},
22             new Person {id=6, name=”Hans”, age=67},
23             new Person {id=11, name=”Willi”, age=100},
24         };
25
26         var result = from o in objs.AsParallel()
27                         where o.name == args[0] && o.age >= 21
28                         orderby o.age ascending
29                         select new {o.name, o.age};
30
31         foreach(var o in result) {
32             System.Console.WriteLine(o);
33          }
34         return 0;
35     }
36 };
```

including parallel execution. With *Parallel LINQ* (PLINQ) [44] the same idea is transformed to LINQ. The PLINQ extension allows queries to be executed in parallel, as long as there are no data dependencies between the different computations. Figure 2.3 shows a simple example for concurrently filtering a list of people that match certain criteria. It is worth to mention that most of (P)LINQ is implemented as library. The main languages changes for supporting (P)LINQ have been the introduction of lambda functions and some syntactic sugar to write the queries in a more SQL-style way.

Fortess [14] is one of the most closely related projects to our approach. Fortress has been funded by DARPA for high-performance computing. The syntax of Fortress closely resembles Java's syntax, but Fortress employs a significantly different evaluation semantics. In Fortress many evaluation contexts, like for instance tuples and for-loops, execute concurrently by default. In the possible case of data races, the programmer either has to force a sequential execution or protect critical accesses with an atomic block. Fortress does not support any mechanism to detect possible data races. It is the responsibility of the programmer to localize potential data-races and take appropriate countermeasures. A very useful feature of Fortress is the usage of UNICODE symbols (e.g the sum or integral symbol), to render formulas and program code in 'pseudo-code' style format. This feature explicitly aims at the target user group of scientist who have a solid understanding of their domain, but might have limited programming skills.

Several *Automatic Parallelization* approaches and techniques for compilers have been proposed. In general these approaches focus on instruction level parallelism (ILP) by exploiting special vector units or improving pipeline utilization. Nowadays all mainstream compilers [1, 2, 4, 5] support ILP at different levels. While these approaches improve single threaded performance, they do not parallelize the program across multiple CPU cores. Therefore more coarse grain approaches for the automatically parallelization of programs have been investigated. Hall et al. [51] describe SUIF, an automatically parallelizing compiler for coarse grain parallelism. SUIF uses a scalar, an array and an inter-procedural analysis to automatically parallelize loops. A similar approach is used by T-Systems Cell-Compiler [9]. The T-System Cell-Compiler automatically extracts parallelism at the loop level to execute on a Cell processor [88]. Because both approaches rely on highly regular problems, they are a good fit for scientific computing but have limited applicability for irregular, general purpose programs.

Deterministic Parallel Java (DPJ, [27, 28]) an extension to Java with parallel `for` loops and blocks. It therefore has quite a resemblance to Open MP but instead using annotation the user needs to explicitly use new language constructs. But unlike Open MP, DPJ is able to statically check correctness through an effect system. In the initial version DPJ [27] only supported deterministic computations, which means that the type system checked that all possible parallel computations operate on disjoint sets of data. DPJ was later extended to support shared state and therefore non-determinism [28]. DPJ has many similarities to ÆMINIUM, especially to ÆMINIUM's data groups and the fork-join approach they are handled. The main differences of DPJ compared to ÆMINIUM are the lack of true data flow parallelism and the usage of effects system instead of permissions.

Craik et al. [40] describe a system which uses ownership information to automatically parallelize code in a dataflow style. Craik's ownership contexts are similar to ÆMINIUM's data groups, but they do not have the concept of *unique* or *immutable* permissions. Their system supports only deterministic parallelism.

While they provide an argument for soundness, our formal model goes further in incorporating a small-step operational semantics model of parallelism and a rigorous progress/preservation proof approach.

Pig latin [74] is a high level data processing language that supports various backends. Conceptually Pig latin aims for the sweet spot between the declarative stye of SQL and the procedural style of MapReduce. Pig latin allows the composition of complex data flow graphs through the composition of single data transformations.

2.3 Background

Access Permissions are a novel abstraction mechanism encoding information regarding how the referenced object can by used through the current reference as well as through possible other reference in the system (i.e., it encodes effect information and alias information). Access permissions are an extension to Fractional Permission [32], which have been introduced 2003 to check for race conditions. Boyland's fractional permissions included *unique*, *immutable* and *shared*. In 2007 Kevin Bierhoff [21] added *pure* and *full* permissions to better support verify typestate information in sequential programs. In 2008 Nels Beckman [18] extended Bierhoff's work to verify typestate in concurrent programs. Our work represents the next logical step. After using access permission to check parallel programs for correctness, we use access permissions to infer potential parallelism in programs.

Regions have originally been proposed 1999 by Aaron Greenhouse [50]. Regions are encapsulations of mutuable state. Greenhouse used regions and an effect system to determine whether there exists data dependencies between two computations or not. In 2002 Leino proposed and extension to their in 1998 proposed data groups [62] allowing for more precise side effect analysis. The main difference between data groups and regions is that a field can only be associated with one region but can be part of many data groups. In ÆMINIUM we use data groups to group objects (i.e., all fields of an object are associated with the same data group). Despite the fact that we do not allow an object (and therefore its fields) to be associated with multiple data groups we still called it a data group, rather than a regions, for semantic reasons.

Ownership, in particular ownership types, have been proposed in 1998 by Clarke [39] and provide a statically enforceable object-level encapsulation. Using ownership and strong encapsulation allows reasoning about disjointness of different objects. ÆMINIUM supports ownership and encapsulation through data groups which are only accessible inside object methods.

2.4 Summary

Table 2.1 provides a compact overview of the discussed systems. The overview shows for each system if the system is either implicit or explicit concurrent. It furthermore describes the generality of each system

Table 2.1: System Comparision

System	Style	Generality	Safety	Modularity
Threads	explicit	general purpose	none	-
Locks/Mutexes	explicit	general purpose	none	re-entrant locks
MPI	explicit	general purpose	isolation	-
Erlang	explicit	general purpose	isolation	-
Cilk	explicit	fork/join parallelism	none	-
Kilim	explicit	general purpose	isolation	-
Axum	explicit	general purpose	isolation	-
Dryad	explicit	data flow	none	-
X10	explicit	general purpose	isolation	-
MapReduce	explicit	data parallelism	isolation	-
SharC	explicit	general purpose	no data races	effects annotation
SvS	explicit	task graphs	no data races	effects annotation
Functional Programming	implicit	functional programming	type safety	monads
HPF/Nesl/ZPL	implicit	data parallelism	type safety	-
Open MP	semi-implicit	data parallelism	none	-
LINQ	implicit	data parallelism	none	-
Fortress	implicit	data parallelism	none	-
Parallelizing Compilers	implicit	data parallelism	none	-
DPJ	implicit	fork/join parallelism	type safety	effects annotations
Craik's System	implicit	data parallelism	type safety	-
Pig latin	implicit	data parallelism	none	-

differentiate systems which are specifically designed for some specific use cases and systems which are designed to work with any problem space. We also analyze the systems which safety guarantees they make with regards to the parallelisms. In particular we differentiate three kinds if safety classes: system which provide none safety guarantees, systems which mainly rely on isolation to avoid race conditions and systems in which the type system guarantees that no data races can occur. Lastly we look add concurrency specific support for modularity for each system.

When looking at this summary we can make two observations. The first observation is that explicit parallel systems seem to be more generic applicable than implicit systems. This observations makes sense because in many explicit approaches the programmer has a certain freedom to use the available parallel constructs in any way they want while in implicit parallel systems the availability/usage of parallel features is limited or automated. The limitation in implicit systems is necessary to allow for automatic parallelization.

Another observation which can be drawn from the overview is that implicit parallel systems seem to offer more safety guarantees than explicit systems. This makes sense as in implicit parallel systems the compiler/runtime performs many parts of the parallelization automatically with little to no user intervention.

To make sure that the parallelization is actually correct many implicit systems limit the ways code can be parallelized and have some sort of checking that the parallelization can be performed safely.

Those observations lead to the question how to write parallel code in the medium to long term future? Manually managing parallelism becomes infeasible with continuously growing systems. Implicit parallel systems relieve the programmer from some of those burdens but are often limited to domain specific areas. There seems to be a trend towards *Domain Specific Languages (DSL)* in general and for parallel computing [34] in particular. A question left open is if there is way to have a generic enough approach which captures most of the common parallel programming paradigms that frees programmer from the burdens of low-level parallel reasoning while still maintaining . We claim that using the permission flow of programs we can achieve such a system. Permissions have been designed with modularity in mind. Also not part of this dissertation permission permission seem to provide a good abstraction for embedding DSL in safe and predictable manner.

Proposed Approach

In this section we provide a high level overview of the ÆMINIUM approach and describe the ÆMINIUM programming language, which realizes a concurrent-by-default programming model [83] with a concrete design and precise semantics. ÆMINIUM uses *access permissions* [18] and group permissions for *data groups* [62] to compute the permission flow throughout the code (explained in the next sub-sections). The compiler uses this information to compute a data flow graph, which can then be executed in parallel on available computing resources.

While the general ÆMINIUM approach is language agnostic, we use an extended Java syntax for presenting the examples in this section. This requires extending the Java syntax to the missing language constructs and permissions annotations. We implemented a working prototype implementation in the Plaid [13] language. Plaid has permissions built-in as an first class language construct and therefore requires only minor extensions to support ÆMINIUM.

3.1 Access Permissions

Access Permissions (AP) combine alias and effects information for object references and have been studied in the past for checking interface protocol compliance and verifying the correct use of synchronization [18]. In ÆMINIUM we use access permissions, and more precisely the flow of the access permissions through the application, to model possible concurrent execution strategies for a program. Access permissions are abstract capabilities associated with object references. The primary purpose of access permissions is to keep track of how many references to a given object exist in a moment in time, and to specify what kind of operations are permitted on the object at that moment. In ÆMINIUM we adapted the following three permissions kinds:

unique A *unique* access permission to an object reference indicates that there is exactly *one* reference (the current reference to that object) at this moment in time. A *unique* access permission allows clients to read and modify the object.

shared A *shared* access permission to an object reference indicates that there is an arbitrary number of references to the object in the system and *all* the permissions are *shared*. A *shared* access permission allows the client to read and modify the object.

immutable An *immutable* access permission to an object reference indicates that there is an arbitrary number of references to the object in the system and *all* of them are *immutable*. An *immutable* access permission allows only read access to the object.

Figure 3.1: Permission Split and Join Operations

Split Operations						Join Operations				
unique	$\rightarrow$	immutable	$+$	immutable		immutable	$+$	immutable	$\rightarrow$	unique
unique	$\rightarrow$	shared	$+$	shared		shared	$+$	shared	$\rightarrow$	unique
immutable	$\rightarrow$	immutable	$+$	immutable		immutable	$+$	immutable	$\rightarrow$	immutable
shared	$\rightarrow$	shared	$+$	shared		shared	$+$	shared	$\rightarrow$	shared

Access permissions follow the rules of *linear logic* [47]. They are analogous to physical resources that are unavailable once consumed. Permissions can be converted from one type to another as long as the previously described invariants hold. For instance, a unique AP can be *split* into two shared APs. Because of the linearity of APs the unique AP is gone, having been replaced by two shared APs. Each of the shared APs can be further split into more shared APs, but not into unique or immutable permissions. Using *fractions* [32] for keeping track of the individual AP allows permissions to be *joined*, eventually enabling the recovery of a unique access permission. We call the *immutable* and *shared* permission *symmetric* , because an *unique* splits always in a pair of either of those permission and further splitting yields to the same *symmetric* permission. Figure 3.1 summaries the available split and join operations.

The type system computes the AP flow to the program and automatically splits/joins APs as needed. In ÆMINIUM two expressions may execute concurrently if their permissions do not interfere: that is, they have a *disjoint* set of *unique* permissions or an arbitrary set of overlapping *shared* and *immutable* permissions. To avoid *data races* ÆMINIUM only allows access to shared data within `atomic` blocks. The AP flow obeys the lexical order of statements, meaning that if two pieces of code need the same unique AP, the unique AP will first flow to the first expression and then to the latter one.

A simple example is shown in Listing 3.1. The `endOfYearAccounting` method in line 5 is called at the end of a fiscal year for every bank account in the system. It takes an unique account object as input and first calls the `depositInterest` method to add the outstanding interest to the current bank account and then calls the `sendFinancialStatement` method to generate the financial statement report and sent it to the owner of the account. Both methods require a unique permission to the account object. Therefore the type checker will first pass the account object into the `depositInterest`. Upon the completion of the `depositInterest` method the type checker will regain the unique permission to the account object which then is passed into the `sendFinancialStatement` method. By definition we can only have one unique permission at a time for any given object. Therefore the `sendFinancialStatement` method needs to wait for the completion of the `depositInterest` to return this unique permission before it can execute. This sequentializes the execution of both methods.

Listing 3.1: End of Year Accounting Example

```
public void depositInterest(unique Account account) { ... }

public void sendFinancialStatement(unique Account account) { ... }

public void endOfYearAccounting(unique Account account) {
  depositInterst(account);
  sendFinancialStatement(account);
}
```

3.2 Æminium in a Nutshell

Before discussing the remaining Æminium language concepts in more details, this section provides a quick overview of how the overall approach works. The first and most important part of the system is the type checking of the code. The following illustrations will go over the bank transfer example in more detail and explain step-by-step how the program is type checked with permissions. This explanation is for illustrative purposes and for all formal aspects of the type systems please refer to Chapter 4. The permissions available at the given step are described in comments as a list of variable and permission pair following the format 'variable' : 'permission'.

```
public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
    // from : unique, to : unique, amount : immutable
    withdraw(from, amount);
    deposit(to, amount);
}
```

At the beginning of every method the type checker has to establish the known fact. This means all the currently available permissions. In general this includes all available parameters and the receiver if available. In the case of the transfer method this means that we a unique permission to the `from` and `to` references and an immutable for the `amount` reference.

```
public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
    withdraw(from, amount); // amount : immutable
                            // to : unique
    deposit(to, amount);
}
```

To type check the method call to the `withdraw` method we need to pass in a unique permission for the `from` reference and an immutable permission to `amount` object. The type checker passes in the unique permission to the `from` object and splits and passes in an immutable permission to the `amount` object. This leaves an unique permission ot the `to` object and an immutable permission to the `amount` object behind.

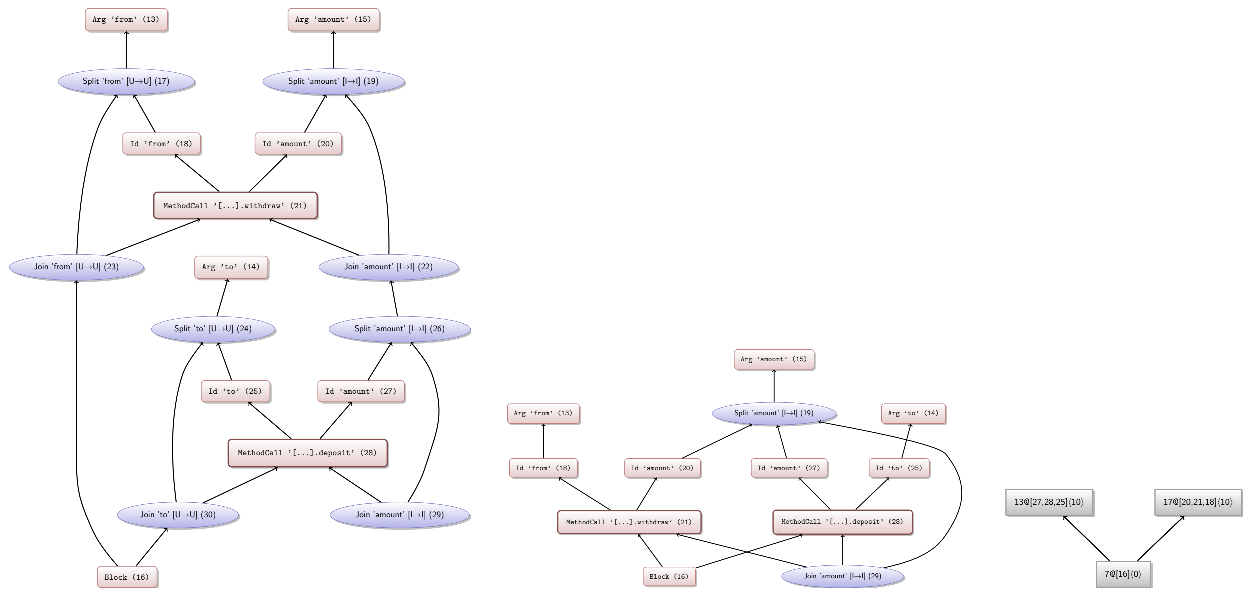
public void transfer(**unique** Account from,
            unique Account to,
            immutable Amount amount) {
    withdraw(from, amount)
    // from : unique, to : unique, amount : immutable
    deposit(to, amount);
}
	After the method call completes the method, indicated by the type annotation, returns the exact permissions which have passed in. The type checker regains a unique permission to the `from` object and an immutable permission to the `amount` object. In our case the type checker immediately joins those immutable permissions back together (cf. Section 5.4.1).

public void transfer(**unique** Account from,
            unique Account to,
            immutable Amount amount) {
    withdraw(from, amount);
    deposit(to, amount); // amount : immutable
                // from : unique
}
	To type check the method call to the `deposit` method we need to pass in a unique permission for the `to` reference and an immutable permission to `amount` object. The type checker passes in the unique permission to the `to` object and splits and passes in an immutable permission to the `amount` object. This leaves an unique permission of the `from` object and an immutable permission to the `amount` object behind.

public void transfer(**unique** Account from,
            unique Account to,
            immutable Amount amount) {
    withdraw(from, amount);
    deposit(to, amount);
    // from : unique, to : unique, amount : immutable
}
	After the method call completes the method, indicated by the type annotation, returns the exact permissions which have passed in. The type checker regains a unique permission to the `to` object and an immutable permission to the `amount` object. In our case the type checker immediately joins those immutable permissions back together (cf. Section 5.4.1). We reached the end of the method an the type checker checks that the permissions we got left match the permission we have to return to the caller, which in this case are the exact same permissions that have been passed in.

Based on the permission flow the compiler can compute a dependency graph for the method. Figure 3.2a shows the dependency graph computed by the compiler. We defer the detailed description of the graph representation we defer to Chapter 5. As mentioned in the text the type checker implementation is eager, meaning it joins permissions back as soon as possible. This leads to the introduction of sequentialization dependencies into the graph. Therefore the compiler transforms the initial dependency graph and transforms into another dependency graph with enhanced parallelism (see Figure 3.2b, cf. Section 5.4.2). The next step comprises dealing with granularity issues by clustering operations into more coarse grain tasks, resulting in a task graph (see Figure 3.2c on the facing page, cf. Section 5.4.3). Based on the task graph the compiler generates source code (cf. Section 5.4.4) which at runtime will create task as necessary and schedules them on the ÆMINIUM runtime (cf. Section 5.5).

Figure 3.2: Dependency and Task Graphs



(a) Dependency Graph from Permission Flow (b) Parallelized Dependency Graph (c) Task Graph

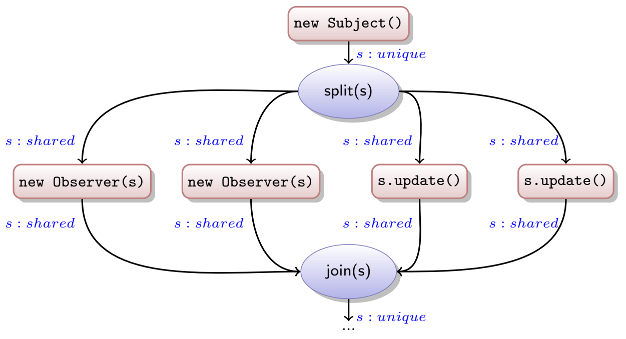
3.3 Data Groups

Although pure APs define a clean execution model for *unique* and *immutable* data, our permission splitting rules will allow all operations on shared data to proceed concurrently. We need a way to express when one operation on a *shared* data structure depends on another. Furthermore, we'd like to control these dependencies, as well as synchronization on *shared* data, at a granularity greater than one object at a time. To illustrate this consider the example shown in Figure 3.3a on the next page. The example shows a simple implementation of the observer pattern [46]. There is `Subject` class which is the source of the events and the `Observer` class which is reception of events. In the `main` function we first create a new `Subject` object and the we create two new `Observer` objects. We pass the subject into the `Observer` constructor so that the newly created object register itself with the subject. After that we trigger the subject twice to notify the subscribed observers. All methods use require a *shared* permission to the `Subject` object because we want to allow the concurrent execution of the code (this means we cannot use *unique* as this would sequentialize the execution) and we want to allow for the possibility to add and remove observers later on (this means we cannot use *immutable* permissions) as those would prohibit the update of the subject object). we defined that operations can run in parallel if the intersection of their required permission does not contain a unique permission. The creation of the observer objects and the calls to the `update` method only overlap in their need to a shared permission to the subject object. This leads, as shown in Figure 3.3b on the following page, to the concurrent exception of all those operations. This is not necessarily what we planned to express. We wanted to first

Figure 3.3: Observer Example

```
class Subject {
    unique Subject() { ... }
    void add(shared Observer obs) : shared { ... }
    void del(shared Observer obs) : shared { ... }
    void update() : shared { ... }
}

class Observer {
    shared Observer(shared Subject s) {
        s.add(this);
    }
    public notify(shared Subject s) : shared { ... }
}

void main() {
  unique Subject s = new Subject();
  shared Observer o1 = new Observer(s);
  shared Observer o2 = new Observer(s);

  s.update();
  s.update();
}
```

(a) Source Code

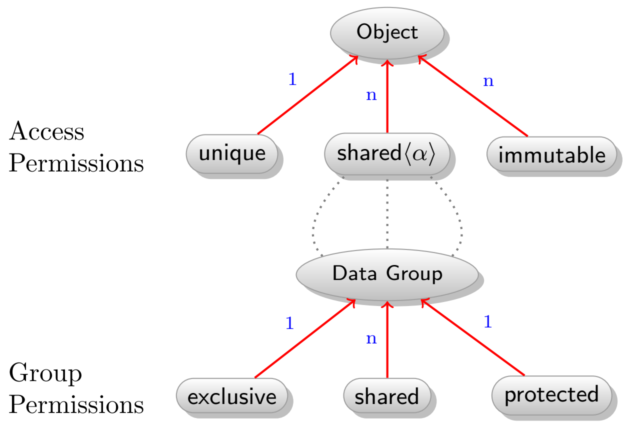


(b) Dependency Graph

create the observers and then allow them to receive the updates. In the current way all four operations can execute in any order. We could force an order by using *unique* permissions for the subject object, but this would inhibit parallelism and might not be achievable if we need to store a reference to the subject object in a field.

To address this challenge we leverage *data groups* (DG, [62]). A data group represents an abstract collection of objects. Using data groups for grouping multiple objects differs from previous work, which used data groups exclusively to partition the state of one object. Data groups resemble closely to Greenhouse's *regions* [49]. Regions allows the grouping of state across multiple objects and its association with a specific lock. Data groups allows grouping of state across multiple states and provides an abstract protection mechanism through `atomic` blocks. When an object is part of a data group, we say that this object is *owned* by that data group. In ÆMINIUM all *shared* objects must be part of exactly one data group. We write *shared*$\langle$*myGroup*$\rangle$ to indicate that the shared object is part of the data group *myGroup*.

Figure 3.4: Permissions in ÆMINIUM. Shows different permission kinds and what each permission controls (including arity). Access permissions control access to objects and group permissions control access to data groups of shared objects. There can only exist one unique, exclusive or protected permission to an object or data group at a time in the system, while there can be an arbitrary number of shared and immutable permissions. Shared permissions refer to the data group to which they belong to (e.g., $shared\langle\alpha\rangle$ means the object belongs to data group α).



Additionally, we adapt the concept of access permissions to data groups and call them *data group permissions* (GP). ÆMINIUM currently defines the following data group permissions:

exclusive There is at most one *exclusive* GP to a data group in the whole system at a time. This resembles a unique AP. Similar to a unique permission, a exclusive GP represents the only currently existing permission through which the data of the data group can be accessed. This allows access to shared data group objects without synchronization.

An *exclusive* group permission (similar to unique access permissions) act like “thread-local” data (although we do not have the notions of threads in ÆMINIUM). An execution path that holds an *exclusive* group permission can safely access the associated shared objects of the group without synchronization. This is an important feature as many data structure intrinsically require shared access permissions to the objects they are composed of (e.g., a doubly linked list which requires at least two valid references to its linked node objects).

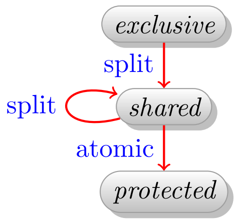
shared A *shared* GP resembles a shared AP: there can be an arbitrarily number of shared GP in the system. Having a shared GP does not grant any kind of access to the associated data because there is the danger of data races.

protected A *protected* GP indicates that the access to its shared data is safe because the access to the *shared* data group has been protected by a corresponding `atomic` block. The semantics of protected

Figure 3.5: Group Permission Splitting/Joining via Shared and Atomic blocks. The notation $gr : gp$ means that we have group permission gp for data group gr.

```
// $\overline{gr : gp}$ with
// $\overline{gp} \in \{exclusive, shared\}$
split ⟨$\overline{gr}$⟩ {
    // $\overline{gr : gp}$ with
    // $\overline{gp} : shared$
    atomic ⟨$gr_i$⟩ {
        // $gr_i : protected$
    }
    // $\overline{gr : gp}$ with
    // $\overline{gp} : shared$
}
// $\overline{gr : gp}$ with
// $\overline{gp} \in \{exclusive, shared\}$
```

(a) Split/Atomic Block


$$\begin{tikzcd} exclusive \arrow[d, "split"'] \\ shared \arrow[loop left, "split"] \arrow[d, "atomic"'] \\ protected \end{tikzcd}$$

(b) Group Permission Conversion Diagram

permissions is that there can only be one protected permission per data group at a time. This is enforced by the runtime system.

Figure 3.4 provides an global overview of all available permissions in the ÆMINIUM system. Access permissions are used to classify object references and consist of *unique*, *shared* and *immutable*. By definition every shared object must be associated with a data group (e.g., α) for which we use a data group permission *exclusive*, *shared* or *protected*.

3.3.1 Management of Data Group Permissions

Unlike the automatic splitting of access permissions, data group permissions are split and joined manually to provide the programmer with better control over dependencies between operations. By default, each operation on a data group depends on the previous operation on that data group; when the operations are conceptually independent, an explicit `split` block is used to split an *exclusive* GP into an arbitrarily number of *shared* GPs (see Figure 3.5). The `split` block specifies data groups for which it splits the available permission (either exclusive or shared) into more shared permissions (one for each statement in the body). Group permissions to data groups not mentioned are simply passed through its body. The available permissions inside the body are partitioned into disjoint sets. Each one of the those permission subsets flows to one statement of the body. This means that if multiple statements in the block require the same unique AP or an exclusive GP (which is not mentioned in the `split` block) then the code will not typecheck because permissions cannot be duplicated. After the completion of all body statements, the shared block joins the generated shared permissions back to the permission that existed before the block was entered.

In order to give programmers control over the granularity of synchronization, each `atomic` block protects access to objects in the particular data groups that are specified at the `atomic` block entry point. It will provide a *protected* GP for the specified data group to its body expression. The specification of the data group is optional as the compiler can automatically infer the required data groups. Providing an explicit annotation however provides a useful documentation of the programmers intent and helps catching unintended data accesses. In particular, the semantics of the `atomic` block is that its body is executed as if it has exclusive access to the shared data associated with the specified data group. Similar to the `split` block, the `atomic` block will upon its completion revert the GP to the state it was in before entering the `atomic` block. The semantics of `split` and `atomic` blocks is illustrated by example in Figure 3.5.

Data groups are declared inside classes in a similar way to fields (see Figure 3.2, line 6). Data groups are only visible inside classes and their subclasses (similar to Java's `protected`). Before accessing data associated with those inner groups, the programmer must gain access to those data groups via an '**unpackInnerGroups** $\{\ldots\}$' construct. The `unpackInnerGroups` block will trade the permission to the owner group of the receiver object for permissions to inner groups defined in the receiver's class. This exchange prohibits recursive method calls from accessing the same inner groups, which would violate the permission invariants (e.g., only one exclusive data group permission per data group). What happens is that when `unpackInnerGroups` is called, the exclusive permission for the "owner" is replaced by exclusive permissions for the inner data groups of the receiver object (i.e., the "this" object). This approach transitively avoids the need for synchronization. Analogously, when the client has either a shared or protected permission to the owner (rather than exclusive), the owner permission is replaced by a shared permission to the inner groups.

We decided to use this semantics to allow more concurrency at the cost of having additional synchronization on sub datagroups. An alternative semantics we considered was that `unpackInnerGroups` replaces the exclusive permission of the "owner" with exclusive permissions for the inner groups and protected permissions of the "owner" with protected permissions for the inner groups. To preserve soundness we would not allow the unpacking of inner groups of shared owners. The reason for this is that if we would allows the unpacking of `shared` inner group permissions another concurrent entity could use an atomic block to gain a second protected group permission (remember the protection of the "owner" group is not expanded to the inner groups to gain more parallelism). This approach would avoid the extra synchronization on the inner groups, but would loose potential parallelism by not be able to unpack shared permissions (which would enable parallelism amongst inner data groups). The `unpackInnerGroups` block can automatically be inferred by the compiler, but adding it explicitly helps the documentation and capturing the programmers intent.

Listing 3.2: A DoubleLinkedList with Data Groups. The example has two `add` methods. The first one requires an exclusive permission to the owner and transitively provides an exclusive permission to the inner groups, and does not requires synchronization. The second version only requires a shared permission to the owner and only provides shared permissions to the inner groups, requiring synchronization i.e. `atomic` blocks. In comments '//' we show which permissions we currently hold via the notation $dg : gp$, meaning for data group dg we have permission gp.

```
class DoubleLinkedListItem⟨owner, data⟩ {
  ... // standard double linked list item
}

class DoubleLinkedList⟨data⟩ {
  group⟨internal⟩ // inner data group

  // 'head' belonging to inner data group 'internal'
  shared⟨internal⟩ DoubleLinkedListItem⟨internal, data⟩ head;

  void add⟨exclusive owner, shared data⟩(shared⟨data⟩ Object⟨data⟩ o)
    : shared⟨owner⟩ // shared permission to the receiver
  {
     // owner : exclusive, data : shared
     unpackInnerGroups {
        // internal : exclusive, data : shared
        // access internal data directly
     }
     // owner : exclusive, data : shared
  }

  void add⟨shared owner, shared data⟩(shared⟨data⟩ Object⟨data⟩ o)
    : shared⟨owner⟩ // shared permission to the receiver
  {
     // owner : shared, data : shared
     unpackInnerGroups {
        // internal : shared, data : shared
        atomic ⟨internal⟩ {
           // internal : protected, data : shared
           // need protection to access internal data
        }
     }
     // owner : shared, data : shared
  }
  ...
}
```

3.3.2 Discussion

The introduction of data groups and data group permissions allows programmers to introduce nondeterminism when they need it, but ensures that they are explicit about where nondeterminism is permitted and helps them

Listing 3.3: Producer/Consumer Example

```
class ProducerConsumer {
  static void producer⟨shared γ⟩(shared⟨γ⟩ Queue⟨γ⟩ q) {
    // α : shared
    atomic ⟨γ⟩ {
          // α : protected
          ...
    }
  }
  static void consumer⟨shared γ⟩(shared⟨γ⟩ Queue⟨γ⟩ q) {
    // α : shared
    atomic ⟨γ⟩ {
          // α : protected
          ...
    }
  }
  static shared⟨γ⟩ Queue⟨γ⟩ createQueue⟨exclusive γ⟩(){...}
  static void disposeQueue⟨exclusive γ⟩(shared⟨γ⟩ Queue⟨γ⟩ q){...}

  static void main⟨exclusive α⟩() {
    // α : exclusive
    shared⟨α⟩ Queue⟨α⟩ q = createQueue⟨α⟩()

    split ⟨α⟩ {
        producer⟨α⟩(q)   // α : shared
        consumer⟨α⟩(q)   // α : shared
    }
    // α : exclusive
    disposeQueue⟨α⟩(q)
  }
}
```

to control the granularity of parallelization and therefore of synchronization. Nondeterminism can only be introduced via explicit `split` blocks, and its impact is limited to accesses within that block. This explicitness helps ensure that programmers have thought about the semantics of their program enough to avoid errors due to unexpected nondeterminism. Furthermore, data groups allow coarse-grained synchronization because an `atomic` block on a data group protects all the objects within that data group, eliminating the need to synchronize separately on each object. In the case of an exclusive group permission, no synchronization is needed at all.

To make this more clear, consider the doubly linked list example in Figure 3.2. In line 5, the `Double-LinkedList` class is defined with group parameter `data`, using the same syntax as *Java* type parameters. The `data` group parameter specifies the data group to which the objects stored in the list belong. Line 6 defines a new data group called 'internal'. Line 9 declares the 'head' field pointing to the chain of 'DoubleLinkedListItems' which are all associated with the 'internal' data group of the surrounding 'DoubleLinkedList'. Because inner groups are not visible outside the class it is impossible for these objects to leave the scope of the class. This strong encapsulation resembles ownership types [39], and allows ÆMINIUM developers to *incrementally* refine their internal data structures to increase internal concurrency (e.g., modifying a hash table that uses one data group for all hash buckets to an implementation that uses one data group per hash bucket).

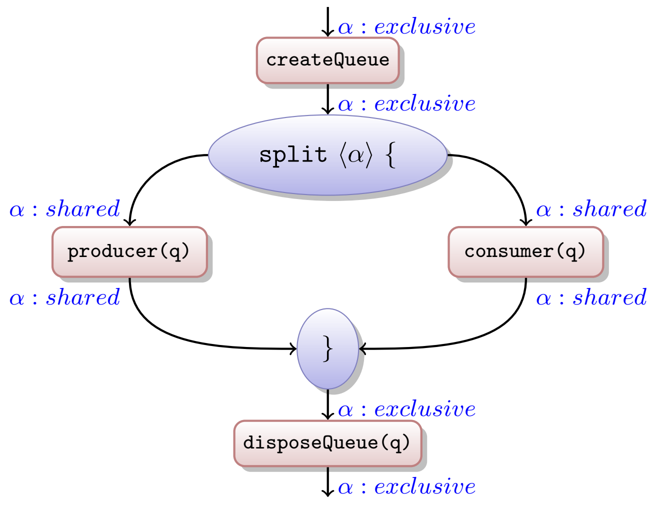
Lines 11 and 22 show the definitions of two add functions that specify data group parameters along with their required permissions. The signature of the two `add` methods are identical, with the exception that the `add` method in line 11 requires an exclusive permission to the data group that owns the receiver, while the add method in 22 requires a shared GP. The effect of this difference can be observed in the implementation of the corresponding bodies. In the case of the `add` method that requires an exclusive permission to the receiver's data group, the `unpackInnerGroups` can provide an exclusive permission to the inner data groups, which in turn allows the programmer to access the shared inner state without any synchronization. In the case of the `add` method that requires a shared permission to the receiver's data group, the `unpackInnerGroups` can only provide a shared permission to the inner data groups, requiring the programmer to synchronize on the inner data group (line 28).

Note that the current design of ÆMINIUM only protects against race conditions and not against deadlocks. The latter has been handled in prior work [31], and is orthogonal to our approach, so it is left out of this discussion for simplicity.

3.4 Producer/Consumer Example

After the discussion of access permission, data groups and their correlation we now present an example for a producer/consumer in ÆMINIUM (see Figure 3.3). The program starts execution at the global entry method `main` (line 19). When entering the body it has an exclusive permission to a data group α. This permission will first flow into the `createQueue` method call (line 21). The exclusive permission matches the method permission requirements as specified in line 16. After the `createQueue` call returns the exclusive permission to α, the permission flows into the *split* block at line 23. As previously described, the `split` block will replace the exclusive permission with one corresponding shared permission for each statement in its body. This leads to the fact that one shared permission to α is flowing in parallel to the `producer` and `consumer` method calls (line 24 + 25). After those calls have been completed, and therefore returned their shared permissions to α, the share block will collect them and join them back together to an exclusive permission (line 26). This newly gained exclusive permission is then fed to the `disposeQueue` method call. Note that if either `producer` or `consumer` want to access the shared queue, they first have to protect their access to this data group via an atomic block (lines 4 and 11). Figure 3.6 shows the resulting permission flow and the derived data flow graph for this example program.

Figure 3.6: Data Flow Graph for Producer/Consumer Example



3.5 Dataflow is not Fork/Join

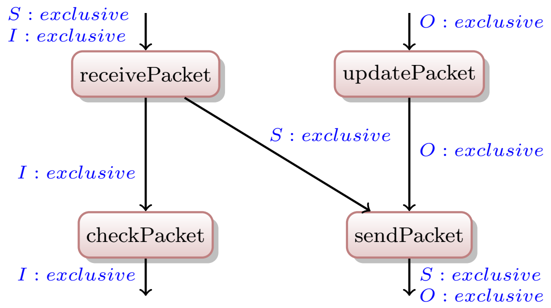
ÆMINIUM supports both *dataflow* and *fork/join* parallelism. To better understand the difference between those concepts consider the example shown in Listing 3.4. The `exchange` function, which could be part of a bi-directional ring network implementation, receives a new packet via the provided socket *s* into the Packet *inp*. It then checks the newly received packet *inp* for errors (e.g., that checksums match). The function then updates the outgoing packet *outp* (e.g., updates header fields and re-computes checksums), before this packet is sent through the socket.

Assuming that all functions called in the `exchange` method require *exclusive* permissions to the corresponding data groups, the permission flow forms a graph as shown in Figure 3.7. The graph shows that receiving the incoming packet can be performed in parallel to updating the outgoing packet. As soon as the incoming packet has been received the newly received packet can be checked. When additionally the updates of the outgoing packet have completed, the outgoing packet can be sent in parallel to checking of the incoming packet. This kind of parallelism is naturally supported by ÆMINIUM's dataflow approach, but cannot be directly expressed in a fork/join paradigm unless extra dependencies or synchronization is used.

Listing 3.4: Exchange Source Code

```
void exchange⟨exclusive S,
              exclusive I,
              exclusive O⟩(shared⟨S⟩ Socket s,
                           shared⟨I⟩ Packet inp,
                           shared⟨O⟩ Packet outp) {
    receivePacket⟨S, I⟩(s, inp);
    checkPacket⟨I⟩(inp);
    updatePacket⟨O⟩(outp);
    sendPacket⟨S, O⟩(s, outp);
}
```

Figure 3.7: Data Flow Graph for `exchange` Function (for simplicity we show only the flow of data group permissions as the access permissions do not cause additional dependencies)



Formalism

This section formalizes the object-oriented μÆMINIUM core language. We briefly discuss the syntax of the language and then elaborate on how the static and dynamic semantics of the calculus prohibit race conditions. We conclude this section by giving an overview of the soundness proof. The complete proof is presented in Appendix A. The goal of μÆMINIUM is to explore a simple, efficient mechanism to track data dependencies via permission flow and to guarantee the absence of race conditions. Because only shared data can lead to race conditions and the tracking of object permissions and data group permissions can be done using similar mechanisms, we focused the core calculus on modeling data groups and data group permissions, assuming that all data is implicitly shared. μ ÆMINIUM's typechecking rules generate a *data group configuration* representing the graph of dependencies between primitive expressions in the language; this configuration is used along with run-time permissions to model parallel execution in the dynamic semantics. Access permission have been previously formalized in [21].

4.1 Syntax

The grammar of μÆMINIUM is shown in Figure 4.1 and is formulated as an extension to *Featherweight Java* (FJ, [56]). FJ is small core calculus for a functional subset of the *Java* programming language. The FJ language is limited to objects with fields, field reads, method calls, casts and object creation.

In a nutshell the major extensions to FJ are: *i*) addition of data group parameters to method calls, class and method declarations; *ii*) addition of group types and extensions of the object types to be parametrized with group parameters; *iii*) new language constructs to deal with data groups and to support field assignments.

We use the overbar notation to abbreviate a list of elements (e.g. $\overline{x : T} = x_1 : T_1, \ldots, x_n : T_n$). Unless otherwise mentioned this notation includes the empty list. We write $\bullet$ to indicate the empty sequence.

A program consists of a set of classes and a `main` method. In μÆMINIUM the global starting expression of *FJ* is explicitly wrapped in a `main` method, to provide an initial data group for the top level objects. A class declaration (*CL*) gives the class a unique name *C* and defines its data group parameters, internal data groups (G), fields (F) and methods (M). Note that the sequence of data group parameters may not be empty, and instead of having an explicit `owner` parameter, the first data group parameter specifies the data group to which the class instances belong. μÆMINIUM does not provide an explicit constructor. Upon creation of a new object all its fields are initialized to `null` and must later be explicitly set. ALthough this might seem to

Figure 4.1: μÆMINIUM **Grammar**

$$
\begin{array}{lrl}
\textit{(programs)} & P & ::= \langle \overline{CL}, main \rangle \\
\textit{(class decl.)} & CL & ::= \texttt{class}\ C\langle\overline{\alpha}, \overline{\beta}\rangle\ \texttt{extends}\ D\langle\overline{\alpha}\rangle\ \{\overline{G}\ \overline{F}\ \overline{M}\} \\
\textit{(field decl.)} & F & ::= T\ f \\
\textit{(group decl.)} & G & ::= \texttt{group}\langle gn \rangle \\
\textit{(method decl.)} & M & ::= T_r\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x})\ \{\ e\ \} \\
\textit{(main meth.)} & main & ::= C\langle\alpha\rangle\ \texttt{main}\langle \mathit{exclusive}\ \alpha\rangle()\ \{\ e\ \} \\
\textit{(values)} & v & ::= o \mid \texttt{null} \\
\textit{(references)} & r & ::= x \mid v \\
\textit{(group ref.)} & gr & ::= r.gn \mid \alpha \\
\textit{(expressions)} & e & ::= a \\
 & & \quad \mid \texttt{unpackGroupsOf}\ r\ \texttt{in}\ e \\
 & & \quad \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \quad \mid \texttt{atomic}\ \langle gr \rangle\ e \\
 & & \quad \mid \texttt{split}\ \langle\overline{gr}\rangle\ \texttt{between}\ e_1 \parallel e_2 \\
 & & \quad \mid \texttt{inatomic}\ \langle gr \rangle\ e \\
\textit{(atoms)} & a & ::= r \\
 & & \quad \mid r.f \\
 & & \quad \mid r.f := r \\
 & & \quad \mid r.m\langle\overline{gr}\rangle(\overline{r}) \\
 & & \quad \mid \texttt{new}\ C\langle\overline{gr}\rangle(\overline{r}) \\
\textit{(types)} & T & ::= C\langle\overline{gr}\rangle \mid \mathbb{G} \mid \bot \\
\textit{(object)} & obj & ::= C[\overline{f = v}] \\
\textit{(group perm.)} & gp & ::= \mathit{exclusive} \mid \mathit{shared} \mid \mathit{protected} \\
\textit{(group state)} & S & ::= U \mid L \\
\textit{(class table)} & CT & ::= \bullet \mid CT, \langle C \mapsto CL \rangle
\end{array}
$$

$$
\begin{array}{ccc}
C, D, E \in \textsc{Classes} & m \in \textsc{Methods} & f \in \textsc{Fields} \\
x, y, \texttt{this} \in \textsc{Vars} & \alpha, \beta, \gamma \in \textsc{Group Vars} & o \in \textsc{Obj. Refs.} \\
 & gn \in \textsc{Group Names} &
\end{array}
$$

be a severe issue, the initialization problem is a general issue which has already been studied and solutions have been proposed [79]. Fields (*F*) are declared with a name and type. Data groups (*G*) are declared by name, which is passed to the group constructor. Methods (*M*) specify their result type, the data group permissions they require, their formal parameters and a body expression.

We syntactically distinguish between expressions and possibly effectful atoms. Atoms are straightforward and consist of field read and assignment, method invocation and new object creation. Besides the standard let binding (`let`), expressions consist of atomic blocks (`atomic`) which specify the data group they protect access to and a body expression; an operation that exchanges permission to the owner of an object for permission to its inner data groups (`unpackGroupsOf`), which specifies the object and an expression which should gain access to the inner groups of the specified object (the `unpackInnerGroups` of ÆMINIUM essentially limits the object reference to the receiver object); and a share primitive (`split`), which specifies which data groups should be shared between the two specified expressions. Note that the sequence of data group references in the share construct must be non-empty. The inatomic primitive (`inatomic`) does not appear at the source level and is only used as an intermediate form for tracking entered atomic blocks. We use a global class table (CT) to map class names to class declarations.

4.2 Static Semantics

This section first provides an overview of all definition forms, then discusses the detailed typing rules. We implicitly assume that names of fields, groups and methods in a class declaration are unique.

4.2.1 Typing Context

The *typing context* Γ contains all the typing information for object references and data group references. We use $\mathbb{G}$ as the type for all data group references and use $\perp$ to denote a *unit* type.

(Typing Context) $\Gamma \quad ::= \quad \bullet \mid \Gamma, x : C\langle\overline{gr}\rangle \mid \Gamma, gr : \mathbb{G} \mid \Gamma, x : \perp$

4.2.2 Store Typing

The *store typing* Σ contains all the typing information for location in the heap.

(Store Typing) $\Sigma \quad ::= \quad \bullet \mid \Sigma, v : C\langle\overline{gr}\rangle \mid \Sigma, v : \perp$

4.2.3 Permission Context

The permission context Δ is a linear context that keeps track of the currently available permissions. We write $gr : gp$ to indicate that we have group permission gp for data group gr.

(Linear Context) $\Delta \quad ::= \quad \bullet \mid \Delta, gr : gp$

4.2.4 Data Group Configuration

The *data group configuration* $\mathcal{G}$ hierarchically tracks the data group requirements of an expression, including any ordering or concurrency among those requirements. It vaguely resembles NESL's [24] approach for tracking profiling information, but instead of tracking operation costs we track permission requirements. A data-group configuration can either be empty (•); a collection of group references ($\{\overline{gr}\}$), indicating the permission requirements of the current expression; the sequential composition of data group configurations ($\oplus$), used to combined data group configurations of expressions that are sequentially ordered, or the parallel composition of data group configurations ($\|$), used to combine data group configurations of expressions that are executed in parallel. We also define a global data group configuration table ($\mathcal{GT}$) which maps class and method tuples to data group configurations.

$$
\begin{array}{lrl}
\text{(DG configuration)} & \mathcal{G} ::= & \bullet \mid \{\overline{gr}\} \mid (\mathcal{G}_1 \oplus \mathcal{G}_2) \mid (\mathcal{G}_1 \parallel \mathcal{G}_2) \\
\text{($\mathcal{G}$ table)} & \mathcal{GT} ::= & \bullet \mid \mathcal{GT}, \langle (C, m) \mapsto \mathcal{G} \rangle
\end{array}
$$

4.2.5 Typing Judgments

We type-check an expression with the judgment $\Gamma|\Sigma|\Delta \vdash_C e : T \mid \mathcal{G}$, which reads: given the typing context Γ, the store typing Σ, the permission context Δ, the expression e checks in the context of class C with type T and has data group configuration $\mathcal{G}$.

We use the judgment $T_f\ f$ ok in C to check that the given field declaration is valid in class C.

We use the judgment $T_r\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x})\ \{\ e\ \}$ ok in C to check that the method declaration is valid in class C.

4.2.6 Helper Functions

Throughout the typing and evaluation rules we use a several helper functions to abbreviate common functionality. For space reasons we delegate the full definitions of these function in Appendix A and just provide a short overview of their effects in Figure 4.2.

4.2.7 Typing Rules

The typing rules are shown in Figure 4.3 and 4.4. Most rules are straightforward; we highlight the most interesting ones. T-PROGRAM starts the checking with a top-level data group α. The T-UNPACKGROUPSIN-* rules exchange a permission to the data group of an object for a permission (exclusive or shared, depending

Figure 4.2: μÆMINIUM **Helper Functions**

$fields(C) = \overline{F}$	returns fields of class C and its superclasses
$groupDecls(C) = \overline{gn}$	returns the declared groups of class C and its superclasses
$override(C, m)\ ok$	checks if a method correctly overrides another method
$requiredPerms(\mathcal{G}) = \overline{gr}$	returns the set of all permissions in $\mathcal{G}$
$requiredTokens(e) = \{\overline{gr@L}\}$	return the set of group access tokens for which e contains an corresponding `inatomic`.
$mdecl(C, m) = M$	looks up the method declaration of m in class C
$mbody(C, m) = \overline{\gamma}.\overline{x}.e \times \mathcal{G}$	looks up the method body of m in class C, and returns the body expression with the method parameter names and the data group configuration

Figure 4.3: Static Semantics of μÆMINIUM **top-level constructs.**

T-PROGRAM

$$\frac{\overline{CL}\ ok \quad main = C\langle\alpha\rangle\ \texttt{main}\langle exclusive\ \alpha\rangle()\ \{\ e\ \} \quad (\alpha : \mathbb{G}) | \bullet | (\alpha : exclusive) \vdash e : T\ |\mathcal{G} \quad (\alpha : \mathbb{G}) \vdash T <: C\langle\alpha\rangle}{\langle\overline{CL}, main\rangle : C\langle\alpha\rangle}$$

T-CLASS

$$\frac{\overline{M}\ \text{ok in C} \qquad \overline{F}\ \text{ok in C}}{\texttt{class}\ C\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends}\ D\langle\overline{\alpha}\rangle\ \{\overline{G}\ \overline{F}\ \overline{M}\}\ ok}$$

T-FIELD

$$\frac{CT(C) = \texttt{class}\ C\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends}\ D\langle\overline{\alpha}\rangle\ \{\overline{G}\ \overline{FM}\} \quad (\overline{\alpha : \mathbb{G}}, \overline{\beta : \mathbb{G}}, this : C\langle\overline{\alpha},\overline{\beta}\rangle, \overline{G : \mathbb{G}}) \vdash E\langle\overline{gr_E}\rangle\ ok}{E\langle\overline{gr_E}\rangle\ f\ \text{ok in C}}$$

T-METHOD

$$\frac{\begin{array}{c} CT(C) = \texttt{class}\ C\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends}\ D\langle\overline{\alpha}\rangle\ \{\overline{G}\ \overline{FM}\} \\ override(C, m)\ ok \qquad \Gamma = (this : C\langle\overline{\alpha},\overline{\beta}\rangle, \overline{\alpha} : \mathbb{G}, \overline{\beta} : \mathbb{G}, \overline{\gamma} : \mathbb{G}) \\ \Gamma \vdash \overline{T_x}\ ok \qquad \Gamma, (\overline{x : T_x}) | \bullet | (\overline{\gamma : gp}) \vdash_C e : T_e \mid \mathcal{G} \qquad \Gamma \vdash T_e <: T_r \end{array}}{T_r\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x})\ \{e\} \quad \text{ok in C}}$$

on the outer permission) to the inner groups of that object. T-SHARE splits the incoming permission context in two, duplicating the named *shared* permissions, while T-ATOMIC allows the protected expression to treat a *shared* data group as *protected*. T-LET supports sequential composition, as specified by the group configuration $\mathcal{G}_1 \oplus \mathcal{G}_2$, while T-SHARE specifies parallel use of any shared groups, as specified by the group configuration $\mathcal{G}_1 \parallel \mathcal{G}_2$. T-FIELD-READ and T-FIELD-ASSIGN requires an *exclusive* or *protected* permission to the first data group parameter (gr_0) of the object being read or assigned. This ensures that either a data group is unshared, or it is locked with an `atomic` section before being used. Field reads and writes generate

Figure 4.4: Static Semantics of μÆMINIUM Expression.

T-UNPACKGROUPSIN-EXCLUSIVE

$$\frac{\begin{array}{c}\Gamma|\Sigma \vdash r : C\langle\overline{gr}\rangle \\ \Delta = \Delta', (gr_0 : exclusive) \qquad groupDecls(C) = \overline{gn} \qquad \Gamma, (\overline{r.gn : \mathbb{G}})|\Sigma|\Delta', (\overline{r.gn : exclusive}) \vdash e : T \mid \mathcal{G}\end{array}}{\Gamma|\Sigma|\Delta \vdash_C \ \mathtt{unpackGroupsOf}\ r\ \mathtt{in}\ e : T \mid (\{gr_0, \overline{r.gn}\} \oplus \mathcal{G})}$$

T-UNPACKGROUPSIN-SHARED

$$\frac{\begin{array}{c}\Gamma|\Sigma \vdash r : C\langle\overline{gr}\rangle \qquad \Delta = \Delta', (gr_0 : gp) \\ gp \in \{shared, protected\} \qquad groupDecls(C) = \overline{gn} \qquad \Gamma, (\overline{r.gn : \mathbb{G}})|\Sigma|\Delta', (\overline{r.gn : shared}) \vdash e : T \mid \mathcal{G}\end{array}}{\Gamma|\Sigma|\Delta \vdash_C \ \mathtt{unpackGroupsOf}\ r\ \mathtt{in}\ e : T \mid (\{gr_0, \overline{r.gn}\}\} \oplus \mathcal{G})}$$

T-SPLIT

$$\frac{\begin{array}{c}\{\overline{gp}\} \subseteq \{exclusive, shared\} \qquad \Delta = \Delta_1, \Delta_2, \Delta_r \\ \Gamma|\Sigma|(\Delta_1, \overline{gr : shared}) \vdash_C e_1 : T_1 \mid \mathcal{G}_1 \qquad \Gamma|\Sigma|(\Delta_2, \overline{gr : shared}) \vdash_C e_2 : T_2 \mid \mathcal{G}_2 \qquad \mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2)\end{array}}{\Gamma|\Sigma|(\Delta, \overline{gr : gp}) \vdash_C \ \mathtt{split}\ \langle\overline{gr}\rangle\ \mathtt{between}\ e_1 \parallel e_2 : \bot \mid \mathcal{G}}$$

T-ATOMIC

$$\frac{\Gamma|\Sigma \vdash gr : \mathbb{G} \qquad \Gamma|\Sigma|(\Delta, gr : protected) \vdash_C e : T \mid \mathcal{G}}{\Gamma|\Sigma|\Delta, (gr : shared) \vdash_C \ \mathtt{atomic}\ \langle gr\rangle\ e : T \mid (\{gr\} \oplus \mathcal{G})}$$

T-NULL

$$\frac{}{\Gamma|\Sigma|\Delta \vdash_C \mathtt{null} : \bot \mid \bullet}$$

T-INATOMIC

$$\frac{\Gamma|\Sigma \vdash gr : \mathbb{G} \qquad \Gamma|\Sigma|\Delta, (gr : protected) \vdash_C e : T \mid \mathcal{G}}{\Gamma|\Sigma|\Delta, (gr : shared) \vdash_C \ \mathtt{inatomic}\ \langle gr\rangle\ e : T \mid (\{gr\} \oplus \mathcal{G})}$$

T-REFERENCE

$$\frac{\Sigma(v) = T}{\Gamma|\Sigma|\Delta \vdash_C v : T \mid \bullet}$$

T-VAR

$$\frac{\Gamma(x) = T}{\Gamma|\Sigma|\Delta \vdash_C x : T \mid \bullet}$$

T-LET

$$\frac{\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1 \mid \mathcal{G}_1 \qquad (\Gamma, x : T_1)|\Sigma|\Delta_1, \Delta_R \vdash_C e_2 : T_2 \mid \mathcal{G}_2}{\Gamma|\Sigma|\Delta_1, \Delta_R \vdash_C \ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : T_2 \mid (\mathcal{G}_1 \oplus \mathcal{G}_2)}$$

T-FIELD-READ

$$\frac{\begin{array}{c}\Gamma|\Sigma \vdash r : D\langle\overline{gr}\rangle, gr_0 : \mathbb{G} \\ gp \in \{exclusive, protected\} \\ fields(D) = \overline{T_f\ f}\end{array}}{\Gamma|\Sigma|\Delta, (gr_0 : gp) \vdash_C r.f_i : T_{fi} \mid \{gr_0\}}$$

T-FIELD-ASSIGN

$$\frac{\begin{array}{c}\Gamma|\Sigma \vdash r_v : T_v, r : D\langle\overline{gr}\rangle, gr_0 : \mathbb{G} \\ gp \in \{exclusive, protected\} \\ fields(D) = \overline{T\ f} \qquad \Gamma \vdash T_v <: T_{fi}\end{array}}{\Gamma|\Sigma|\Delta, (gr_0 : gp) \vdash_C r.f_i := r_v : T_v \mid \{gr_0\}}$$

T-NEW

$$\frac{\begin{array}{c}CT(D) = \ \mathtt{class}\ D\langle\overline{\alpha}, \overline{\beta}\rangle\ \mathtt{extends}\ E\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \\ \Gamma|\Sigma \vdash \overline{gr : \mathbb{G}}\end{array}}{\Gamma|\Sigma|\Delta \vdash_C \ \mathtt{new}\ D\langle\overline{gr}\rangle() : [{}^{\overline{gr}}/_{\overline{\alpha},\overline{\beta}}]D\langle\overline{\alpha}, \overline{\beta}\rangle \mid \bullet}$$

T-CALL

$$\frac{\begin{array}{c}\Gamma|\Sigma \vdash r : T_r, \overline{p : T_p}, \overline{gr : \mathbb{G}} \qquad \Delta \vdash \overline{gr : gp} \qquad T_r = D\langle\overline{gr_D}\rangle \\ CT(D) = \ \mathtt{class}\ D\langle\overline{\alpha}, \overline{\beta}\rangle\ \mathtt{extends}\ E\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \qquad mdecl(D, m) = T_{result}\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x})\{\ e\ \} \\ \Gamma \vdash \overline{T_p} <: \overline{[{}^{\overline{gr},\overline{gr_D}}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]T_x} \qquad \Gamma \vdash T_r <: [{}^{\overline{gr},\overline{gr_D}}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]D\langle\overline{\alpha}, \overline{\beta}\rangle\end{array}}{\Gamma|\Sigma|\Delta \vdash_C r.m\langle\overline{gr}\rangle(\overline{p}) : [{}^{\overline{gr},\overline{gr_D}}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]T_{result} \mid \{\overline{gr}\}}$$

a data group configuration that is just the group being read or assigned. Finally, T-CALL ensures that the data groups required by the called function are provided by the caller.

4.3 Dynamic Semantics

This section first provides an overview of the definition forms used, then discusses the evaluation rules in detail. Instead of generating an explicit data flow graph, the dynamic semantics use the data group configuration together with runtime permission tokens to model the permission flow at runtime and emulate the dependencies.

4.3.1 Store

The *store* μ is a mapping of object references o to objects obj. A store can either be a potentially empty set of object mappings or `race`, which indicates the case that a race condition occurred during the execution (our soundness theorem will show that these races cannot occur in well-typed code). An object is a record consisting of all instance fields. The inner groups (i.e., data groups that are declared by every object) along with their corresponding state are managed separately in the group access token context (cf. Section 4.3.3)

$$\text{(store)} \quad \mu \quad ::= \quad \overline{\langle o \mapsto obj \rangle} \mid \texttt{race}$$

During the evaluation of an expression, differential stores (μ_δ) containing the accessed objects are generated. Those differential stores are merged via the $\uplus$ operator. To generate a new global heap we write $\mu' = [\mu_\delta]\mu$ for element wise update/substitution of objects.

$$\mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2} = \begin{cases} \mu_{\delta_1}, \mu_{\delta_2} & dom(\mu_{\delta_1}) \cap dom(\mu_{\delta_2}) = \bullet \\ \\ \texttt{race} & \textsc{Otherwise} \end{cases} \qquad \mu' = [\mu_\delta]\mu = \begin{cases} \texttt{race} & \mu_\delta = \texttt{race} \\ [o \mapsto obj]\mu & \forall \langle o \mapsto obj \rangle \in \mu_\delta \end{cases}$$

4.3.2 Runtime Permission Context

The *runtime permission context* δ is used to model permission flows at runtime and is either empty or consists of a set of $o.gn$ (i.e. runtime permissions). The runtime semantics do not allow an expression to execute until all of its required permissions, as expressed in its group configuration, are available. A runtime permission can be split and can flow along different paths, just as static permissions can.

The top level permission context always contains only one initial permission to the global data group of the main function. More runtime permissions are successively generated by unpacking inner groups.

$$\text{(runtime permission context)} \quad \delta \quad ::= \quad \bullet \mid \delta, o.gn$$

4.3.3 Group Access Token Context

The *group token context* Ψ is a set of group access tokens, i.e., group references along with their current locking state $S = \{U|L\}$. A locking state U indicates an unlocked state meaning that one atomic block referring to that data group can be entered. A locking state L indicates a locked state meaning that an atomic block referring to that data group is currently executing. There is a controversial discussion [30] regarding the correct semantics for atomic blocks. Some argue that transactional semantics should be used while others argue that lock-based semantics should be used. A major problem with using a lock based approach is associated with the risk of creating deadlocks when different threads try to acquire two locks in the reverse order. The main problems with STM are the lack of unified semantics and an inherently performance problem though the massive overhead caused by STM implementations. We decided to use a lock-based approach for its simplicity of implementation and semantics. In future we might reconsider this decision and evaluate a transactional semantics [71].

There exists exactly one group access token for every data group in the system and unlike runtime permissions, group access tokens *cannot* be split. In several rules the unlocked group access token context is split in a non-deterministic way. This models non-determinism of how atomic blocks can lock data groups. Locked group access tokens are forced to flow into the expression that contains the corresponding `inatomic` . This approach is not strictly necessary but allows us to formulate a stronger preservation induction hypothesis.

$$\text{(group context)} \quad \Psi \quad ::= \quad \bullet \mid \Psi, o.gn@S$$

4.3.4 Evaluation Judgment

To evaluate expressions we use the judgment $\mu|\delta|\Psi|\mathcal{G} \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'$, which reads as follows: given the store (μ), the runtime permissions (δ), the group access tokens (Ψ), the data group configuration ($\mathcal{G}$) the expression e steps to e' and produces a differential store (μ_δ), an updated set of group access tokens (Ψ') and an updated data group configuration ($\mathcal{G}'$).

4.3.5 Program State

A program state is a quintuple of the form $(\mu|\delta|\Psi|\mathcal{G}|e)$, consisting of a store (μ), a runtime permission context (δ), a group access token context (Ψ) of available tokens, a data group configuration ($\mathcal{G}$) and an expression (e). A program state represents a consistent state of the execution. To transition from one program state to another, the expression takes a step following the evaluation judgment and then generates a new global store (see E-TRANS-N in Figure 4.5).

Figure 4.5: μÆMINIUM Program State Transitions Rules

E-TRANS-Z

$$\frac{}{(\mu|\delta|\Psi|\mathcal{G}|e) \mapsto (\mu|\delta|\Psi|\mathcal{G}|e)}$$

E-TRANS-N

$$\frac{\mu|\delta|\Psi|\mathcal{G} \vdash e \mapsto e_1 \dashv \mu_\delta|\Psi_1|\mathcal{G}_1 \qquad \mu_1 = [\mu_\delta]\mu \qquad (\mu_1|\delta|\Psi_1|\mathcal{G}_1|e_1) \mapsto^* (\mu'|\delta|\Psi'|\mathcal{G}'|e')}{(\mu|\delta|\Psi|\mathcal{G}|e) \mapsto^* (\mu'|\delta|\Psi'|\mathcal{G}'|e')}$$

Figure 4.6: Dynamic Semantics of μÆMINIUM Atoms

E-FIELD-READ

$$\frac{\mathcal{G} = \{v_g.gn\} \qquad v_g.gn \in \delta \qquad \mu \vdash \langle v \mapsto C[\overline{f = v_f}]\rangle \qquad \mu_\delta = \langle v \mapsto C[\overline{f = v_f}]\rangle}{\mu|\delta|\Psi|\mathcal{G} \vdash v.f_i \mapsto v_{fi} \dashv \mu_\delta|\Psi|\bullet}$$

E-FIELD-ASSIGN

$$\frac{\begin{array}{c}\mathcal{G} = \{v_g.gn\} \qquad v_g.gn \in \delta \qquad \mu \vdash \langle v_r \mapsto obj_r\rangle \\ obj_r = C[\overline{f_r = v_{f_r}}, f_{ri} = v_{fi}, \overline{f_r = v_{f_r}}] \qquad obj_r' = C[\overline{f_r = v_{f_r}}, f_{ri} = o_v, \overline{f_r = v_{f_r}}] \qquad \mu_\delta = \langle v_r \mapsto obj_r'\rangle\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash v_r.f_{ri} := o_v \mapsto o_v \dashv \mu_\delta|\Psi|\bullet}$$

E-NEW

$$\frac{\mathcal{G} = \bullet \qquad groupDecls(C) = \overline{gn} \qquad o_{new}\ fresh \qquad \mu_\delta = \langle o_{new} \mapsto C[\overline{f = \texttt{null}}]\rangle}{\mu|\delta|\Psi|\mathcal{G} \vdash \ \texttt{new}\ C\langle\overline{v_g.gn}\rangle() \mapsto o_{new} \dashv \mu_\delta|\Psi, \overline{o_{new}.gn@U}|\bullet}$$

E-CALL

$$\frac{\begin{array}{c}\mathcal{G} = \{\overline{v_g.gn}\} \\ \overline{v_g.gn} \in \delta \qquad \mu \vdash \langle v_r \mapsto C[\overline{f = v_{f_r}}]\rangle \qquad mbody(C,m) = \overline{\alpha}.\overline{x}.e \times \mathcal{G}_e \qquad \mathcal{G}' = [{}^{\overline{v_g.gn}}/_{\overline{\alpha}}][{}^{\overline{v_p}}/_{\overline{x}}][{}^{v_r}/_{this}]\mathcal{G}_e\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash v_r.m\langle\overline{v_g.gn}\rangle(\overline{v_p}) \mapsto [{}^{\overline{v_g.gn}}/_{\overline{\alpha}}][{}^{\overline{v_p}}/_{\overline{x}}][{}^{v_r}/_{this}]e \dashv \bullet|\Psi|\mathcal{G}'}$$

4.3.6 Evaluation Rules

The Evaluation rules for atoms are shown in Figure 4.6 and the rules for expressions are shown in Figure 4.7 and 4.8. Once again we describe the most interesting rules. E-FIELD-READ demonstrates the basic approach: we look up the permissions required based on the group context $\mathcal{G}$ (which was computed by the typechecking rules), and the read cannot execute unless and until the required permission is in the permission context δ. Other atom rules are similar. The E-UNPACKGROUPSOF-* rules make the inner permissions available to the enclosed expression if and only if the permission to the outer object is available; otherwise the enclosed expression can only take steps for which these permissions are not required. There are three variants of the let and share rules: one where the first expression takes a step, one where the second steps, and one where

Figure 4.7: Dynamic Semantics of μÆMINIUM Expressions (1/2)

E-UnpackGroupsOf-Replace

$$\frac{\begin{array}{c}\mathcal{G} = (\{v'.gn', \overline{v_r.gn}\} \oplus \mathcal{G}_e) \\ \delta = \delta', v'.gn', \qquad \mu|\delta', \overline{v_r.gn}|\Psi|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'_e \qquad \mathcal{G}' = (\{v'.gn', \overline{v_r.gn}\} \oplus \mathcal{G}'_e)\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e \mapsto \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-UnpackGroupsOf-None

$$\frac{\begin{array}{c}\mathcal{G} = (\{v'.gn', \overline{v_r.gn}\} \oplus \mathcal{G}_e) \\ v'.gn' \notin \delta \qquad \mu|\delta|\Psi|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'_e \qquad \mathcal{G}' = (\{v'.gn, \overline{v_r.gn}\} \oplus \mathcal{G}'_e)\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e \mapsto \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-Let-1

$$\frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2) \qquad \delta_1 = \delta \cap \mathit{requiredPerms}(\mathcal{G}_1) \qquad \Psi = \Psi_1, \Psi_2 \\ \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \qquad \mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e'_1 \dashv \mu_\delta|\Psi'_1|\mathcal{G}'_1 \qquad \mathcal{G}' = (\mathcal{G}'_1 \oplus \mathcal{G}_2) \qquad \Psi' = \Psi'_1 \cup \Psi_2\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mapsto \texttt{let}\ x = e'_1\ \texttt{in}\ e_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-Let-2

$$\frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2) \qquad \delta_2 = \delta - \mathit{requiredPerms}(\mathcal{G}_1) \qquad \Psi = \Psi_1, \Psi_2 \qquad \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \\ \mathit{requiredTokens}(e_2) \subseteq \Psi_2 \qquad \mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e'_2 \dashv \mu_\delta|\Psi'_2|\mathcal{G}'_2 \qquad \Psi' = \Psi_1 \cup \Psi'_2 \qquad \mathcal{G}' = (\mathcal{G}_1 \oplus \mathcal{G}'_2)\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mapsto |\ \texttt{let}\ x = e_1\ \texttt{in}\ e'_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-Let-12

$$\frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2) \qquad \delta_1 = \delta \cap \mathit{requiredPerms}(\mathcal{G}_1) \qquad \delta_2 = \delta - \delta_1 \qquad \Psi = \Psi_1, \Psi_2 \\ \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \qquad \mathit{requiredTokens}(e_2) \subseteq \Psi_2 \qquad \mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e'_1 \dashv \mu_{\delta_1}|\Psi'_1|\mathcal{G}'_1 \\ \mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e'_2 \dashv \mu_{\delta_2}|\Psi'_2|\mathcal{G}'_2 \qquad \Psi = \Psi'_1 \cup \Psi'_2 \qquad \mathcal{G}' = (\mathcal{G}'_1 \oplus \mathcal{G}'_2) \qquad \mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2}\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mapsto \texttt{let}\ x = e'_1\ \texttt{in}\ e'_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-Let-Value

$$\frac{\mathcal{G} = (\bullet \oplus \mathcal{G}_2) \qquad \mathcal{G}' = [^v/_x]\mathcal{G}_2}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{let}\ x = v\ \texttt{in}\ e_2 \mapsto [^v/_x]e_2 \dashv \bullet|\Psi|\mathcal{G}'}$$

E-UnpackGroupsOf-Value

$$\frac{}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ v \mapsto v \dashv \bullet|\Psi|\bullet}$$

both expressions step (this can occur even in the sequentializing LET construct if the permissions required do not overlap). The rules for split differ in that LET divides the permissions without duplicating any, while SPLIT duplicates the permissions named in the `split` block. Finally, the rules for the `atomic` block do not pass a permission to the named data group inwards until a lock is acquired, at which point the state of the lock changes to $@L$ and the expression changes to `inatomic` for tracking purposes. For a more detailed description of each rule cf. Appendix A.

Figure 4.8: Dynamic Semantics of μÆMINIUM Expressions (2/2)

E-SPLIT-1

$$\frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2) \qquad \delta_1 = \delta \cap requiredPerms(\mathcal{G}_1) \qquad \Psi = \Psi_1, \Psi_2 \qquad requiredTokens(e_1) \subseteq \Psi_1 \\ requiredTokens(e_2) \subseteq \Psi_2 \qquad \mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e_1' \dashv \mu_\delta|\Psi_1'|\mathcal{G}_1' \qquad \Psi' = \Psi_1' \cup \Psi_2 \qquad \mathcal{G}' = (\mathcal{G}_1' \parallel \mathcal{G}_2)\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1' \parallel e_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-SPLIT-2

$$\frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2) \qquad \delta_2 = \delta \cap requiredPerms(\mathcal{G}_2) \qquad \Psi = \Psi_1, \Psi_2 \qquad requiredTokens(e_1) \subseteq \Psi_1 \\ requiredTokens(e_2) \subseteq \Psi_2 \qquad \mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e_2' \dashv \mu_\delta|\Psi_2'|\mathcal{G}_2' \qquad \Psi' = \Psi_1 \cup \Psi_2' \qquad \mathcal{G}' = (\mathcal{G}_1 \parallel \mathcal{G}_2')\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-SPLIT-12

$$\frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2) \qquad \delta_1 = \delta \cap requiredPerms(\mathcal{G}_1) \qquad \delta_2 = \delta \cap requiredPerms(\mathcal{G}_2) \qquad \Psi = \Psi_1, \Psi_2 \\ requiredTokens(e_1) \subseteq \Psi_1 \qquad requiredTokens(e_2) \subseteq \Psi_2 \qquad \mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e_1' \dashv \mu_{\delta_1}|\Psi_1'|\mathcal{G}_1' \\ \mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e_2' \dashv \mu_{\delta_2}|\Psi_2'|\mathcal{G}_2' \qquad \mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2} \qquad \Psi' = \Psi_1' \cup \Psi_2' \qquad \mathcal{G}' = (\mathcal{G}_1' \parallel \mathcal{G}_2')\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1' \parallel e_2' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-ATOMIC-STEP1

$$\frac{\mathcal{G} = (\{v.gn\} \oplus \mathcal{G}_e) \qquad v.gn \notin \delta \qquad \mu|\delta|\Psi|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}_e' \qquad \mathcal{G}' = (\{v.gn\} \oplus \mathcal{G}_e')}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{atomic}\ \langle v.gn\rangle\ e \mapsto \texttt{atomic}\ \langle v.gn\rangle\ e' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-ATOMIC-STEP2

$$\frac{\begin{array}{c}\mathcal{G} = (\{v.gn\} \oplus \mathcal{G}_e) \\ \delta = \delta', v.gn \qquad v.gn@U \notin \Psi \qquad \mu|\delta'|\Psi|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}_e' \qquad \mathcal{G}' = (\{v.gn\} \oplus \mathcal{G}_e')\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{atomic}\ \langle v.gn\rangle\ e \mapsto \texttt{atomic}\ \langle v.gn\rangle\ e' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-ATOMIC-INATOMIC

$$\frac{\mathcal{G} = (\{v.gn\} \oplus \mathcal{G}_e) \qquad v.gn \in \delta \qquad \Psi = \Psi'', v.gn@U \qquad \Psi' = \Psi'', v.gn@L}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{atomic}\ \langle v.gn\rangle\ e \mapsto \texttt{inatomic}\ \langle v.gn\rangle\ e \dashv \bullet|\Psi'|\mathcal{G}}$$

E-INATOMIC-STEP

$$\frac{\begin{array}{c}v.gn \in \delta \qquad \Psi = \Psi'', v.gn@L \\ \mathcal{G} = (\{v.gn\} \oplus \mathcal{G}_e) \qquad \mu|\delta|\Psi''|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'''|\mathcal{G}_e' \qquad \Psi' = \Psi''', v.gn@L \qquad \mathcal{G}' = (\{v.gn\} \oplus \mathcal{G}_e')\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{inatomic}\ \langle v.gn\rangle\ e \mapsto \texttt{inatomic}\ \langle v.gn\rangle\ e' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

E-SPLIT-VALUE

$$\frac{\mathcal{G} = (\bullet \parallel \bullet)}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ v_1 \parallel v_2 \mapsto \texttt{null} \dashv \bullet|\Psi|\bullet}$$

E-INATOMIC-VALUE

$$\frac{\Psi = \Psi'', v.gn@L \qquad v.gn \in \delta \qquad \Psi' = \Psi'', v.gn@U}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{inatomic}\ \langle v'.gn\rangle\ v \mapsto v \dashv \bullet|\Psi'|\bullet}$$

4.4 Properties

We prove the correctness of our system by induction on the derivation of program state transitive rules (cf. Figure 4.5). We prove the type *safety* following the standard approach [76] by proofing *progress* and *preservation* separately.

Our *progress* lemma (cf. Appendix A, Lemma 5) states, when given a well formed program state, then either our expression e is a value, or the program state can take a step (i.e., advance to another program state), e is waiting for some resource to become available or execution has stopped because of a null dereference. We prove the correctness of the progress lemma through induction on typing derivation. We show that for every program state (and therefore well typed expression) we can apply our progress lemma.

Our *preservation* lemma (cf. Appendix A, Lemma 6) states, when we have a well formed program state and perform a single step the resulting program state is again well formed. We prove preservation through induction over the evaluation rules.

We define *type safety* (cf. Appendix A, Lemma 1) to state that starting from every well formed program state we can take an arbitrary amount of steps and always produce another well formed program state which is not *stuck* (meaning the expression is not a value and cannot take a step or is not waiting for resources, cf. Appendix A, Definition 2). We prove type safety by induction over the program transitions rules by applying our progress and preservation lemma.

The full definitions and prove details can be found in Appendix A. By proofing the soundness of our system we proved that any well typed program is data race free. This validates our first hypothesis that our system avoids data races.

Implementation

The goal of this chapter is to describe the prototype implementation of ÆMINIUM. The implementation of ÆMINIUM is based on the *Plaid* programming language [12]. The *Plaid* programming language is a *typestate* oriented programming language [85] with built-in support for *access permission*. Having access permissions already built-in into the language allows us to leverage *Plaid*'s typechecker. Additionally we get *Plaid*'s typestate checking infrastructure for free, and vice-versa, users of the *Plaid* language can get free access to concurrent execution of their programs. The intent of this chapter is to provide a brief overview of how the Plaid language is implemented and how we extended this implementation to realize the ÆMINIUM implementation. We will explain language features if they are necessary for the implementation strategy. For more in depth information about *Plaid*'s language features and semantics refer to official language specification [12] and our related publications [13, 72, 85].

5.1 Plaid Primer

This section provides a short introduction to the Plaid programming language explaining all necessary constructs required for this thesis. Please refer to the official *Plaid* language specification [12] for a more in-depth overview of Plaid. By design the *Plaid* language resembles the *Java* language as much as possible. The main conceptional difference between *Plaid* and *Java* is the usage of states instead of classes. Conceptionally, *Plaid* uses state abstractions to naturally encode the various states an object can be in a direct and checkable way. We discuss state composition and state change semantics in [85]. An overview of *Plaid's* type system is given in [72]. Those concepts are orthogonal to ÆMINIUM parallelization approach and we therefore limit ourselves to a subset of *Plaid* which most closely resembles normal *Java*.

Listing 5.1 on the next page shows a simple counter example emphasizing the commonalities with *Java*. In line 1 on the following page we define a new state `Object`. States, similar to *Java* classes, are a collection of state and methods to manipulate this state. Instead of using the `class` keyword *Plaid* uses the `state` keyword to declare such a collection. Like in *Java* we call the instanced of states objects. Line 2 on the next page shows that the `Object` state defines only one method called `toString`. *Plaid*'s method declaration follows the same syntax as a *Java* method declarations with the following exceptions. All method declarations in *Plaid* start with the keyword `method` to indicate the start of a new method declaration. Note that *Plaid* does not support *Java's* modifiers (i.e., `public`, `final`, `abstract`, etc) but has its own (discussed later). After the `method` keyword we have the return type of the method followed by the method name and its parameter list. After the parameter list we have the so called *environment* of the method declared in square brackets. The

Listing 5.1: Basic Plaid Example

```
1  state Object {
2      method immutable String toString() [ local immutable Object this ];
3  }
4
5  state Counter case of Object {
6      var immutable Integer count = 0;
7
8      method void inc() [ unique Counter this ] {
9          this.counter = this.counter + 1;
10     }
11
12     method void dec() [ unique Counter this ] {
13         this.counter = this.counter  1;
14     }
15
16     method immutable Integer get() [ local immutable Counter this ] {
17         this.counter
18     }
19
20     method immutable String toString() [ local immutable Counter this ] {
21        "Counter(" + this.count.toString() + ")"
22     }
23 }
```

environment is an implicit parameter list specifying all the objects that are implicitly passed into the method. As shown in the example the environment contains the declaration of `this` reference. Note the additional `local` keyword in front of the `immutable` permission of the `this` reference. `local` permissions is a permission modifier to correctly implementing the permission join operation between permissions without requiring the user to worry about concrete fractions (cf. [72]). The `this` reference is implicitly passed into the method and therefore we need to specify which permissions we need. After the environment we usually would declare the method body in curly braces, but in this case we finish the declaration with a semicolon to indicate an *abstract* method declaration.

In line 5 we define a new state `Counter` as a sub-state of `Object`. Plaid uses the `case of` instead of *Java's* `extends` to declare sub-typing. The `Counter` defines a local field in line 6. All fields and variable declarations start with either `val` (immutable) or `var` (mutable). In lines 8, 12 and 8 the `Counter` defines various methods to increase, decrease or retrieve the current counter value. In *Plaid*, like in *Smalltalk* [48], everything is an object. This means unlike in *Java* there are no primitive types (like `int`, `boolean`, etc). The addition operation '`this.count + 1`' in line 8 is translated into a method call in on the first operand '`this.count.+(1)`'. This is possible because *Plaid* supports operator overloading. Another important obervation is the absence of the `return` statement in *Plaid*. *Plaid* automatically returns the value of the

Listing 5.2: Plaid Fibonacci Example

```
1 method immutable Integer fibonacci(immutable Integer n) {
2     match ( n <= 2 ) {
3         case True { 1 }
4         default {
5             fibonacci(n−1) + fibonacci(n−2)
6         }
7     }
8 }
```

Listing 5.3: Plaid Boolean

```
1 state Boolean { ... }
2
3 state True case of Boolean { ... }
4
5 state False case of Boolean { ... }
```

last statement in a method body. In line 20 on the facing page the `Counter` objects implements the abstract `toString` method as defined by its super state.

Pattern matching is the only control flow mechanism built into the *Plaid* programming language. The pattern matching in *Plaid* currently works on type level does not allow automatic binding of internal state to temporary variables. The simplest way to describe *Plaid's* match statement is to think of *Java's* `switch` statement combined with the *instanceof* operations to compare for matching types instead of values. An example of *Plaid's* pattern matching is shown in Figure 5.2. The example shows a *Plaid* implementation of the Fibonacci number computation. The example uses a global method defined in line 1. Global methods in *Plaid* are like static methods in *Java* meaning they can be called without having an object instance available. In line 2 the `match` block starts. It will take the result of the expression `n <= 2` and checks which case matches the result type. The result of the comparison is of type `Boolean`.

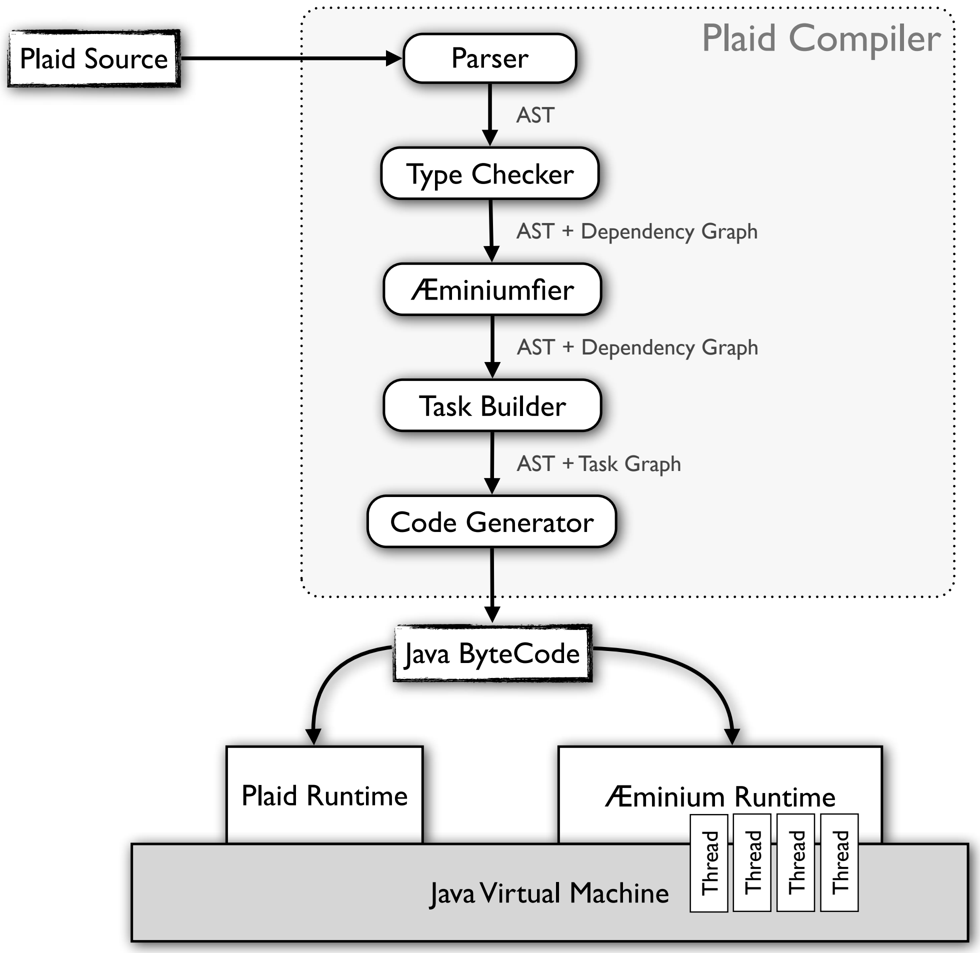
Note that in *Plaid* booleans are not part of the language and are implemented as part of the standard library. Figure 5.3 shows an abbreviated version of *Plaid's* boolean declaration. Line 1 defines the top-level `Boolean` type. In line 3 and 5 define two orthogonal sub-types, one for true values and one for false values.

Coming back to the Fibonacci example in Figure 2 line 3 we define a case to check if the value of the comparison operations is of type `True`. If so we simply return the constant value one. In line 4 the declares the default case which is used when no other case applies. In this case we simply use the recursive definition of fibonacci number to compute the result. Note that the result value of the method body is the value to which the last statement reduces. In this case the last statement is the `match` block which evaluates to the value of the taken case.

5.2 System Architecture

The overall system architecture is shown in Figure 5.1 on the next page. This section provides a high level overview of the general approach before we explore the details in the reminder of this chapter. The compiler user writes *Plaid* code and feeds it into our compiler. The compiler first translates the *Plaid* source code

Figure 5.1: System Architecture



into an *Abstract Syntax Tree (AST)*. The newly generated AST is then used by the type checker to check that the input program does not violate *Plaid* typing rules. Besides checking the programs conformance the type checker also computes a sequential dependency graph based on the permission flow. The AST and the dependency graph is then used by the *Æminiumfier* (cf. Section 5.4.2) which analyses and transforms the sequential dependency graph into a parallel dependency graph. The parallel dependency graph and the AST are then used by by the *Task Builder* (cf. Section 5.4.3) to cluster operations into more coarse tasks. The generated task graph and AST is used by the *Code Generator* (cf. Section 5.4.4) to generate the final *Java* byte code.

The generated code uses the *Plaid* and ÆMINIUM runtime libraries to create and manage objects and parallelism. The *Plaid* runtime is responsible for managing states, objects and *Java* interoperability. The

ÆMINIUM runtime (cf. Section 5.5) is responsible for managing the execution of the tasks generated by the program.

5.3 The Plaid Language Compiler

This section provides a high-level overview of the Plaid compiler infrastructure. The goal of this section is to provide enough information to the reader to understand the changes and extension that have been made to implement ÆMINIUM. By no means is this section supposed to be an exhaustive presentation of the whole Plaid language infrastructure. The source code is publicly available at the Plaid Google Code repository [78].

5.3.1 General Compiler Architecture

The Plaid compiler has been written in *Plaid* itself. Some of the core design was inspired by various other compiler designs (such as Polyglot [73] and the *Java* compiler) . The *Plaid* compiler is a source-to-source compiler i.e, the compiler reads *Plaid* source code and generates *Java* source files. Those *Java* source files are then compiled to *Java* bytecode. The overall compiler architecture is shown in 5.2 on the following page. The compiler reads the specified *Plaid* source files and generates an internal compiler job for each file. Each job maintains a list of operations (called *passes*) to track which operations are left to perform for a particular job. There exist two different kinds of passes. The first kind implements one specific functionality that needs to be performed for the job (e.g., the *Parsing Pass* is responsible to parsing the source code into an *abstract syntax tree* (AST)). The second kinds of passes represents a synchronization operation across all jobs ,e.g., the *Parsing Barrier Pass* which forces all jobs to finish the *Parsing Pass* before executing the next pass on their list. We need the synchronization passes to ensure that we precompute all necessary information for later passes. The following sections describe each pass individually.

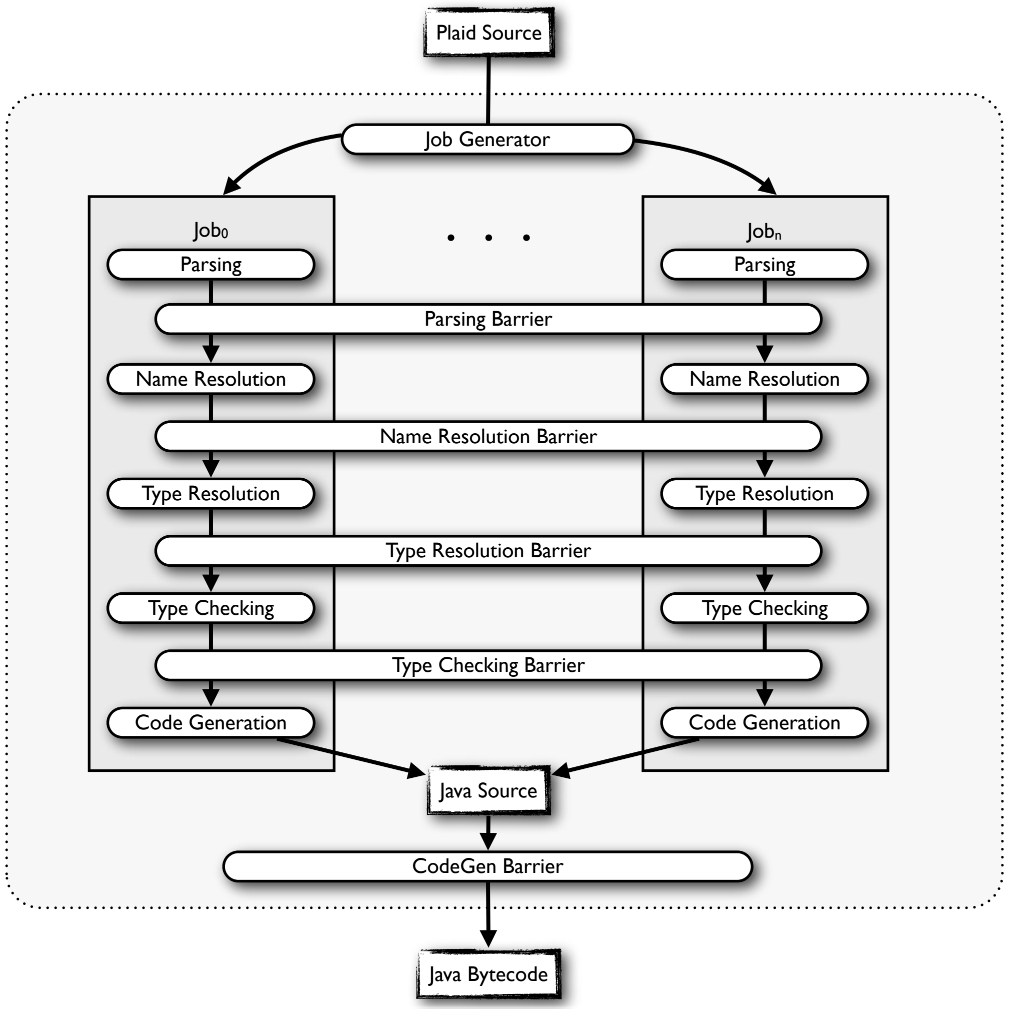
5.3.2 Parsing Pass

The parsing converts the *Plaid* source code into an *Abstract Syntax Tree* (AST). This transformation does not alter, modify or extend the information represented in the source code, but solely represents a one-to-one mapping.

5.3.3 Name Resolution Pass

The name resolution pass resolves top-level constructs and variables. By resolving we mean that we associating *symbols* with all variables in the source code and all globally resolvable identifiers (e.g., state

Figure 5.2: ***Plaid*** **Compiler Architecture. The dashed box identifies the** ***Plaid*** **compiler boundaries.**



names). Symbols represent an uniform abstraction of low level information (e.g., type of element, mutability, etc.). This pass additionally resolves fully qualified names (FQN, e.g., “`plaid.lang.Boolean`”) and additionally resolves *Java* static fields and methods.

5.3.4 Type Resolution Pass

The type resolution pass uses the symbol information and populates the AST with type information. The *Plaid* syntax is ambiguous without type information. *Plaid* supports first class function (i.e., lambdas) which can be storred in object fields. There it is syntactically not clear if we apply some arguments to a lambda function stored in a field or if we call a method. Listing 5.4 on the next page highlights this problem. In

Listing 5.4: Plaid Syntax Ambiguity

```
state Foo {
    val fn () → void m1 = fn () ⇒ {};

    method void m2() [immutable Foo this] { }

    method void bar() [immutable Foo this] {
        this.m1();
        this.m2();
    }
}
```

the `bar` method it is not clear if `this.m1()` (or `this.m2()`) is a method call or an application of a function stored in a field. Because the parser cannot determine which one is the correct answer both cases are parsed as function applications. During the type resolution pass we eventually will determine if the target of the application is actually a function or a real method. Therefore the type resolution pass will convert all applications that target a method into proper method calls.

5.3.5 Type Checker Pass

The type checking pass uses the type information generated by the type resolution pass to perform the actual type checking of the code. The *Plaid* type system is not part of this dissertation and we refer the interested reader to [72] for detailed information regarding the full type system. Plaid runs on the Java Virtual Machine and supports interoperability with Java, meaning it is possible to create and use any Java class in a Plaid program. Because Java code does not naturally support permissions Plaid's type checker needs to be extended to automatically deduct appropriate permissions for Java code (e.g., always assume the most conservative permission possible). As of this writing this Java interoperability feature is still work in progress. To allow the usage of Java code in Plaid programs the compiler supports a special annotation called `@sequential`. Annotating a method with this annotation disables the typechecking for its body and therefore allows the usage of Java code inside. The main usage for this annotation is the creation of wrapper states which encapsulate Java objects and methods and provide a permission annotated interface to the Plaid compiler.

5.3.6 Code Generation Pass

The code generator pass translates the annotated AST into *Java* source code. This source code is compiled and run against the *Plaid* runtime system.

The Plaid runtime is a *Java* library which provides pre-defined data types and helper routines to create and initialize objects. Plaid does not support primitive types like for instance *Java* does. This means that in Plaid everything is an Object (like in *Smalltalk*). As we will see in Chapter 6 this approach has an effect on the overall performance of Plaid programs. We omit the exact details of the Plaid runtime library as they are not part of this thesis and not important for the overall ÆMINIUM implementation.

5.4 ÆMINIUM Inside Plaid

5.4.1 Extensions to Type Checker Pass

ÆMINIUM leverages *Plaid*'s type checker which handles the permission flow and checking of the code. Therefore our main tasks were:

Dependent Typesystem To support data groups we extended Plaid's type system to support a simple form of dependent types [77]. The implementation follows the approach described in [77]. To fully support data groups we had to add minor changes to the type checker to track data group permissions additionally to access permissions.

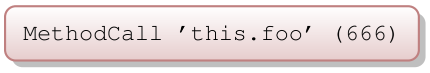
Dependency Computation We extended the Plaid type checker to generate a dependency graph based on the permissions flow according to the sequential order in which the type checker performs its operations.

Optimization Using the generated dependency graph we perform multiple optimization. The first kind of optimizations we perform tries to maximize parallelism by finding available parallelism based on the permission flow. The second optimization we perform is the clustering of operations into tasks. This operation tries to be as aggressive as possible to reduce the number of created tasks while maintaining as much concurrency as possible.

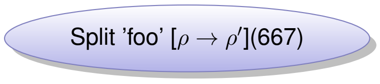
Code Generation We needed to update the code generation to support the unique ÆMINIUM constructs and to generate parallel code.

Dependency Information Graph Representation

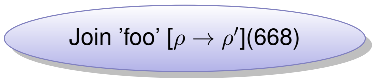
For further processing we generate a dependency graph containing all the permission flow dependencies (for both permission kinds, access permission and data group permissions). The graph $G_\delta = (V_\delta, E_\delta)$ consists of a non-empty set of dependency nodes ($V_\delta = \overline{\delta}$) and edges ($E_\delta = \{(\delta, \delta') : \delta \in V_\delta \wedge \delta' \in V_\delta\}$). We differentiate the following kind of dependency nodes:



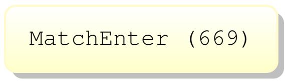
AST nodes are simple wrappers to represent actual operations of the AST (e.g., method calls, field assignments, etc). We represent AST nodes as light red boxes with rounded corners. We highlight method calls by drawing their border slightly thicker than the average AST node. The label of AST nodes starts with the identification of which AST node it represents, followed by concrete information (e.g., which method we call, the specific variable we read, etc). At the end we have an *unique* identifier in parenthesis. We use this identifier to uniquely identifier each node in the graph.



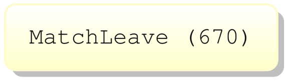
Split nodes represent a *split* operation of a permission performed by the type checker. We represent all nodes representing permission operations (i.e, splits and joins) as light blue ellipses. The label starts with the *Split* to identify the node as a split node, followed by the variable we perform the split operation on. Next we have the actual split operation encoded between square brackets. We abbreviate the different permissions with their initial character (i.e., **I**mmutable, **U**nique and **S**hared). We write $\rho \rightarrow \rho'$ when we split permission ρ to ρ' (e.g., unique $\rightarrow$ immutable). At the end we have an unique identifier for the node. We only encode the split off permission and not the residual permission as it easily computed given the available information. Split nodes depend on the last usage of the symbol it is splitting because this is the place were the original permission came from.



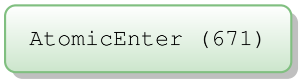
Join nodes represent a *join* operation of a permission performed by the type checker. The graphical representation of join nodes is similar to split nodes with two distinct differences. The first difference is that we start the label with *Join* instead of *Split*. The second difference is in the semantics of the permission operation. In the case of join nodes, $\rho \rightarrow \rho'$ specifies a join operation of ρ into ρ'. We encode only one of the input permissions as the other permission can be inferred by the first input permission and the resulting permission. Join nodes depend on the last usage of the symbol it is joining and the node which represents the residual permission created by the split operation.



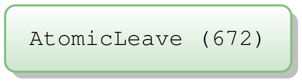
Match Enter Nodes represent the start of a `match` expression. We represent the start of a `match` block with a yellow rectangle with rounded corner. The label consists of the *MatchEnter* and the unique identifier in parenthesis. A *MatchEnter* node depends on all last usages of the symbols used in its cases and condition.



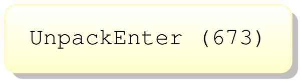
Match Leave Nodes represent the end of a match block. We represent the end of a `match` block with a yellow rectangle with rounded corner. The label consists of the *MatchLeave* and the unique identifier in parenthesis. The *MatchLeave* node depends on all last usages of the symbols used in its cases and condition of the corresponding `match` block.



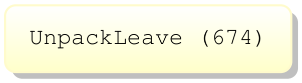
Atomic Enter Nodes represent the start of an `atomic` block. We represent the start of an `atomic` block with a green rectangle with rounded corner. The label consists of the *AtomicEnter* and the unique identifier in parenthesis. The *AtomicEnter* node depends on all last usages of the symbols used in its body expression and to all specified data groups symbols.



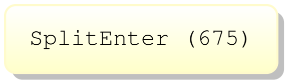
Atomic Leave Nodes represent the end of an `atomic` block. We represent the start of an `atomic` block with a green rectangle with rounded corner. The label consists of the *AtomicLeave* and the unique identifier in parenthesis. The *AtomicLeave* node depends on all last usages of the symbols used in its body expression of the corresponding `atomic` block.



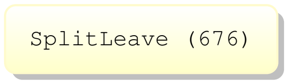
Unpack Enter Nodes represent the start of a `unpackInnerGroups` block. We represent the start of a `unpackInnerGroups` block with a yellow rectangle with rounded corner. The label consists of the *UnpackEnter* and the unique identifier in parenthesis. The *UnpackEnter* node depends on all last usages of the `this` symbol.



Unpack Leave Nodes represent the end of a `unpackInnerGroups` block. We represent the end of a `unpackInnerGroups` block with a yellow rectangle with rounded corner. The label consists of the *UnpackLeave* and the unique identifier in parenthesis. The *UnpackLeave* node depends on all last usages of the inner data groups symbols used of the body expression of the corresponding `unpackInnerGroups` block.

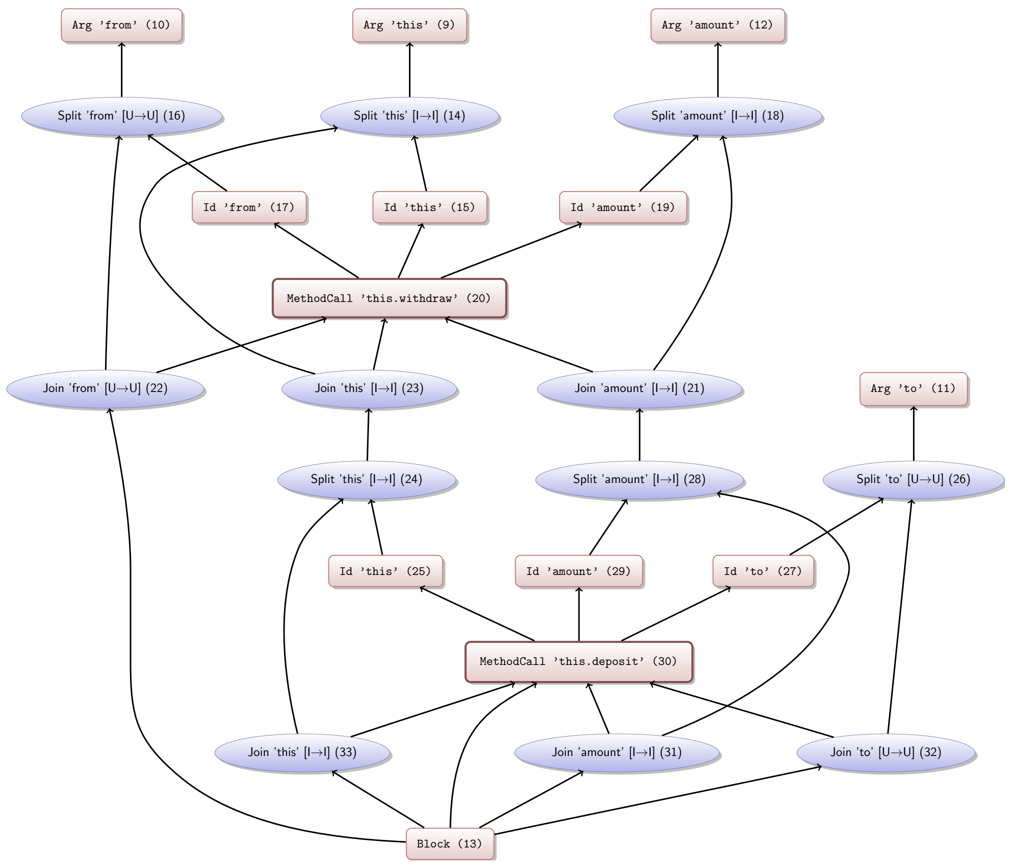


Split Enter Nodes represent the start of a `split` block. We represent the start of a `split` block with a yellow rectangle with rounded corner. The label consists of the *SplitEnter* and the unique identifier in parenthesis. The *SplitEnter* node depends on all last usages of the symbols used in its body expression and to all specified data groups symbols.



Split Leave Nodes represent the end of a `split` block. We represent the end of a `split` block with a yellow rectangle with rounded corner. The label consists of the *SplitLeave* and the unique identifier in parenthesis. The *SplitLeave* node depends on all last usages of the symbols used in its body expression and all specified data groups symbols.

Figure 5.3: Plaid Bank Transfer Example Revisited



```
state BankOperations {
    method void withdraw(unique BankAccount account, immutable Integer amount)
                         [immutable BankOperations this] { }

    method void deposit(unique BankAccount account, immutable Integer amount)
                        [immutable BankOperations this] { }

    method void transfer(unique BankAccount from,
                         unique BankAccount to,
                         immutable Integer amount) [immutable BankOperations this] {
        this.withdraw(from, amount);
        this.deposit(to, amount);
    }
}
```

Dependency Information Graph Example

Figure 5.3 revisits the transfer examples from Chapter 1. The main changes of the example are that we use actual Plaid syntax and follow Plaid's object oriented approach. We use a `BankOperation` state to group

Table 5.1: Parallelizing Peephole Optimizations for

Name	Description
Chained Splits	Simplifies chains of split nodes introduced by binary permission split rules.
Chained Joins	Simplifies chains of split nodes introduced by binary permission split rules.
Unique Join/Split	Removes unnecessary split/join operations which split which slit nothing off a unique permission.
Symmetric Join/Split	Transforms sequential dependencies to symmetric permissions into parallel dependencies.

those helper methods together. Figure 5.3 on the previous page show the dependency graph inferred by the compiler. This graph is slightly more busy than the idealized graph shown in Figure 5.3 on the preceding page. First, this is because the graph contains all the information from the source code (e.g., variables reads). Second, in addition to the parameters, we track the permission of the receiver object (i.e., `this`). As described in Section 5.4.1 the graph represents the sequential execution of the type checker. Following the `this` permission you can see that the second method call to `this.deposit` (30) depends on the first method call to `this.withdraw` (20) through the permission flow of the receiver ($30 \rightarrow 25 \rightarrow 24 \rightarrow 23$) and the amount argument ($30 \rightarrow 29 \rightarrow 28 \rightarrow 21 \rightarrow 20$).

This example clearly shows the eager permission joining approach of the current Plaid typechecker implementation. By eager we mean that the type checker joins permissions back as soon as they become available, instead of a lazy approach where permissions are only merged back when it is required. The next section describes how we can enhance the available parallelism in such an eager dependency graph by following the rules defined in Section 3.1.

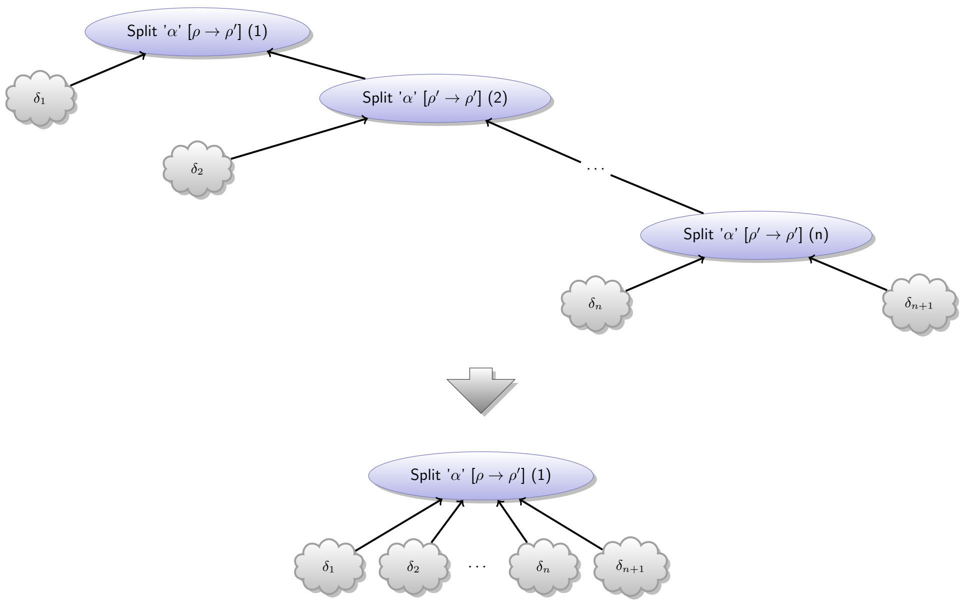
5.4.2 ÆMINIUM Parallelizing Pass

The ÆMINIUM parallelizing pass runs directly after the type checking pass and transforms the sequential dependency graph inferred by the type checker into a parallel version by applying multiple *peephole optimizations* [65]. A peephole optimization searches for specific patterns inside generated code (in our case the 'code' is the dependency graph) and replaces those patterns by a simpler or more efficient one. The following sections explain each performed optimization and Table 5.1 provides a short summary.

Simplification of Chained Splits

Type checking follows a bottom up approach. This leads to cases where multiple subsequent permissions can be split off the same variable before they get merged back. A simple example of such a case would the type checking of a method call where the same variable is passed multiple times as parameter to the call. This

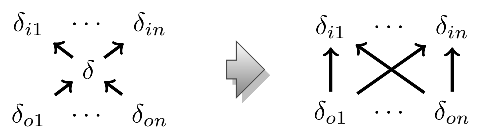
Figure 5.4: Chained Split Block Optimization



```
Input: $G = (V, E)$ dependency graph
Output: $G' = (V', E')$ dependency graph without split chains
begin
    $(V', E') \leftarrow (V, E)$ ;
    foreach $\delta \leftarrow V$ do
        if $\delta$ is split node then
            if $\delta$.permIn == $\delta$.permOut then
                if DEP($\delta$) is split node then
                    $(V', E') \leftarrow$ DELETE_NODE($\delta, (V', E')$) ; /* see Figure 5.5 on the
                    following page */
                end
            end
        end
    end
    return $(V', E')$;
end
```

chaining of permission splits is unnecessary and can be optimized. Instead of having a binary split node and building chains of them we simply merge those nodes to create one n-ary split node. Figure 5.4 shows this operations as graph and algorithm. The graph on top shows a chain of split nodes along with node further defined nodes depending on them ($\delta_1, ..., \delta_{n+1}$). The optimization is applied locally to individual nodes. For every node in the graph the algorithm checks whether the current node is a split node. If it is a split node

Figure 5.5: Node Delete Operation


δ_{i1} $\cdots$ δ_{in}
δ
δ_{o1} $\cdots$ δ_{on}
δ_{i1} $\cdots$ δ_{in}
δ_{o1} $\cdots$ δ_{on}

Deleting a node δ from the ÆMINIUM dependency graph simply removes the node from the graph and makes all nodes which dependent on him ($\delta_{o1}, \dots, \delta_{on}$) depend on all the nodes the removed node depended on ($\delta_{i1}, \dots, \delta_{in}$). The algorithm is shown below.

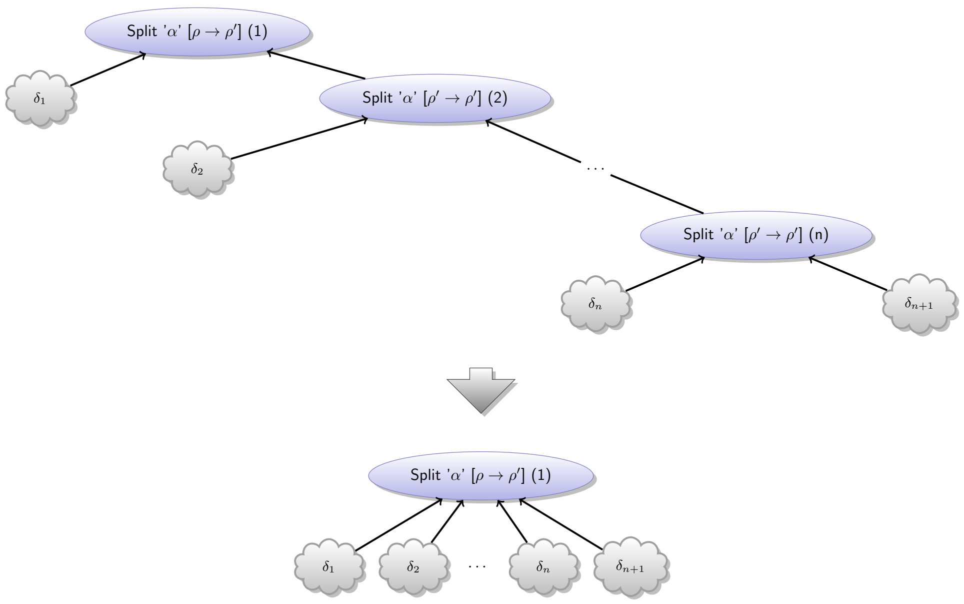
```
Input: $\delta$ the node to remove
Input: $G = (V, E)$ dependency graph
Output: $G' = (V', E')$ dependency graph without $\delta$
begin
    $(V', E') \leftarrow (V, E)$ ;
    $V' \leftarrow V' \setminus \delta$ ;                               /* remove node from nodes */
    foreach $\delta_{dep} \in \{\delta_{dep} : (\delta, \delta_{dep}) \in E'\}$ do
        foreach $\delta_{rdep} \in \{\delta_{rdep} : (\delta_{rdep}, \delta) \in E'\}$ do
            $E' \leftarrow E' \setminus (\delta_{rdep}, \delta)$ ;              /* remove backwards dependency */
            $E' \leftarrow E' \setminus (\delta, \delta_{dep})$ ;               /* remove forwards dependency */
            $E' \leftarrow E' \cup (\delta_{rdep}, \delta_{dep})$ ;                   /* add new dependency */
        end
    end
    return $(V', E')$ ;
end
```

it will check if the input permission is the same as the output permission and if the current node depends on another split block. If all conditions hold the algorithm deletes the current split block from the graph by preserving the dependencies (see Figure 5.5).

Simplification of Chained Joins

Similar to chained splits the type checker can generate chained join nodes to merge those chained spitted permissions back to the original permission. Therefore the same principle as for chained splits can be applied and we can reduce those chains to a single join node. Figure 5.6 on the next page shows the approach and the algorithm. The algorithm operates on a individual nodes. It first filters out all joins nodes. Then for every join node the algorithm checks whether it joins the input permission into the same kind of permission. If it does it checks if there is any other join node depending on him. If all conditions hold the algorithm deletes the current node by preserving the dependencies.

Figure 5.6: Chained Join Block Optimization



Input: $G = (V, E)$ dependency graph
Output: $G' = (V', E')$ dependency graph without split chains

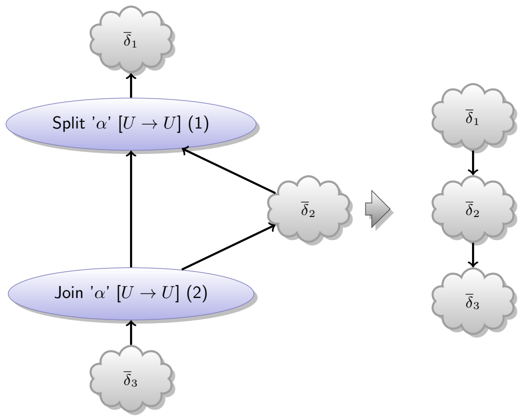
```
begin
    $(V', E') \leftarrow (V, E)$;
    foreach $\delta \leftarrow V$ do
        if $\delta$ is join node then
            if $\delta$.permIn == $\delta$.permOut then
                if $\exists \delta' : (\delta', \delta) \in E \wedge \delta' is\ join\ node$ then
                    $(V', E') \leftarrow$ DELETE_NODE($\delta$, $(V', E')$) ; /* see Figure 5.5 on the
                    facing page */
                end
            end
        end
    end
    return $(V', E')$;
end
```

Simplification of Unique Split/Join Sequences

Following the type checking rules the compiler splits of the `unique` permission from a variable and leaves a `none` permission associated with the variable. Later when the `unique` permission is returned to the variable the type checker merges the incoming `unique` permission with the available `none` permission.

This is a typical scenario for method calls where the permission gets conceptually splits off from the variable and later (after the method call) merged back. Figure 5.7 shows the scenario on the left hand side where unique permission from α has been split of to satisfy the operations $\overline{\delta}_2$. Figure 5.7 also show the algorithm to implement this operation which simply removes those unnecessary nodes.

Figure 5.7: Simplify Unique Join/Split sequences



Input: G = (V,E) dependency graph
Output: G' = (V', E') dependency graph without split chains

```
begin
    $(V', E') \leftarrow (V, E)$ ;
    foreach $\delta \leftarrow V'$ do
        if $\delta$ is join node then
            if $\delta$.permIn == $\delta$.permOut == unique then
                if $\exists \delta' : (\delta, \delta') \in E \wedge \delta' is\ split\ node$ then
                    $E' \leftarrow E' \backslash (\delta, \delta')$ ;
                    $(V', E') \leftarrow$ DELETE_NODE($\delta$, $(V', E')$) ;        /* see Figure 5.5 */
                    $(V', E') \leftarrow$ DELETE_NODE($\delta'$, $(V', E')$) ;       /* see Figure 5.5 */
                end
            end
        end
    end
    return $(V', E')$;
end
```

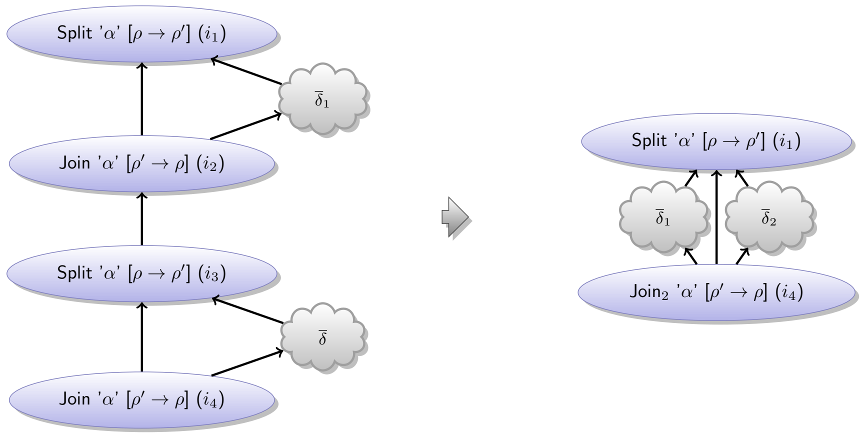
Figure 5.8: Remove Symmetric Join/Split

```
Input: G = (V,E) dependency graph
Output: G' = (V', E') dependency graph without split/join patterns
begin
    $(V', E') \leftarrow (V, E)$ ;
    foreach $\delta \leftarrow V'$ do
        if $\delta$ is split node then
            if SIZE(DEPS($\delta$)) == 0 then
                $\delta' \leftarrow$ DEPS($\delta$).get(0) ;
                if $\begin{pmatrix} \delta' \ is \ split \ node \\ PERM\_OUT(\delta) == PERM\_IN(\delta') \\ SYM\_PERM(PERM\_IN(\delta')) \\ INATOMIC(\delta) == false \end{pmatrix}$ then
                    if HAS_LOWER_JOIN($\delta$) then
                        $\delta'' \leftarrow$ findLowerJoin($\delta$) ;
                        $(V', E') \leftarrow$ fixSymetricJoinSplitWithLowerJoin($\delta, \delta', \delta'', (V', E')$);
                    else
                        $(V', E') \leftarrow$ fixSymetricJoinSplitWithoutLowerJoin($\delta, \delta', (V', E')$) ;
                    end
                end
            end
        end
    end
    return $(V', E')$;
end
```

Simplification of Symmetric Join/Split Sequences

As described in Section 3.1 we define that operations which only overlap in symmetric permissions can be executed in parallel. The current version of the type checker implements a greedy approach for merging permissions back. For every operation the greedy approach splits off the required permissions and joins them back as soon as they become available again (i.e., the operation completes). This leads to the problem that if two operations require a symmetric permission the type checker creates unnecessary dependencies. For instance in Figure 5.3 on page 55 the `immutable` permission of the `amount` parameter is first split into two immutable permissions (node 18), one for the method call to withdraw and one to keep around. Because of the greedy approach the join the split off permission as soon as the method call is over back with the `immutable` permission (node 21). Right after we joined the permissions back we split the permission again to satisfy the permission requirements of the method call to `deposit` (node 30). These extra dependencies are only necessary in the case both operations require different symmetric permissions. If for instance one the

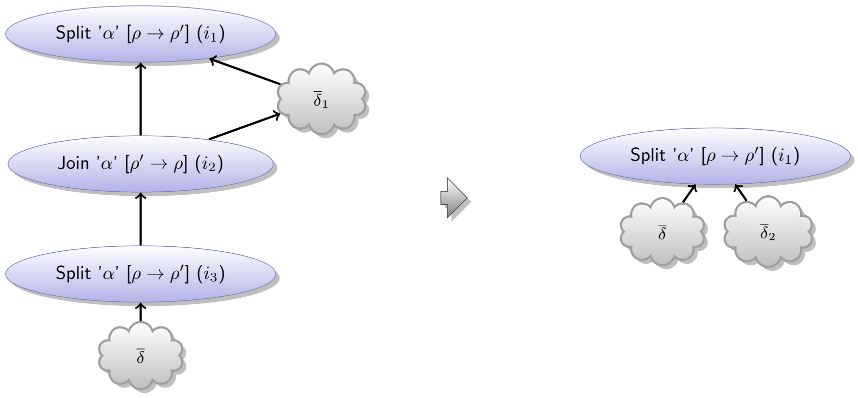
Figure 5.9: Symmetric Join/Split Optimization



```
Input: $\delta_{s2}$ – the split of the join/split pattern
Input: $\delta_{j1}$ – the join of the join/split pattern
Input: $\delta_{j2}$ – the join which merges the permissions of $\delta_{s2}$ back
Input: $G = (V, E)$ – dependency graph
Output: $G' = (V', E')$ – dependency graph without join/split pattern for $\delta_{j1}, \delta_{s1}$
begin
    $(V', E') \leftarrow (V, E)$ ;
    $\delta_{s1} \leftarrow \mathsf{SPLIT\_OF}(\delta_{j1})$ ;
    $E' \leftarrow E' \backslash (\delta_{j1}, \delta_{s1})$ ;
    $E' \leftarrow E' \backslash (\delta_{j2}, \delta_{s2})$ ;
    for $\delta \in \{\overline{\delta'} : (\delta_{j1}, \delta') \in E'\}$ do
        $E' \leftarrow E \backslash (\delta_{j1}, \delta)$ ;
        $E' \leftarrow E \cup (\delta_{j2}, \delta)$ ;
    end
    for $\delta \in \{\overline{\delta'} : (\delta', \delta_{s2}) \in E'\}$ do
        $E' \leftarrow E \backslash (\delta, \delta_{s2})$ ;
        $E' \leftarrow E \cup (\delta, \delta_{s1})$ ;
    end
    $E' \leftarrow E' \cup (\delta_{j2}, \delta_{s1})$ ;
    $V' \leftarrow V' \backslash \{\delta_{s2}, \delta_{j1}\}$ ;
    $E' \leftarrow E' \backslash (\delta_{s2}, \delta_{j1})$ ;
    return $(V', E')$;
end
```

first operation requires a `shared` permission while the second one requires an `immutable` permission we need to merge first back into an `unique` before we can split them into a different kind of permission again.

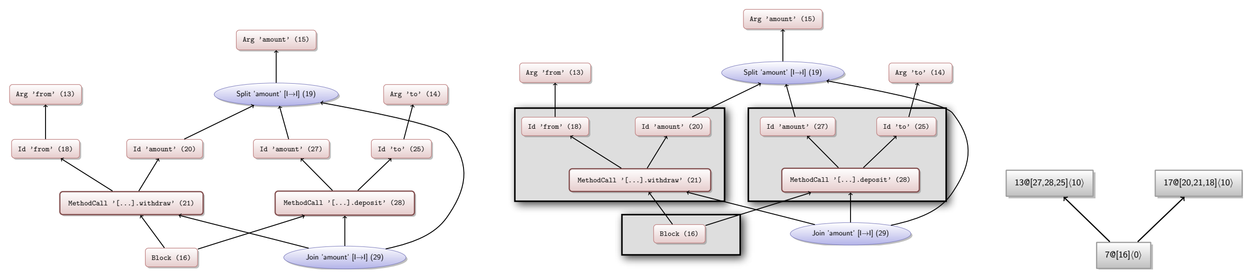
Figure 5.10: Symmetric Join/Split Without Lower Join Optimization



```
Input: $\delta_{s2}$ – the split of the join/split pattern
Input: $\delta_{j1}$ – the join of the join/split pattern
Input: $\delta_{j2}$ – the join which merges the permissions of $\delta_{s2}$ back
Input: $G = (V, E)$ – dependency graph
Output: $G' = (V', E')$ – dependency graph without join/split pattern for $\delta_{j1}, \delta_{s1}$
begin
    $(V', E') \leftarrow (V, E)$ ;
    $\delta_{s1} \leftarrow \mathsf{SPLIT\_OF}(\delta_{j1})$ ;
    $E' \leftarrow E' \backslash (\delta_{j1}, \delta_{s1})$ ;
    $E' \leftarrow E' \backslash (\delta_{j2}, \delta_{s2})$ ;
    for $\delta \in \{\overline{\delta'} : (\delta_{j1}, \delta') \in E'\}$ do
        $E' \leftarrow E \backslash (\delta_{j1}, \delta)$ ;
    end
    for $\delta \in \{\overline{\delta'} : (\delta', \delta_{s2}) \in E'\}$ do
        $E' \leftarrow E \backslash (\delta, \delta_{s2})$ ;
        $E' \leftarrow E \cup (\delta, \delta_{s1})$ ;
    end
    $V' \leftarrow V' \backslash \{\delta_{s2}, \delta_{j1}\}$ ;
    $E' \leftarrow E' \backslash (\delta_{s2}, \delta_{j1})$ ;
    return $(V', E')$;
end
```

To solve this issue we want to detect such unnecessary join/split patterns and eliminate them such that both operations can operate in parallel. Figure 5.9 on the facing page shows how we remove those inner join/split nodes and reorganize the graph so that we initially split multiple symmetric permission off the original permission and execute the operations in parallel. Figure 5.8 on page 61 shows the algorithm to detect these join/split patterns. For all split nodes we check if it depends on a join node and if the join node

Figure 5.11: Task Builder Approach



(a) Dependency Graph (b) Dependency Graph with Task Overlay (c) Task Graph

merges to the same kind of symmetric permission the split block splits to and that we are not in a sequential context (i.e., inside an atomic block). If the conditions applies we found a join/split case that can be optimized. Before we can perform the transformation we have to check for one of two cases. The first case, as shown in Figure 5.9 on page 62, where we have a second join block merging the split off permission back together. And a second case, shown in Figure 5.10 on the previous page, where we do not have a later join of the split permission (e.g., in the case we split of a permission that is returned by the method). The algorithm removes the inner join/split nodes and connect the operations to the upper split and the lower join node.

5.4.3 Task Builder Pass

Generating a new task for every node in the dependency graph (i.e., a single operation) is prohibitive expensive because the ratio of actual work per task compared to the task creation and execution overhead is too small. Therefore we use developed the *Task Builder Pass* who goals it is to combine multiple operations into bigger tasks. Figure 5.11 shows the basic idea. The task builder takes as input a dependency grap (see Figure 5.11a) and then computes the which operations can be mapped into the same task without loosing parallelism. Figure 5.11b shows the input graph with the task clustering. The task builder outputs a task graph only containing consisting of only tasks (see Figure 5.11b).

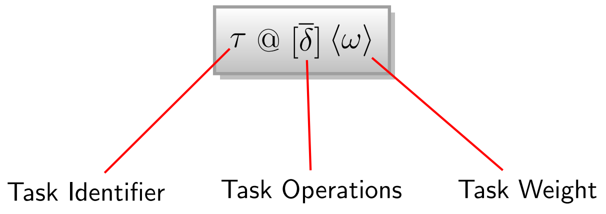
The general idea idea behind the task builder is called *edge zeroing*. The task builder uses a certain cost metric to estimate the overall execution costs of a specific dependency graph. The algorithm then analyses for every edge in the dependency graph how removing the edge and merging the connecting nodes affects the execution cost of the whole graph. If the execution cost do not increase the task builder performs the actual removal of the current edge from the graph and merges the connecting nodes together. The following sections explain the task builder in more details.

ÆMINIUM Task Graph Representation

An ÆMINIUM task graph ($G_\tau = (V_\tau, E_\tau)$) is a tuple consisting of a non-empty sets of tasks ($V_\tau = \overline{\tau}$) and edges ($E_\tau = \{(\tau, \tau') : \tau \in V_\tau \wedge \tau' \in V_\tau\}$) between those tasks. Without loss of generally we define if $(\tau, \tau') \in E$ then task τ *depends* on task τ'.

In our illustrations we represent tasks as a gray rectangle (▢) with a label. Figure 5.12 shows that a task label is composed of three pieces of information. The first information is the unique *task identifier* τ. The next information encoded on the label are the operations which are performed by the task which are enclosed by square brackets. We use δ to represent an arbitrary dependency information operation and use $\overline{\delta}$ to identify a non empty collection of such operations. We write $\overline{\delta}_\tau$ to refer to the operations associated with task τ. The last bit of information consists of the tasks weight ω enclosed in angle brackets. We write ω_τ to refer to the weight of task τ. The weight of a task in our setting is an abstract measure for the tasks runtime. If we do not care about the additional information we write τ for $\tau@[\overline{\delta}]\langle\omega\rangle$.

Figure 5.12: Task Notation



Task Simplifier

Before we start to group operations into tasks we simplify the graph by delete all helper nodes (see Figure 5.5 on page 58). Helper nodes are nodes which have no representation in the source code (i.e., split and join nodes) and are solely used to keep track of dependencies via the heap accesses. After removing those helper nodes we create an initial task graph in which every operation gets wrapped in their own task and have the dependencies of the task graph mirror the dependencies of the dependency graph. This results in a task graph which exactly looks as the dependency graph with the exception that the nodes are tasks representing wrapping the individual operations of the dependency graph. We remove all dependencies from the graph for which we have transitive dependencies. This means, for instance, if $A \rightarrow B$ and $A \rightarrow C$ and $C \rightarrow B$ then A depends on B directly via $A \rightarrow B$ and indirectly via $A \rightarrow C \rightarrow B$. We can safely remove the $A \rightarrow B$ and save unnecessary dependencies and synchronization/communication costs.

Task Builder Algorithm

We now have a dataflow graph of operations. Unfortunately the granularity of every operation is rather small while the overhead of executing and synchronizing all those tasks is rather high. This leaves two options, either increasing the granularity level of the operations we perform or decreasing the overhead costs for parallel execution of those tasks. Unfortunately there is a minimum overhead we have to cope with on commodity hardware and software. We therefore focus on the optimizing the task granularity to outweigh the overhead costs while still preserving enough parallelism to achieve performance gains.

One way to increase the granularity is to cluster operations into bigger *tasks*. We developed a task building algorithm (TBA) based on the idea of *Sarkar's Algorithm* (SA, [81]) to cluster a fine grain data flow graph into coarse grain task graph. The main idea of the Sarkar Algorithm is to optimize a data flow graph by merging nodes as long as the merging does not increase the overall runtime. The deviation between our initial TBA algorithm and SA is the fact that SA only computes a mapping of tasks to groups while our algorithm makes this transformation in place and output the optimized task graph directly. The TBA extends the normal graph representation by adding cost functions for nodes and weights for edges. Those costs are used to determine the overall execution cost of the dataflow graph. The fundamental approach of the algorithm is to reduce the commutation time by removing edges and merging the connecting tasks. In particular the algorithm tries to optimize the *parallel execution time* of the dataflow graph. The parallel execution time assumes that we have an infinite amount of parallel processing units available. The parallel execution time is defined to be the longest path from a starting node to an end node in the dataflow graph. We call this longest path the *critical path (CP)* of a weighted data flow graph. The algorithm is shown in Figure 5.13 on the facing page and works as follows:

1. compute the critical path length (CP) for the input graph
2. for every edge in the graph
   (a) compute the critical path length (CP') without the current edge
   (b) if the critical path without the edge did not increase the critical path
      i. update the current critical path length to CP'
      ii. remove edge from graph
      iii. merge nodes which the edge connected.
      iv. simplify newly generated transitive edges

The Sarkar Algorithm, on which our task builder algorithm is based on, was originally designed for mapping whole program dataflow graphs to hardware CPUs. We generate Java source code which uses the Plaid runtime. With all of this running on the JVM abstraction on various platforms it is not eminently clear

Figure 5.13: Task Builder Algorithm

```
Input: $G = (V, E)$ – task graph graph
Output: $G' = (V', E')$ – task graph without join/split pattern for $\delta_{j1}, \delta_{s1}$
begin
    $(V', E') \leftarrow (V, E)$ ;
    $CP \leftarrow$ computeCriticalPath($(V', E')$) ;
    for $(\tau, \tau') \in E'$ do
        $CP' \leftarrow$ computeCriticalPathWithoutEdge($(V', E'), (\tau, \tau')$) ;
        if $CP' \leq CP$ then
            $CP \leftarrow CP'$ ;
            $\tau_{merged} \leftarrow \tau \cup \tau'$ ;
            $V' \leftarrow (V \cup \tau_{merged}) \backslash \{\tau, \tau'\}$ ;
            $E' \leftarrow E' \backslash (\tau, \tau')$ ;
            for $(\tau, \tau_x) \in E'$ do
                $E' \leftarrow (E' \cup (\tau_{merged}, \tau_x)) \backslash (\tau, \tau_x)$;
            end
            for $(\tau_y, \tau') \in E'$ do
                $E' \leftarrow (E' \cup (\tau_y, \tau_{merged})) \backslash (\tau_y, \tau')$;
            end
            $(G', E') \leftarrow$ SIMPLIFY_TRANS_EDGES$(V', E')$ ;
        end
    end
    return $(V', E')$;
end
```

what the exact weights for the each task or edge should be. Similar to Sarkar we could extend our system to use profiling information to get more accurate runtimes for specific methods. Another approach would be to use a whole program analysis or specific type systems extension [24, 80] for more accurate results. One alternative would be to ask the programmer to provide us with the necessary runtime information (see Section 5.4.3). Acar et al. [11] propose with *oracle scheduling* an interesting dynamic approach to the granularity issues. In oracle scheduling the user provides for every function a second function which quickly estimates for a concrete input set how long the function will run. This information can the be used at runtime to decide case by case if a method call is worth spanning off a task or not.

In our first approach we use a simple approach and associated the following weights[1]: we assigned ω_{edge} for all edges, ω_{call} for all method calls and ω_{cheap} for all other operations (e.g., variable read, field assignment). This is not exact because different methods might run differently long. But there is no general way of solving this issue, especially in a static context. Solving this issue would require some abstract

[1]The exact values not important for this analysis, except for the fact that ω_{cheap} is set to 0 to allow aggressive merging of trivial operations.

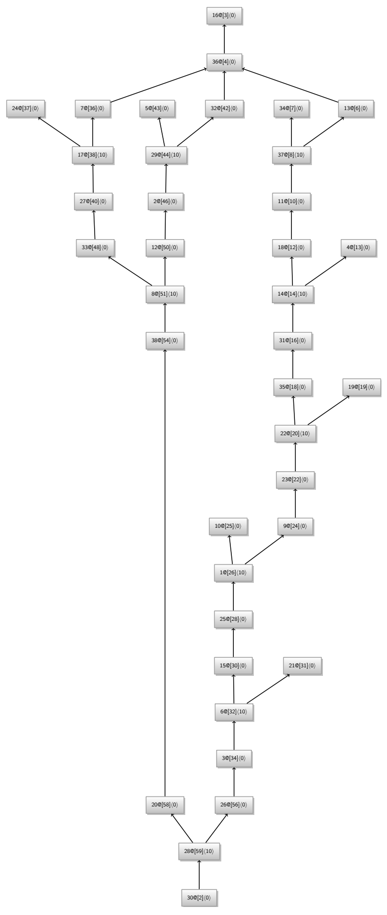
Listing 5.5: Broken Sarkar Example

```
state BrokenSarkarExample {
    method immutable Integer compute() [immutable BrokenSarkarExample this] {
        val immutable Integer a = 0;

        val immutable Integer b1 = a + 1;
        val immutable Integer b2 = b1 + 1;
        val immutable Integer b3 = b2 + 1;
        val immutable Integer b4 = b3 + 1;
        val immutable Integer b = b4 + 1;

        val immutable Integer c = a + 1;

        val immutable Integer d = a + 1;

        val immutable Integer e = c + d;

        b + e
    }
}
```

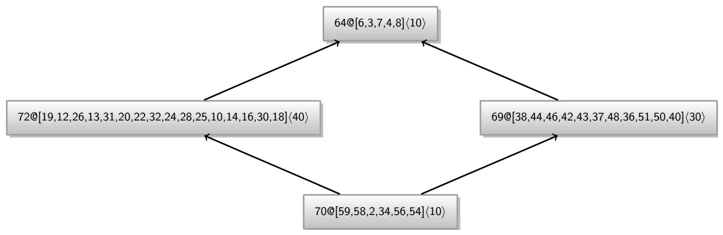
function that could statically determine how long any given method runs. Having such a function would trivially solve the halting problem. Additionally we have to deal with dynamic dispatch, which makes it almost impossible to determine which function will be called at runtime. While implementing and evaluating this first approach it became clear that there was one issue with having fixed node weights. The problem is that the algorithm might optimize parallelism away because the overall runtime of a method is determined by a long critical path.

To make this problem more clear let's consider the example shown in Listing 5.5, specifically designed to highlight the problem. In this example, we can identify three potential parallel paths which depend on the variable `a`. We have the *b-branch* (consisting of `b1`, `b2`, `b3`, `b4` and `b`), the *c-branch* (consisting of `c`) and the *d-branch* (consisting of `d`). We further combine the *c-branch* and *d-branch* into *e-branch*. At the end we combine the results of the *b-branch* with the *e-branch*. Figure 5.14a on the next page shows the compiler generated task graph before we run the task builder algorithm on it. Note the long chain of tasks on the right hand side corresponding to the *b-branch* and the 'Y' shaped tasks on the left forming the *cde-branches*. Assuming that every method call is expensive (remember that in Plaid all operators are transformed into method calls e.g., `a + 1` $\mapsto$ `a.+(1)`) we can see that the CP of the method is `a(b*)`. In particular we see that the *b-branch* has five method call operations while the combined *cde-branches* have a total of three method call operations. Assuming that all method calls have the same weight SA would merge the *cde-branches* into one task, because doing so does not increase the overall runtime as the critical path is not increases. This is the correct behavior when we have the exact weight of each method call, as this approach reduces the number of tasks and communication between those tasks along with the required computational resources. Figure 5.14b shows the output produced for this program for SA. As described earlier we do not have the exact runtime costs and merging the tasks of the *cde-branches* together reduces possible parallelism between the *b-branch* and *c-branch*. In the next section we discuss how we can avoid this loss of parallelism.

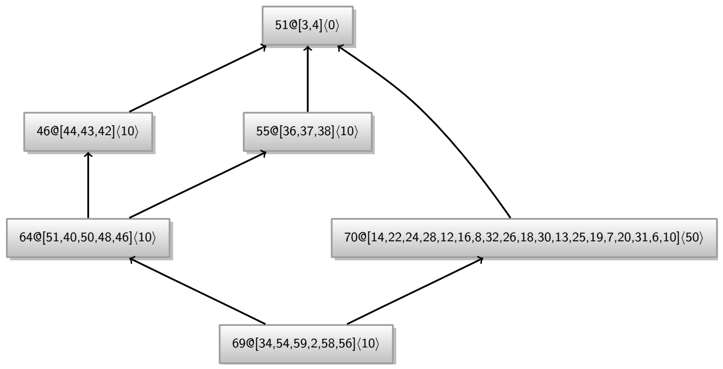
Figure 5.14: Broken Sarkar Example For Basic Sarkar Algorithm



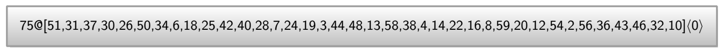
(a) Task Graph Before SA



(b) Task Graph After SA



(c) Task Graph After SA With Local Regression



(d) Task Graph After SA With Local Regression and Cheap Function Calls

Task Builder Algorithm With Local Regression

The previous section discussed the basics of TBA and demonstrated one of its weaknesses for our scenario. To avoid that the TBA optimizes parallelism away we extend the base algorithm to check additionally that

Figure 5.15: Task Builder Algorithm With Local Regression

```
Input: $G = (V, E)$ – task graph graph
Output: $G' = (V', E')$ – task graph without join/split pattern for $\delta_{j1}, \delta_{s1}$
begin
    $(V', E') \leftarrow (V, E)$
    $CP \leftarrow$ computeCriticalPath($(V', E')$)
    for $(\tau, \tau') \in E'$ do
        $CP' \leftarrow$ computeCriticalPathWithoutEdge($(V', E'), (\tau, \tau')$)
        $LR \leftarrow$ minimumPathRegressionForGraphWithoutEdge($(V', E'), (\tau, \tau')$)
        if $CP' \leq CP \wedge LR == false$ then
            $CP \leftarrow CP'$
            $\tau_{merged} \leftarrow \tau \cup \tau'$
            $V' \leftarrow (V \cup \tau_{merged}) \backslash \{\tau, \tau'\}$
            $E' \leftarrow E' \backslash (\tau, \tau')$
            for $(\tau, \tau_x) \in E'$ do
                $E' \leftarrow (E' \cup (\tau_{merged}, \tau_x)) \backslash (\tau, \tau_x)$
            end
            for $(\tau_y, \tau') \in E'$ do
                $E' \leftarrow (E' \cup (\tau_y, \tau_{merged})) \backslash (\tau_y, \tau')$
            end
            $(G', E') \leftarrow$ SIMPLIFY_TRANS_EDGES$(V', E')$
        end
    end
    return $(V', E')$
end
```

we only delete an edge and merge tasks if this does not cause additional effort for any other path in the graph. We therefore check for every node that the removal of an edge does not increases its *minimum path*. The minimum path is the counter part to the maximum path length but using the path with the smallest costs instead of the biggest costs. The usage of the minimal path allows us to retain parallelism while still benefitting from SA merging capabilities. Figure 5.15 shows this extended version of TBA. We marked the extensions in gray. The output graph of this extended algorithm is shown in Figure 5.14c on the preceding page. The graph shows that the algorithm merged all tasks of the *b-branch* into one task on the right, and merged the *cde-branches* into individual tasks in the left, preserving the ‘Y’ shape. This means that we are still capable of executing the *c-branch* in parallel with the *d-branch*.

Cheap Functions

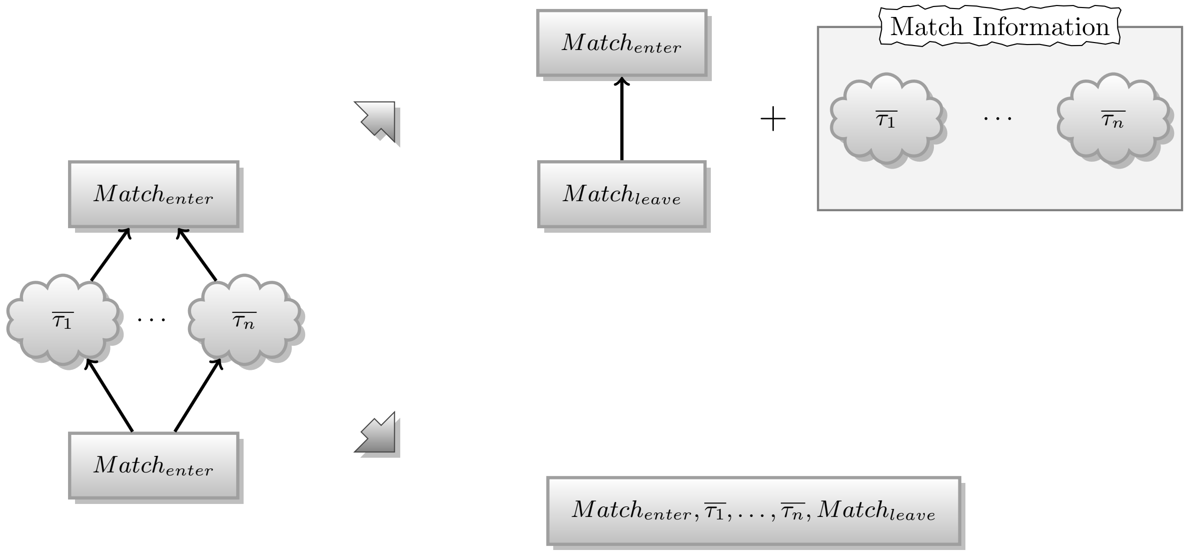
We described in Section 5.4.3 that we use a simply cost model. While we cannot get access to the exact time a method call costs we can do better than just assuming all method calls cost the same. In particular we divide method call costs into two categories. The first category method calls are equivalent to the old method calls, meaning that we do not know the runtime costs and assign a weight of ω_{call}. The second category are the so called cheap method calls for which we assign the ω_{cheap} weight to indicate that executing those method calls is much cheaper than creating tasks for them. To identify such cheap method calls we introduced the `@cheap` annotation for method declarations. Every call to a method with this annotation is assumed to a cheap call. By default we declared all method of basic types (i.e., `String`, `Integer`, etc.) as cheap methods. The compiler checks that methods with cheap annotations are actually cheap (i.e, they have no parallelism inside and only perform cheap operations). We choose an user annotation to allow for a modular design. Explicitly specifying methods as cheap forms a contract and all possible implementations of this method needs to obey this. The cheapness property must be preserved by every overriding method. This allows use to compile agains the definition of a method and later use any possible implementation (which are checked by the compiler to obey the cheap requirement). The cheap annotations allows the user to give additional hints to the compiler. The compiler does not only check for the consistency of the cheap annotation but my also point out methods which should have a cheap annotation. We do not automatically insert cheap annotations because of the modular compilation approach of Plaid. Transparently introducing a cheap annotation at one state can break another sub-classing state which violates the cheap annotation. We therefore insist on the explicit specification of cheap annotations to form an contract.

If we look at the example in Listing 5.5 we can see that all method call in this method are calls to the plus method of the integers. Figure 5.14d on page 69 show the optimized task graph after introducing cheap function annotations for built-in types. The figure shows that the whole graphs has been collapsed into one single task indicating that there is no benefit of parallel execution.

Matching Support for Task Builder Algorithm

Now that we have a basic algorithm for clustering our operations into a manageable set of tasks we have to deal with the remaining features of Plaid. As described in 5.4.1, our dataflow graph also contains control flow edges caused by `match` blocks. We therefore have to make the TBA aware of those different edges. The main problem is that a `match` block represents a choice of possible execution paths at runtime. Because we do not know statically which of the paths are executed we use a *MatchEnter* and *MatchLeave* nodes to represent the start/end of a `match` block. These virtual markers allows use to identify `match` blocks in the graph and help us to differentiate between static dependencies and control flow dependencies only known at runtime. Without this additional nodes and special handling the TBA could easily end up in merging operations of different execution branches into the same task.

Figure 5.16: Match Simplification



The way we deal with match blocks is by recursively optimizing them in a bottom-up approach. Before we apply the TBA we first simplify the match blocks in the task graph. The simplification process finds the top-level match blocks and then recursively optimizes its cases by using the TBA (which recursively simplifies top-level match-blocks and so on). The operation of the `simplifyMatches` function is shown in Figure 5.16. On the left side we see the task graph representing a match block that is supposed to be optimized. As shown in the right hand side of Figure 5.16 the `simplifyMatches` can simplify matches in one of two ways.

In the first, case `simplfiyMatches` removes case tasks from the task graph and connects the match leave task to the match enter task (upper right hand size). The function additionally generates a *match information* ($\mathcal{M} = (\delta_{enter}, \delta_{leave}, \overline{\mathcal{C}})$) object. The *match information* object consists of the dependency information identifying the match enter (δ_{enter}) and match leave (δ_{leave}) and a case information object for each case ($\mathcal{C} = (\overline{\tau_{all}}, \overline{\delta_{depEnter}}, \overline{\delta_{leaveDeps}})$). A case information object tracks all the tasks forming the case task graph ($\overline{\tau_{all}}$), the dependency information of the operations that depend on the beginning of the match block ($\overline{\delta_{enter}}$) and the operations which the match leave depends on ($\overline{\delta_{leave}}$). We keep the match enter and match leave separated so TBA can merge the start or end of a matching block with other operations to reduce the number of tasks. This is the reasons we cannot represent the simplified match block by a simple task. If we would use just a single task and we merged it with other operations, the compiler would not know (without further computation) which operations happened before and which happened after the match block. We therefore keep the start and the end task separated. To avoid that TBA merges this special case we have

Figure 5.17: Task Builder Algorithm With Match Support

```
Input: $G = (V, E)$ – task graph graph to optimize
Output: $G' = (V', E')$ – optimized task graph
Output: $\mathcal{M}$ – Match Case Information
begin
    $(V', E', \mathcal{M}) \leftarrow$ simplifyMatches$(V, E)$
    $CP \leftarrow$ computeCriticalPath$((V', E'))$
    for $(\tau, \tau') \in E'$ do
        $CP' \leftarrow$ computeCriticalPathWithoutEdge$((V', E'), (\tau, \tau'))$
        $MB \leftarrow$ isSimplifiedMatch$(\tau, \tau')$
        $LR \leftarrow$ minimumPathRegressionForGraphWithoutEdge$((V', E'), (\tau, \tau'))$
        if $CP' \leq CP \ \wedge \ LR == false \wedge MB == false$ then
            $CP \leftarrow CP'$
            $\tau_{merged} \leftarrow \tau \cup \tau'$
            $V' \leftarrow (V \cup \tau_{merged}) \backslash \{\tau, \tau'\}$
            $E' \leftarrow E' \backslash (\tau, \tau')$
            for $(\tau, \tau_x) \in E'$ do
                $E' \leftarrow (E' \cup (\tau_{merged}, \tau_x)) \backslash (\tau, \tau_x)$
            end
            for $(\tau_y, \tau') \in E'$ do
                $E' \leftarrow (E' \cup (\tau_y, \tau_{merged})) \backslash (\tau_y, \tau')$
            end
            $(G', E') \leftarrow$ SIMPLIFY_TRANS_EDGES$(V', E')$
        end
    end
    return $(V', E', \mathcal{M})$
end
```

to extend TBA slightly to never remove edges between match enter and leave tasks. Figure 5.17 shows the extension to TBA (changes in gray).

In the second possible simplification that can occur is shown in the lower right hand side of Figure 5.16 on the preceding page. If all cases reduce to a single task then all possible execution path represent a sequential code path. In this case the whole match block and all its cases are collapsed into a single task.

Figure 5.18 on the following page shows the algorithm for `simplifyMatches`. The algorithm works as follows. We iterate over all tasks and check if we found a top-level match block leave task. We identify top-level match blocks by traversing all possible paths from the match leave task upwards and count how many unmatched match open tasks we encounter. An unmatched match open task is a match open task for which we have not seen a corresponding match leave task. In the case of a top-level match we should find exactly one open match enter task (namely the one which corresponds to the current match leave tasks).

Figure 5.18: `simplifyMatches`

```
Input: $G = (V, E)$ – task graph to optimize
Output: $G' = (V', E')$ – optimized task graph
Output: $\mathcal{M}$ – Match Case Information

begin
    $\mathcal{M} \leftarrow \emptyset, \mathcal{C} \leftarrow \emptyset$
    $(V', E') \leftarrow (V, E)$
    for $\tau \in V'$ do
        if IS_TOP_LEVEL_MATCH_BLOCK_LEAVE$(\tau)$ then
            $\tau_{leave}[\delta_{M_{leave}}]\langle\_\rangle \leftarrow \tau$
            $\tau_{enter}[\delta_{M_{enter}}]\langle\_\rangle \leftarrow$ MATCH_ENTER_TASK$(\tau_{leave})$
            $\{\overline{\tau_1}, \ldots, \overline{\tau_n}\} \leftarrow$ findCaseTasks$(\tau)$
            for $\overline{\tau_c} \in \{\overline{\tau_1}, \ldots, \overline{\tau_n}\}$ do
                $V_c \leftarrow \overline{\tau_c}$
                $E_c \leftarrow \{(\tau_c, \tau_c') : \{\tau_c, \tau_c'\} \in V_c \wedge (\tau_c, \tau_c') \in E\}$
                $\overline{\delta_{c_{leave}}} \leftarrow \{\overline{\delta} : \tau'' \in V_c \wedge (\tau_{leave}, \tau@[\overline{\delta}]\langle\_\rangle) \in V\}$
                $\overline{\delta_{c_{enter}}} \leftarrow \{\overline{\delta} : \tau'' \in V_c \wedge (\tau@[\overline{\delta}]\langle\_\rangle, \tau_{enter}) \in V\}$
                $V' \leftarrow V' \backslash V_c$
                $E' \leftarrow E' \backslash E_c$
                $E' \leftarrow E' \backslash \{(\tau', \tau_{enter}) : \tau' \in V_{c_{enter}}\}$
                $E' \leftarrow E' \backslash \{(\tau_{leave}, \tau') : \tau' \in V_{c_{leave}}\}$
                $(V_c', E_c', \mathcal{M}') \leftarrow$ taskBuilder$(V_c, E_c)$
                $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}'$
                $\mathcal{C} \leftarrow \mathcal{C} \cup (V_c', \overline{\delta_{c_{enter}}}, \overline{\delta_{c_{enter}}})$
            end
            if $\forall c \in \mathcal{C} : |x.\overline{\tau_{all}}| == 1$ then
                $\tau_{merge} \leftarrow$ mergeTasks$(\tau_{enter}, \overline{\mathcal{C}.\overline{\tau_{all}}}, \tau_{leave})$
                for $(\tau_{enter}, \tau') \in E'$ do
                    $E' \leftarrow (E' \backslash (\tau_{enter}, \tau')) \cup (\tau_{merge}, \tau')$
                end
                for $(\tau', \tau_{leave}) \in E'$ do
                    $E' \leftarrow (E' \backslash (\tau', \tau_{leave})) \cup (\tau', \tau_{leave})$
                end
                $V' \leftarrow (V' \backslash \{\tau_{enter}, \tau_{leave}\}) \cup \tau_{merge}$
            else
                $E' \leftarrow E' \cup (\tau_{leave}, \tau_{enter})$
                $\mathcal{M} \leftarrow \mathcal{M} \cup (\delta_{M_{enter}}, \delta_{M_{leave}}, \mathcal{C})$
            end
            $\mathcal{C} \leftarrow \emptyset$
        end
    end
    return $(V', E', \mathcal{M})$
end
```

Finding more than one unmatched match enter task indicates that the current match block is nested inside another match block. Once we identified a top-level match block the algorithm computes the a set of task sets, each representing one case of the match block. For each case we completely detach the case sub-task graph from the original task graph, forming smaller independent task graph. We recursively use the TBA to optimize this smaller task graphs. After TBA we generate a case information object and accumulate the match information objects for he recursive TBA applications. After all cases have been optimized we check if all case task graphs consist of exactly one task. If so we collapse the whole match block into one task and substituting it for the match block in the input task graph. If at least one of the cases contains some parallelism we just connect the match leave to the match enter task and generate a match information object for this match block.

Atomic Blocks and the Task Builder Algorithm

After making TBA aware of the matching blocks, the only outstanding feature missing are atomic blocks. By choice we do not support nested concurrency inside atomic blocks. Therefore we do not want to generated any concurrent execution inside atomic blocks. This means that all atomic blocks execute sequentially. Therefore we merge all tasks forming an atomic block (i.e., tasks for the body of the atomic blocks and the atomic block tasks) into one (sequential) task before we apply TBA. This means that atomic blocks look just like normal tasks to the TBA and are handled automatically without having to change the algorithm itself.

5.4.4 ÆMINIUM Code Generation Pass

This section presents the code generation approach of ÆMINIUM. We purposly separated optimizations out into their own sections to focus our presentation on the core idea of the basic code generation and optimization approaches.

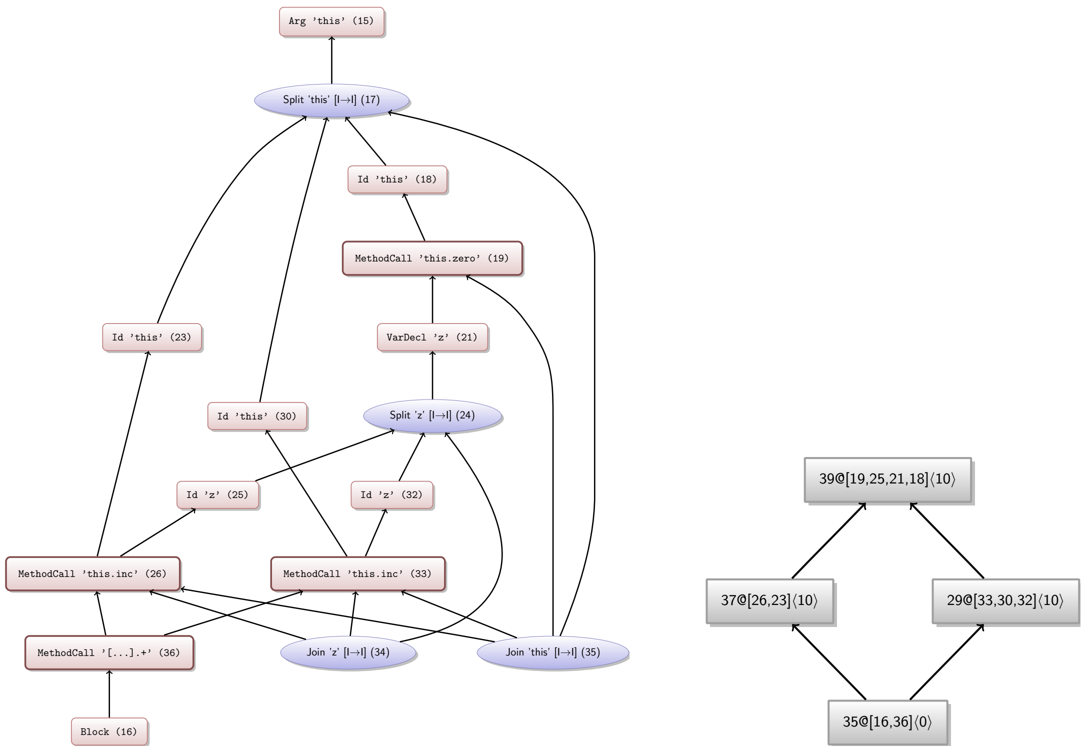
The stating concept of the ÆMINIUM code generation is to schedule only the start tasks of each method. Every task upon its completion, decrements the dependency counter of all tasks which it depends on it. This includes the start tasks. If the dependency counter of a dependent task drops to zero it will be scheduled. This approach leads to an inlined schedule for every method. This approach has the advantage that the runtime system (cf. 5.5) only needs to support an efficient mechanism of executing tasks and does not need to deal with dependency management issues.

To illustrate the idea consider the small example shown in Listing 5.6. For the purpose of this example we focus on the `getTwo` method which first creates a zero which is used to compute two ones which are finally added together. Figure 5.19 on the next page shows the compiler generated dependency and task graph of the `getTwo` method. The task graph looks as expected. First we have task 39 which calls the `zero` method to

Listing 5.6: `getTwo` Code

```
package plaid.examples.codegenExample;

state CodeGenExample {
    method immutable Integer zero() [ immutable CodeGenExample this ] { 0 }

    method immutable Integer inc(immutable Integer value) [ immutable CodeGenExample this ] {
        value + 1
    }

    method immutable Integer getTwo() [ immutable CodeGenExample this ] {
        val immutable Integer z = this.zero();
        this.inc(z) + this.inc(z)
    }
}
```

Figure 5.19: Dependency and Task Graph for `getTwo` Method



(a) Dependency Graph (b) Task Graph

obtain the zero, then we have tasks 37 and 29 depending on task 39 to compute the zero value. At last we have task 35 depending on tasks 37 and 29 to add their computed ones together.

Listing 5.7: `getTwo` **Generated Code**

```
package plaid.examples.codegenExample;

import java.util.ArrayList;
import plaid.fastruntime.PlaidObject;
import plaid.fastruntime.aeminium.Task;

public final class CodeGenExample{
    public PlaidObject zero(PlaidObject this$plaid) { ... }
    public PlaidObject inc(PlaidObject this$plaid, PlaidObject value) { ... }
    public PlaidObject getTwo(PlaidObject this$plaid) {
        /* array for local variables */
        final plaid.fastruntime.PlaidObject[] _ = new PlaidObject[11];

        // TA$K$[0] = 37@[26,23]<10>
        // TA$K$[1] = 29@[33,30,32]<10>
        // TA$K$[2] = 35@[16,36]<0>
        // TA$K$[3] = 39@[19,25,21,18]<10>
        final Task[] TA$K$ = new Task[4];

        /* create tasks */
        TA$K$[0] = new Task(1) {
                @Override public void compute() {
                    _[10] = this$plaid;
                    _[8] = ((plaid.generated.Iinc$1$plaid)_[10].getDispatch()).inc(_[10],_[9]);
                    if ( TA$K$[2].decDepCount() == 0 ) { Task.invokeAll(TA$K$[2]); }
                }
            };
        TA$K$[1] = new Task(1) {
                @Override public void compute() {
                    _[6] = /*z*/_[4];
                    _[7] = this$plaid;
                    _[5] = ((plaid.generated.Iinc$1$plaid)_[7].getDispatch()).inc(_[7],_[6]);
                    if ( TA$K$[2].decDepCount() == 0 ) { Task.invokeAll(TA$K$[2]); }
                }
            };
        TA$K$[2] = new Task(2) {
                @Override public void compute() {
                    _[0] = ((plaid.generated.Iplus$plaid$1$plaid)_[8].getDispatch()).plus$plaid(_[8],_[5]);
                }
            };
        TA$K$[3] = new Task(0) {
                @Override public void compute() {
                        _[3] = this$plaid;
                        _[2] = ((plaid.generated.Izero$0$plaid)_[3].getDispatch()).zero(_[3]);
                        /*z*/_[4] = _[2];
                        _[1] = plaid.fastruntime.Util.unit();
                        _[9] = /*z*/_[4];
                        Collection<Task> nextTa$ks = new ArrayList<E>(2);
                        if ( TA$K$[0].decDepCount() == 0 ) { nextTa$ks.add(TA$K$[0]); }
                        if ( TA$K$[1].decDepCount() == 0 ) { nextTa$ks.add(TA$K$[1]); }
                        Task.invokeAll((Task[]) nextTa$ks.toArray(new Task[nextTa$ks.size()]));
                }
            };
        /* schedule start tasks */
        Task.invokeAll(TA$K$[3]);
        /* return */
        return _[0];
    }
}
```

Listing 5.7 shows the generated code of the codegetTwo method function. In line 12 the method creates a `final` array of `PlaidObjects` to hold all the local variables (replacing the current stack). The array needs to be `final` in order to be captured by the tasks which use *Java's* anonymous class feature. The methods then creates another `final` array of `Tasks` in line 18 to hold the references to the tasks we generate. Starting at line 21 we generate all for tasks of this method (using the array index mapping: $[0] \mapsto 37, [1] \mapsto 29, [2] \mapsto 35, [3] \mapsto 39$). Note that the task constructor takes the dependency count of the created task. This dependency count will be used to determine of the a task is ready to be scheduled or not. In line 55 we finally start our start tasks (i.e., task 39 or array entry [3]) by passing it to the `invokeAll` method. The `invokeAll` method adds the provided tasks to the internal thread pool and only returns once all the passed tasks have completed their execution. Once task 39 is executing its `compute` method (line 42) it first executes all of its operations (i.e., calling the `zero` method and storing its result value in the local variable array). After those operations have been completed the task decrements the reference count of the tasks that depend on it by calling the `decDepCount` method on those tasks. The `decDepCount` method returns the outstanding dependencies of the task. In this example both tasks have only one dependency and therefore the `decDepCount` method returns 0, indicating that both tasks are able to run. Both tasks will be added to the list of next tasks and schedule via another `invokeAll` method call in line 51.

Both tasks 29 and 37 will upon their execution call the `inc` method and store the result in the local variable array. Afterwards both tasks will decrement the outstanding dependency count on their dependent tasks. In this example both tasks will decrement the dependency count of task 35. The last task to decrement task 35 dependency count schedules task 35 while the other task simply completes its execution without any further actions. Task 35 will take the values stored in the local variable array and add them together and stores the result back in the local variable array. After task 35 finishes the "`inc`" task which scheduled it can finish. This allows the completion of task 39 which was waiting for the completion of both "`inc`" tasks. The completion of task 39 allows the schedule call in line 55 to return. After the completion of the task execution the only thing left is the to return the result of the method in line 57.

The generic code generation approach is shown in Figure 5.8. We handle control flow (i.e., `match` blocks) by pulling the matching condition check into all case tasks and execute the associated operations only if the matching condition applies (a more efficient approach to handle control flow is presented in Section 5.4.4). The generic approach consists of the following parts:

① **Extraction of variables.** We extract all locally used variables in an array at the beginning of each method by associating every variable a unique slot. We need this extra level of indirection as we leverage *Java*'s anonymous inner class feature to generate specific tasks. Because anonymous inner classes are only able to catch final variables we need this additional layer of indirection to allows us to set those variables from inside a task without causing an non-initialized error of the *Java* compiler.

Listing 5.8: Pseudo Code for Generalized Selective Task Scheduling

```
public PlaidObject m(...) {
    // create variables
  ① PlaidObject[] _ = new PlaidObject[$\left|\{VarDecl(x) \in \{\overline{\delta} : \overline{\tau @ [\overline{\delta}]\langle\omega\rangle}\}\}\right|$];

    // create task objects
  ② $\forall \tau_i \in \overline{\tau}$ : Task $T_{\tau_i}$ = new Task(|$\mathsf{DEPS}(\tau_i)$|) {
        public void compute() {
            $\mathsf{IS\_CASE\_TASK}(\tau_i) \implies$ if ( $\mathsf{CASE\_MATCH\_COND}(\tau_i)$ ) { $\overline{\delta}_{\tau_i}$ }
            $\neg\mathsf{IS\_CASE\_TASK}(\tau_i) \implies \overline{\delta}_{\tau_i}$
            $\forall \tau' \in \mathsf{RDEPS}(\tau_i)$ : if ( $T_{\tau'}.decDepCount() == 0$) { schedule($T_{\tau'}$); }
        }
    };

    // compute dependencies and schedule tasks
  ③ $\forall \tau_i \in \mathsf{START\_TASKS}(\overline{\tau})$ : schedule($T_{\tau_i}$);

    // wait for 'body' task to finish
  ④ return $T_{\mathsf{BODY\_TASK}(\overline{\tau})}.wait()$;
}
```

② **Task Creation.** For each task in the graph we generated a new task object by using *Java*'s anonymous inner class feature. In the method body of the `compute` method we create the code for of the associated operations. The code generation for those operations follows the sequential code generation rules, with the exception that reads/writes to variables get expanded to deal with the extra level of indirection introduced by the local variable array. We further wrap every task that is part of a control flow path (i.e., `match` block) to check the corresponding control flow decision to see if its code is supposed to execute or not.

③ **Task scheduling.** After all tasks have been created we schedule all start tasks.

④ **Waiting for body task.** The last statement of the function will wait for the task which contains the method body operation to complete and return it's value.

Match Selective Task Scheduling

While functionally correct, running all the tasks for every possible control flow path is prohibitive expensive. Ideally we should execute tasks only if necessary. Listing 5.9 shows and improved version that only executes tasks of control flow paths which are actually executed at runtime using the decomposed match information

Table 5.2: Code Generation HelperFunctions

Name	Description
$\mathsf{DEPS}(\tau)$	Return a collection of tasks on which τ depends on. $\mathsf{DEPS}(\tau) = \{\tau' : (\tau, \tau') \in E\}$
$\mathsf{RDEPS}(\tau)$	Return a collection of tasks on which depend on τ. $\mathsf{RDEPS}(\tau) = \{\tau' : (\tau', \tau) \in E\}$
$\mathsf{BODY\_TASK}(\overline{\tau})$	Return the task which contains the end of the method body operations.
$\mathsf{START\_TASKS}(\overline{\tau})$	Returns the set of tasks without dependencies. $\mathsf{START\_TASKS}(\overline{\tau}) = \{\tau' : \tau' \in \overline{\tau} \land \mathsf{DEPS}(\tau') = \emptyset\}$
$\mathsf{CASE\_MATCH\_COND}(\tau)$	Returns the code to check if the surrounding matching block condition matches the case of this task.
$\mathsf{IS\_CASE\_TASK}(\tau)$	Return $true$ in the case that this task is part of matching case, $false$ otherwise.
$\mathsf{OPEN\_MATCHES}(\tau)$	Returns a set of match descriptor for every *open match enter block* in $\overline{\delta}$. A match descriptor consists of the information which task contains the *open match enter* and *open match leave* information and a set of tasks for the specified cases.
$\mathsf{CASES\_OF\_MATCH}(\mathsf{m})$	Returns a set of the available cases of the match.
$\mathsf{TASKS\_OF\_CASE}(\mathsf{c})$	Return the tasks associated with the given case.
$\mathsf{MATCH\_LEAVE\_TASK}(\mathsf{m})$	Return the task which contains the *open match leave block* information of the given match.
$\mathsf{NON\_CASE\_TASKS\_OF}(\overline{\tau})$	Returns $\overline{\tau'}$ with $\overline{\tau'} \subseteq \overline{\tau}$ where for all τ' is not part of a case path of a match.

generated by the TBA. The core concept it to defer scheduling control flow dependent tasks until we know at runtime which tasks to schedule. In particular the code generation strategy comprises of the following steps:

① **Extraction of variables.** No changes.

② **Task Creation.** We are creating all task at the beginning When generating code we differentiate between tasks containing *open match blocks* and the tasks not containing *open match blocks*. In the later case we simply generate the operations associated with this task. In the first case, identified through the *open match block* we know that we have a non-simplified match block (i.e., its cases are explicitly represented in the data flow graph). The code we generate for this task consists of the following steps:

- ⓶ⓐ First we execute all possible operations that could have been merged into the task.
- ⓶ⓑ For every task we first check if the condition will activate this case at runtime.

Listing 5.9: selective match

```
public PlaidObject m(...) {
    // create variables
  ① PlaidObject[] __ = new PlaidObject[$\left|\{VarDecl(x) \in \{\overline{\delta} : \overline{\tau@[\overline{\delta}]\langle\omega\rangle}\}\}\right|$];

    // create task objects
  ② $\forall\tau_i \in \overline{\tau}$ : Task $T_{\tau_i}$ = new Task(|$\mathsf{DEPS}(\tau_i)$|) {
        public void compute() {
          ②a $\overline{\delta_{\tau_i}}$ // execute possible merged in operations
            $\forall\mathcal{M} \in \mathsf{OPEN\_MATCHES}(\tau_i)$ : {
                $\forall\mathcal{C} \in \mathcal{M}.\overline{\mathcal{C}}$ : {
                  ②b if ( $\mathsf{MATCHES\_COND}(\mathcal{M}, \mathcal{C})$ ) {
                        // schedule case tasks whithout subcase tasks
                      ②c $\forall\tau_c \in \mathsf{START\_TASKS}(\mathcal{C}.\overline{\tau_{all}})$ : schedule($T_{\tau_c}$);
                    }
                }
            }
        }
    };

    // compute dependencies and schedule tasks
  ③ $\forall\tau_i \in \mathsf{START\_TASKS}(\overline{\tau})$ : schedule($T_{\tau_i}$);

    // wait for dependencies of the body task to finish
  ④ return $T_{\mathsf{BODY\_TASK}(\overline{\tau})}$.wait();
}
```

②c If the conditions applies we schedule the tasks of this case. But we do not schedule all the tasks but "top-level" tasks (i.e., tasks which are not part of a match structure inside the case).

③ **Task scheduling.** No changes.

④ **Waiting for body task.** No changes.

To illustrate this approach consider the simple task graph shown in Figure 5.20 on the following page. The graph consists of two tasks (τ', τ'') which represent some arbitrary operations that do not contain any *open match enter block* We then have a task (τ_{enter}) which contains an *open match enter block*. Without loss of generality we show that the corresponding match blocks consists of two cases. We represent the two difference case with task clouds (τ_{c1}, τ_{c2}). We abstract the details of those case tasks away as the approach work recursively inside those cases. At the end we have the *open match leave block* (τ_{leave}) depending on both case tasks to finish.

Figure 5.20: Input graph for selective task example

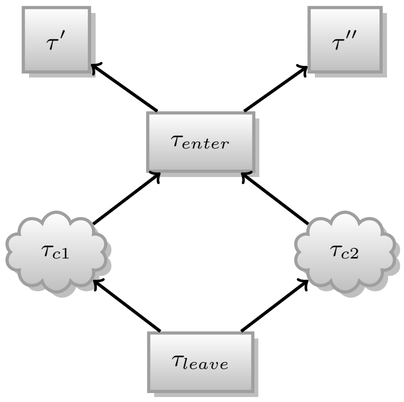
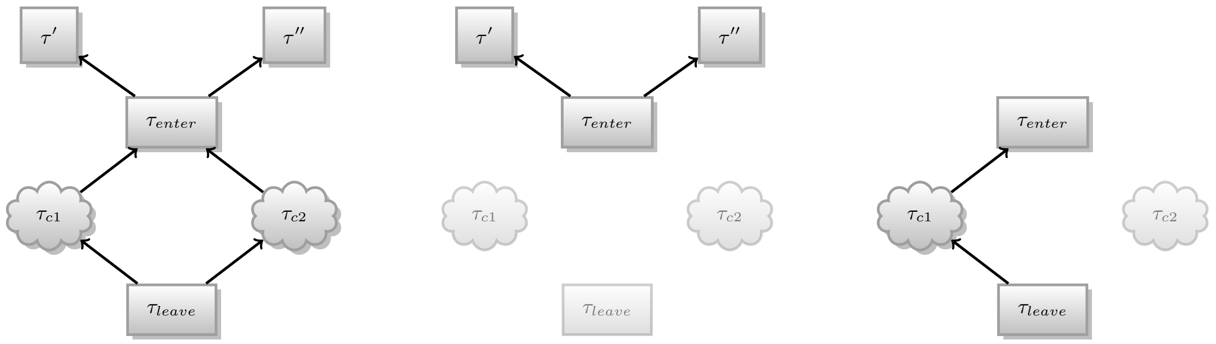


Figure 5.21 visualizes different stages during the execution of this generated code. We represent unscheduled tasks as slightly shaded. In Figure 5.21a we represented the state of step ② has been executed. All tasks have been created but not yet scheduled (indicated through the shaded representation). After the execution of step ③ only the "top-level" tasks have been scheduled with the corresponding dependencies (shown in Figure 5.21b). The method execution will the wait in step ④ for the completion of the body task (note that waiting does not mean blocking, because of the work stealing algorithm). In the mean meantime tasks τ' and τ'' can be executed. As soon as those tasks complete the τ_{enter} can start to execute. As describe, it will first execute any merged operations (see ②a) and then determine which of the cases is selected. Once the case has been determined the start task(s) of this case will be scheduled. The last tasks of the case task graph will trigger the scheduling of the τ_{leave} task.

Figure 5.21: Selective Match Execution Example



(a) Task Creation after Step ② (b) Scheduled Tasks after Step ③ (c) Task Graph after Excuting the Match Enter Task

Table 5.3: Overview of Code Generation Optimizations

Name	Description
Sequentializing Single Task Graphs	Generate sequential code for methods which have a task graph of only one node.
Inlining Starter Task	Inline start task into method body code block .
Inlining Body Task	Inline body task into the method body code block.

Code Generation Optimizations

While the task builder tries to minimize the number of task, there are still a few optimizations that can be performed during code generation to further reduce the number of creates tasks. The following sections present several optimizations that can help to reduce the number of created tasks (cf, Table 5.3 for a summary overview). We discuss each of the optimizations separately to focus on the core concept of each optimization. We present all optimizations for the method call levels, but notice that all optimizations are also applicable for case optimizations of match blocks. To focus on the optimization technique and for brevity reasons we use the generic scheduling algorithm as base for our extensions when we present those optimizations.

Sequentializing Single Task Graphs If the task builder manages to reduces the task graph of a whole method body to a single task, then the code generation will inline this task. This results in the generation of a sequential method body equivalent to the sequential method body that would have been generated by the standard code generator.

Listing 5.10: Single Task Function Graph

method PlaidObject m(...) {
  $\boxed{\tau@[\overline{\delta}]\langle\omega\rangle}$
}

Listing 5.11: Single Task Function Code

method PlaidObject m(...) {
  $\overline{\delta}_\tau$
}

Inlining Body Task Because the method always has to wait for the body task we can inline this task into the method body and avoid the creation and synchronization overhead for this task. Figure 5.12 on the next page shows the code generation strategy which comprises of the following steps:

① **Extraction of variables.** No changes.

② **Task Creation.** We create all tasks except out body task.

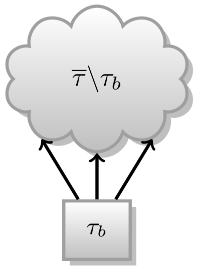
③ **Task scheduling.** No changes.

④ **Wait for dependencies.** Wait for all tasks the body task depended on to complete.

⑤ **Execute body task.** Execute the remaining operations of the body task and return the value of the last statement.

Listing 5.12: Inline Body Task Graph

method PlaidObject m(...) {



}

Listing 5.13: Inline Body Task Code

```
public PlaidObject m(...) {
    // create variables
  ① PlaidObject[] _ = new PlaidObject[$|\{VarDecl(x) \in \{\overline{\delta} : \overline{\tau@[\overline{\delta}]\langle\omega\rangle}\}\}|$];

    // create task objects
  ② $\forall \tau_i \in \{\overline{\tau} \backslash$ BODY_TASK$(\overline{\tau})\}$ : Task $T_{\tau_i}$ = new Task(|DEPS$(\tau_i)$|) {
        public void run() {
            IS_CASE_TASK$(\tau_i)$ $\implies$ if ( CASE_MATCH_COND$(\tau_i)$ ) { $\overline{\delta}_{\tau_i}$ }
            $\neg$IS_CASE_TASK$(\tau_i)$ $\implies$ $\overline{\delta}_{\tau_i}$
            $\forall \tau' \in$ RDEPS$(\tau_i)$ : if ( $T_{\tau'}$! = BODY_TASK$(\overline{\tau})$ &&
                                   $T_{\tau'}.decDepCount() == 0$ ) {
                schedule($T_{\tau'}$);
            }
        }
    };

    // compute dependencies and schedule tasks
  ③ $\forall \tau_i \in$ START_TASKS$(\overline{\tau})$ : schedule($T_{\tau_i}$);

    // wait for dependencies of the body task to finish
  ④ $\forall \tau_i \in$ DEPS(BODY_TASKS$(\tau)$) : $T_{\tau_i}$.wait();

  ⑤ return $\overline{\delta}_{\text{BODY\_TASKS}(\tau)}$;

}
```

Inlining Single Starter Task If a tasks graph has only one starter task we can inline this task similar to the inlining of the body task.

① **Extraction of variables.** No changes.

② **Execute start task code**. Execute the operations associated with the start task directly in method body.

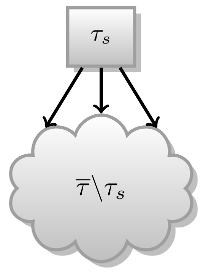
③ **Task Creation.** We create all tasks except the start task.

④ **Task scheduling.** Schedule all start tasks which depends on the original start task.

⑤ **Wait for body task.** No changes.

Listing 5.14: Inlining Start Task Graph

```
method PlaidObject m(...) {
```



```
}
```

Listing 5.15: Inlinig Start Task Code

```
public PlaidObject m(...) {
    // create variables
  ① PlaidObject[] __ = new PlaidObject[$\left|\{VarDecl(x) \in \{\overline{\delta} : \overline{\tau @ [\overline{\delta}] \langle \omega \rangle}\}\}\right|$];

    // execute start task code
  ② $\overline{\delta}_{\mathsf{START\_TASK}(\overline{\tau})}$

    // create task objects
  ③ $\forall \tau_i \in \{\overline{\tau} \backslash \mathsf{START\_TASK}(\overline{\tau})\}$ : Task $T_{\tau_i}$ = new Task($|\mathsf{DEPS}(\tau_i)|$) {
        public void compute() {
            $\mathsf{IS\_CASE\_TASK}(\tau_i) \implies$ if ( $\mathsf{CASE\_MATCH\_COND}(\tau_i)$ ) { $\overline{\delta}_{\tau_i}$ }
            $\neg\mathsf{IS\_CASE\_TASK}(\tau_i) \implies \overline{\delta}_{\tau_i}$
            $\forall \tau' \in \mathsf{RDEPS}(\tau_i)$ : if ( $T_{\tau'}.decDepCount() == 0$) { schedule($T_{\tau'}$); }
        }
    };

    // compute dependencies and schedule tasks
  ④ $\forall \tau_i \in \mathsf{RDEPS}(\mathsf{START\_TASKS}(\overline{\tau}))$ : schedule($T_{\tau_i}$);

    // wait for dependencies of the body task to finish
  ⑤ return $T_{\mathsf{BODY\_TASK}(\overline{\tau})}$.wait();
}
```

Dynamic Load Balancing

Despite the implemented optimizations our system can produce significant more tasks than we have parallel execution units. To eliminate the high costs of task creation and scheduling we implemented dynamic load balancing approach. Listing 5.16 on the following page shows the our current dynamic load balancing approach. Every method which supports parallel execution, first performs a check whether we have enough parallelism (i.e., enough generated tasks to utilize the available computation units) or not by calling the `PARALLELIZE` method. If this method returns `false` it means that we have enough work and should not generate new work. In this case we simply execute the sequential method body instructions. If the return value is `true` we need to generate more parallel work and we execute the parallel method body implementation as described earlier.

Listing 5.16: Dynamic Load Balancing

```
public PlaidObject m(PlaidObject pthis, ...) {
    if ( PARALLELIZE() == false ) {
      ... // sequential code
    } else {
      ... // parallel code
    }
}
```

An important observation is that when we execute the sequential code branch the sequentiality is only enforced for the current method. If the sequential code call a function which contains potential parallel executions this function will do the same check to determine if it should parallelize the code or not. This is an important feature of the system as it allows us to recover from heavily imbalanced code paths. The drawback of this approach is that we have to check for parallelization on every method which has potential parallelism.

Atomic Block Implementation

Our implementation allows seamlessly mixing of code with and without data groups. If we use code without data groups we are talking about plain *shared* permissions and atomic blocks without any datagroup parameters. In this datagroup-less mode we implicitly pass a share datagroup permission to an anonymous global datagroup into every method. Figure 5.17 on the next page shows that we simply translate an atomic block into the an `enterAtomic` and `leaveAtomic` method call on the corresponding datagroup. Once we entered an global atomic block we decided for simplicity reasons to sequentialize the execution of its body. This means that when we call a method from inside a global atomic block this methods needs to execute sequentially even if it could execute in parallel. There are two approaches to achieve this behavior. The first option is to have a dynamic check at runtime to force sequential execution. The second option is to have two versions of every method. One version which is called by default and another version which can only be called from inside an atomic block directly or transitively (cf. AtomJava [53]). We decide to go for the dynamic approach because it can be easily merged with the dynamic load balancing and avoids code explosion. Figure 5.18 on the facing page shows the implementation of the global atomic block sequentiallizing check.

In the case that we have actual data groups we translate an atomic block the same way with the exception of replacing the `GLOBAL_DATAGROUP` with the corresponding data groups specified by the user. Note that we do not have to sequentialize the execute of methods called from inside a non-global atomic block as we have explicit specified datagroup permissions which automatically enforce sequentialization where necessary.

Listing 5.17: Atomic Block Translation

```
atomic {                      GLOBAL_DATAGROUP.enterAtomic();
  ...           ⟹              ...
}                             GLOBAL_DATAGROUP.leaveAtomic();
```

Listing 5.18: Global Atomic Test

```
public PlaidObject m(PlaidObject pthis, ...) {
    if ( GLOBAL_DATAGROUP.inAtomic() ) {
        ... // sequential code
    } else {
        ... // parallel code
    }
}
```

5.5 ÆMINIUM Runtime

To execute the task we adapted the Fork-Join framework (FJ, [61]). The Fork-Join framework is thread pool utilizing a work-stealing approach (cf. [45]). The work-stealing approach utilizes thread-local work queues. This means if a thread creates additional tasks those tasks get added its local work queue instead of adding them to a global work queue. This approach avoid that the global work queue becomes a single point of congestion. If a thread runs out of he will try to steal tasks for the other threads in the thread pool. In other words, the work stealing algorithm uses a distributed work queue to avoid congestion. We changed the implementation slightly by adding a callback to FJ threads. As soon as a thread runs out of work it notifies the dynamic load balancer so that at least one thread is waiting for work. The dynamic load balancer uses this information to generate additional work as soon as possible.

Additionally to the modified FJ runtime we added two new classes. The first class is the `Task` class (see Listing 5.19) which extends FJ `RecursiveAction` and is used as base class for the task we create. The class consists of a constructor which takes the dependency count of this task as parameter. It furthermore has

Listing 5.19: ÆMINIUM Runtime Task

```
public abstract class Task extends RecursiveAction {
        public Task(int depsCount) { ... }
        final public int decDepCount() { ... }
}
```

Listing 5.20: ÆMINIUM Runtime Datagroup

```
public final class Datagroup implements PlaidObject {
        public final void enterAtomic() {
        public final void leaveAtomic()
}
```

a function `decDepCount` to decrement the internal dependency counter by one. This function returns the new dependency count.

The second class introduced is the `Datagroup` class (see Listing 5.20). Note that the this class implements the `PlaidObject` interface. This allows us to store datagroup objects directly as fields without having to go through the indirection via a `PlaidJavaObject`. The datagroup class features two simple functions. The `enterAtomic` method to indicate that we entered an atomic block for the specific data group and the `leaveAtomic` method to indicate that the atomic block has finished. In the current implementation the datagroup class represent a simple wrapper for a re-entrant lock which gets locked upon entering an atomic block and unlock when we leave it. Other implementation options such as Software Transactional Memory (STM) systems are applicable to our system as we. Based on our previous experience with STMs (cf. [19]) we decided to use a lock based approach for its simplicity and smaller performance overhead.

Evaluation

In this chapter we present the evaluation of the ÆMINIUM implementation. The evaluation consists of several case studies evaluating various aspects of the system. We demonstrate that ÆMINIUM is capable of improving the performance of an application, compared to its serial version, and is practical enough to be useful in several different scenarios. We conclude the chapter by elaborating on our experience with developing the case studies and on potential improvements to ÆMINIUM.

6.1 Methodology

Two of our hypothesis were that we can achieve performance improvements through ÆMINIUM and that ÆMINIUM is practical (cf. Chapter 1.3). We evaluate both by conducting several case studies. For each case study we evaluate how much performance benefit can be achieved compared to its sequential counter part (using the standard *speedup* metric). Practicality is slightly less straightforward than the performance evaluation. This is mainly due the fact that there is no exact definition what practicality means and how to measure it. Systems that are very practical in one situation (e.g., like SQL for processing uniform data in tables) can be completely impractical (e.g., not efficient or cumbersome) in other situations (e.g., using SQL for implementing a web server). We evaluate ÆMINIUM like similar systems have been evaluated. We choose to use case studies of common domains or problems which are known to benefit from parallelization, to show that ÆMINIUM is powerful enough to encode them. A common use case are applications with high latency input/output operations. A common solution is the use of concurrency to overlap communication and computation. To evaluate ÆMINIUM's capabilities to handle high latency input/output use cases we developed a web server application (cf. Section 6.3). Many algorithms rely on basic data structures such as lists, dictionaries, sets, etc. In a concurrent setting the scalability of those basic data structures is of paramount importance to the overall scalability of the algorithm. To evaluate how well ÆMINIUM handles common data structures we developed a case study implementing a dictionary (i.e., a hashmap, cf Section 6.4). A common use case for concurrency is the parallelization of highly computation-intensive applications to shorten the execution time (e.g., weather forecast). To demonstrate ÆMINIUM's suitability for this domain we developed a case study to numerically compute the integral of a function for a given interval (cf. Section 6.5 and 6.6).

The second part of this chapter evaluates how much overhead ÆMINIUM imposes on the programmer when writing code. We measure how many lines of source code (SLOC, measured with `wc`) we had to modify by: annotating types (i.e., adding permission information to types), how often we had to add additional group parameters to method calls, and how many ÆMINIUM specific operations we used (e.g., `atomic` blocks). We decided to use this metric as this is the most commonly used approach to quantify annotation overhead of comparable systems. Using the 'default' approach when measuring annotation overhead makes our evaluation comparable to others. We report and compare the numbers for the fully annotated program (i.e., every possible annotation is written out by the programmer) and a version which leverages Plaid's default permission system. Plaid's default permission system allows the programmer to omit permission annotations and let the compiler use a default permission for those objects. An example of this is *Plaid*'s `Integer` state. In Plaid `Integer` objects, like *Java* integer objects, are always `immutable` as numbers cannot be changed. This means that the programmer can write `Integer` and the compiler will automatically expand it to `immutable Integer`.

All of our benchmark results (with exception of the webserver benchmark) have been measured on an eight-core machine (Intel Xeon CPU X5460@3.16GHz) with $32GB$ of main memory. The eight-core machine runs under Fedora Linux 7 and all the benchmarks were run under the Java HotSpot 64-Bit Server VM (build 20.4-b02). For the we webserver we uses quad-core machine (Intel i7@2.0GHz) with $8GB$ of main memory. The quad-core machine runs OSX Mountain Lion using the Java HotSpot 64-Bit Server VM (build 20.14-b01-477). Unless otherwise stated we use the default settings of the installed Java Virtual Machine. All benchmarks ran at least 20 times each and we report the average runtimes along with the standard deviation, showing low variance in those runs. All measured values can be found in Appendix B.

6.2 Plaid's Baseline Performance Evaluation

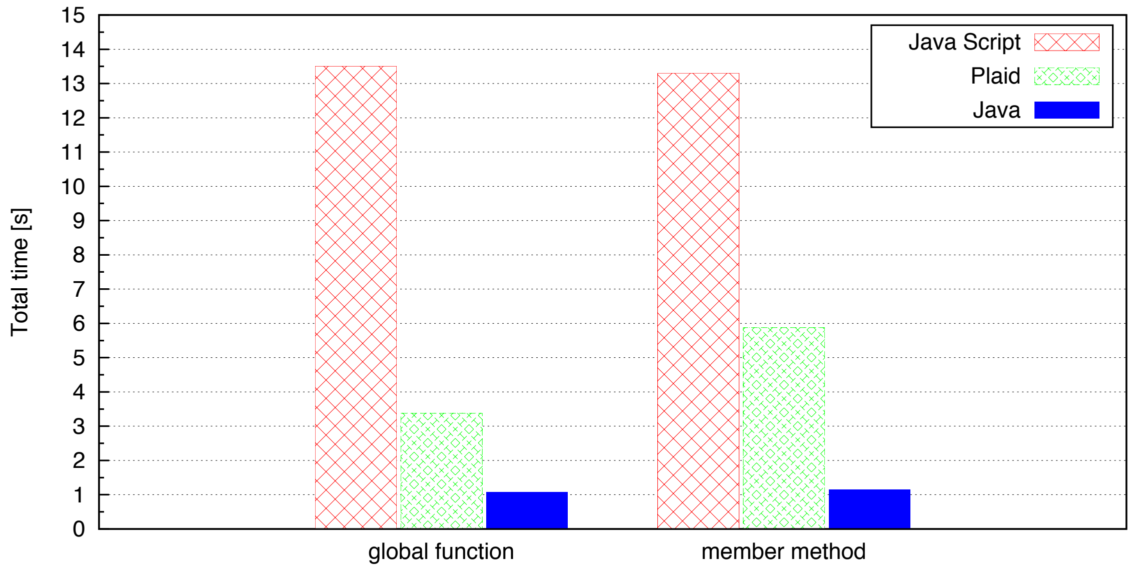
To get a better understanding of the performance numbers, this sections evaluates the baseline performance of Plaid. In particular we compare the performance of Plaid against *Java* (as a statically typed comparison) and *JavaScript* (for a dynamically typed comparison). For this evaluation we defined a simple fork/join

Figure 6.1: Baseline Benchmark Method : `forkJoin`

```
**Input**: $level$ dependency graph
**begin**
    $isZero \leftarrow level == 1$ ;
    **if** $isZero == false$ **then**
        $nextLevel \leftarrow level - 1$ ;
        forkJoin($nextLevel$) ;
        forkJoin($nextLevel$) ;
    **end**
**end**
```

based benchmark to evaluate the overall performance overhead for method calls, control flow and basic arithmetic operations. The benchmark emulates a divide-and-conquer algorithm. The pseudo code for benchmark is shown in Figure 6.1 on the preceding page (cf. Appendix C.1 for the source code of the actual implementations). When invoked the method first checks if the passed-in parameter is zero. In the case the value is zero, the method simply returns. In the case the parameter was not zero, the method decrements the parameter value and invokes itself recursively twice with the decremented parameter value. We implemented this benchmark in all three programming languages (Plaid, Java and JavaScript) in two different variants. The first variant uses global methods (i.e., methods not bound to an specific object instance) and member methods (i.e., bound to a specific object instance) to evaluate if there is a performance penalty for using member methods with dynamic dispatch.

Figure 6.2: Base Line Performance Evaluation. Comparing performance of Java, JavaScript and Plaid



	global function			member methods		
	JavaScript	Plaid	Java	JavaScript	Plaid	Java
Average Runtime	13.5s	3.4s	1.1s	13.5	5.9s	1.2s
Standard Deviation	0.16s	0.04s	0.02s	0.23s	0.06s	0.02

We run all benchmarks on our eight-core machine (cf. Section 6.1) repeating each benchmark 20 times. The average runtimes and standard deviation of those runs are shown in Figure 6.2. We choose a reasonably high start value to produce enough work for the *Java* implementation. The results of our evaluation are shown in Figure 6.2. There are several observations worth pointing out. The first one being that regardless of the programming language, global method calls are faster than ordinary member method calls. While in *Java* and *Java Script* those differences are marginal, the performance improvements for *Plaid* are much more noticeable. Comparing the $5.9s$ for the *Plaid* member method implementation to the $3.4s$ of *Plaid*'s global methods results in almost 74% faster method calls for global methods. This big performance difference is caused by the way *Plaid*'s dynamic dispatch is implemented. *Plaid*'s dynamic dispatch uses the same principle as in *Thorn* [25], Jython [6] or JRuby [8]. The idea is to generate a dispatch interface for every method which is implemented by all classes which have a matching method signature. Because *Plaid* allows dynamic composition of states at runtime the system will generate appropriate classes through dynamic code generation with proxy functions to forward all method calls to the actually generated methods. This leads to one level of indirection where first the proxy method is called and then calls the compiler generated method. This is obviously a slower approach than calling a global method which does not require dynamic dispatch and results into a direct static method call to the compiler generated code.

The other important observation regarding the overall performance of *Plaid*. *Plaid* performs in this benchmark generally better than *JavaScript* but worse than *Java*. *JavaScript* takes about $13.4s$ to complete the benchmark while *Java* only need about $1.1s$. This means that *JavaScript* is about 12 times slower than *Java* in this benchmark. *Plaid* needs about $5.9s$ in the case of member methods and about $3.4s$ in the global method case to complete the benchmark. This means that using dynamic dispatch results in a slowdown of 5.3 times compared to *Java* and using global methods in a slow down of 3.1 times compared to *Java*. Having a slightly worse performance than plain *Java* makes sense as the *Plaid* compiler translates *Plaid* code into *Java* code (cf. Section 5.3) and *Plaid* has additionally to support dynamic *Plaid*. Note that the numbers are of fully typed programs. Using the dynamic subset of *Plaid* might result in worse performance because some optimizations require type information.

6.3 Case Study: Webserver

This case study analyses a webserver application for serving static webpages. The goals of this case study are to see how well ÆMINIUM helps to parallelize code and how well it works to improve performance by overlapping computation and communication. The full source code of the *Plaid* implementation is listed in Section C.2.

6.3.1 Code Generation Analysis

This section analyses the generated code by the *Plaid* compiler. The *Plaid* webserver implementation consists of four files. Two simple wrapper states for *Java*'s socket classes, a main method to start the webserver and the actual webserver state. This section solely focuses on the webserver state as the remaining code mainly consists of wrapper methods for *Java* classes.

Table 6.1: `WebServer` **Method Overview**

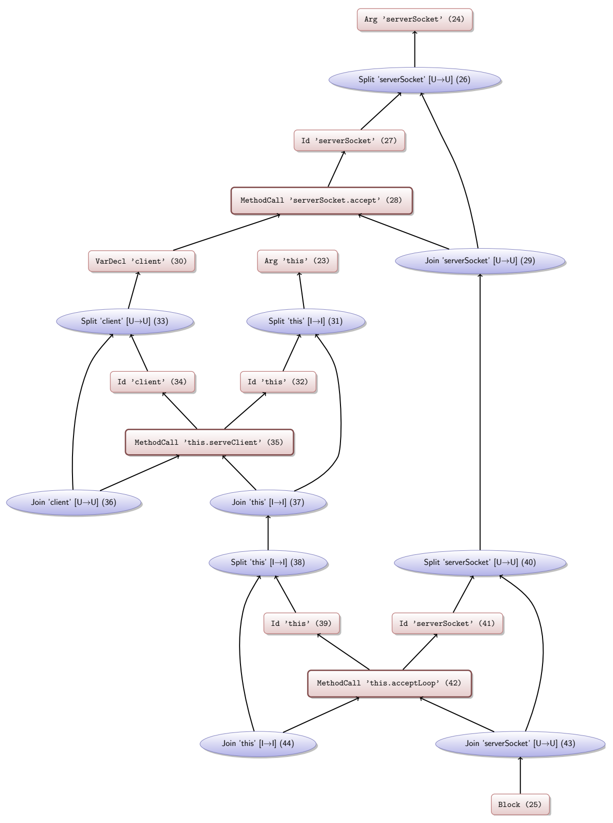
Name	Annotations	#Tasks	#Inlined	Description
`getRoot`	`@sequential`	1	1	Locates root directory in file system.
`run`	●	1	1	Starts the accept loop.
`acceptLoop`	●	3	2	Accepts new connections in an infinite loop.
`fileExists`	`@sequential`	1	1	Checks if requested file exists and is accessible.
`serveClient`	●	1	1	Processes accepted connection.
`transferData`	`@sequential`	1	1	Transfers binary data from file to socket.
`transferFile`	●	1	1	Helper function to transfer requested file.
`transferHeader`	●	1	1	Generates HTML response header.

Listing 6.1: `acceptLoop` **code**

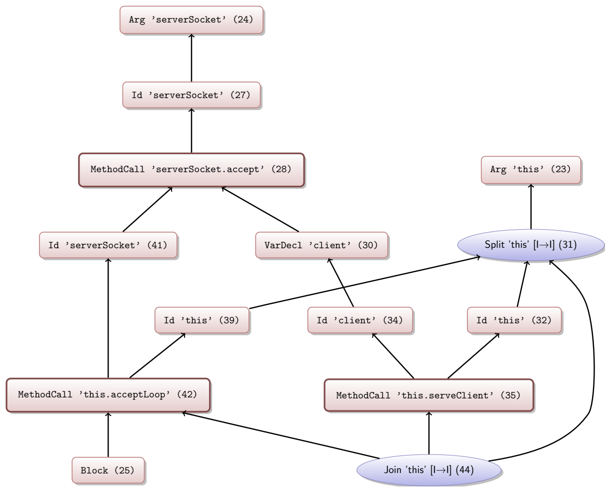
```
state ServerSocket {

    method unique Socket accept() [ unique ServerSocket this ] { . . . }

    . . .
}

state WebServer {

    method void acceptLoop(unique ServerSocket serverSocket) [immutable WebServer this] {
        val unique Socket client = serverSocket.accept();
        this.serveClient(client);
        this.acceptLoop(serverSocket);
    }

    method void serveClient(unique Socket client) [immutable WebServer this] { . . . }

    . . .
}
```

Table 6.1 shows all the methods defined in the `WebServer` state along with their annotations, the number of computed tasks by the Plaid compiler, how many tasks have been inlined (cf. Section 5.4.4 all implemented task inlining optimizations), and a short description of the methods. All access to the file system has been extracted into three methods annotated with `@sequential` (as they need to use *Java* code to access the file system). For all but one method the *Plaid* compiler reduced the method body to one task. As described in Section 5.4.4, one task method graphs do not encode any parallelism and therefore result in sequential code for this method. On the flip side this means that only the `acceptLoop` method was considered by the compiler to contain (enough) parallelism.

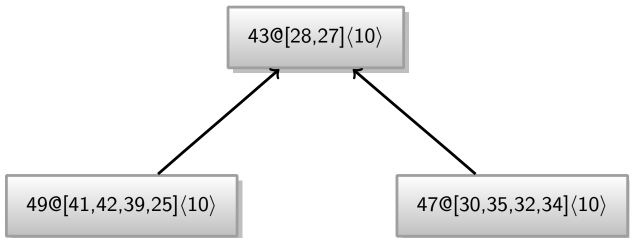
Figure 6.3: Compiler generated dependency and task graphs of the `acceptLoop` method.



(a) Original Dependency Graph



(b) Æminiumified Dependency Graph



(c) Task Graph

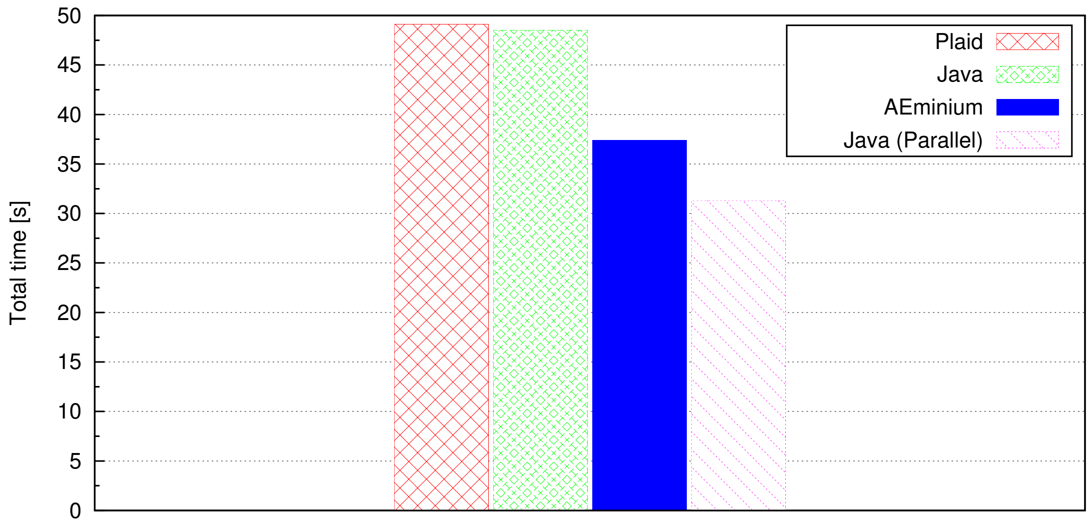
Listing 6.1 on the previous page shows the source code of the `acceptLoop` method along with the method signature of all called methods. As shown the `acceptLoop` method first calls the `accept` method on the passed in `ServerSocket`. The `accept` method waits until there is a new client connection available and returns the corresponding `Socket` object. After a new connection has been established the method calls the `serveClient` method to process the client request. Then the `acceptLoop` method call itself recursively to accept the next client.

The dependency graph of the `acceptLoop` method is shown in Figure 6.3a on the facing page. Due to the eager type checker implementation (cf. Chapter 5) all three statements are dependent on each other because they require an `immutable` permission to the receiver. As described, the Æminiumifier pass will infer potential parallelism and transforms the graph into a parallel version. The Æminiumified dependency graph of the `acceptLoop` method is shown in Figure 6.3b on the preceding page. The recursive call to `acceptLoop` depends now on the `accept` call of the `ServerSocket` instead of the `serveClient` method call. The `serveClient` method call still depends on the `accept` method call because it needs to wait for the client socket to be returned. The resulting task graph is shown in Figure 6.3c on the facing page. With task @43 representing the `socket` method call, task @49 representing the recursive `acceptLoop` method call and task @47 encapsulates the `serveClient` method call. This allows the processing of the client connection in parallel with the recursive method call, which in turns represents the opportunity to accept and process the next client. This allows the ÆMINIUM compiled version to handle multiple connections at a time.

6.3.2 Performance Evaluation

We evaluated the performance of our webserver by comparing the sequential *Plaid* version against the ÆMINIUM parallelized version of the webserver. We ran each benchmark 20 times. We show the avergage runtimes and standard deviation in Figure 6.4 on the next page. As a reference, we additionally compare the performance to sequential and parallel *Java* versions. To evaluate the performance we installed each version on our quad-core machine (cf. Section 6.1) hosting the *Python* 2.7 documentation [7]. We then mirrored this documentation with *Parallel URL Fetcher* ([10], puf) to our local machine. Puf is a multi-threaded download utility which tries to fetch its data with up to 20 connections. We ensured that all accessed files are cached in the memory buffers to limit our measurements to network input/output latency and avoid additional hard drive input/output latency. We repeated each measurement three times in a round robin fashion to avoid temporary network congestion that impacts one specific version. Figure 6.4 on the following page shows the average performance values measured. The *Plaid* version of the webserver is the slowest ($49.1s$) followed by the sequential *Java* version ($48.5s$). This makes sense as *Plaid* is generally slower than *Java*. The performance difference is not as big as shown in Section 6.2 because this application consists of a computational and communication component. The communication component is the same for both cases, therefore the slower performance of *Plaid* does not affect the overall performance that much. The ÆMINIUM compiled version of the webserver is the second fastest ($37.4s$) version. It is approximating 31% faster than its sequentially compiled counterpart. The reason for this is that the webserver in the ÆMINIUM compiled version is able to handle multiple requests in parallel. This allows the overlapping of communication and computation and results in a higher throughput. The manually parallelized *Java* version delivered the best performance (31.2). The performance difference between the parallelized *Java* and the ÆMINIUM version is bigger

Figure 6.4: WebServer Performance Graph



	Plaid	Java (sequential)	ÆMINIUM	Java (parallel)
Average Runtime	49.1s	48.5s	37.4s	31.3s
Standard Deviation	2.1s	3.5s	1.92	0.8s

compared to their sequential counter parts. This effect is caused by the parallel execution and the overlap of communication and computation which hides the communication costs to some degree. Because the communication effect is reduced the computation part gains relatively more weight. Section 6.2 established that *Plaid* is several times slower that *Java* code. Therefore the ÆMINIUM version is slower than parallel *Java* version.

6.3.3 Annotation Overhead

Table 6.2 on the next page shows the annotation overhead measured for the *Plaid* webserver applications as described in Section 6.1. We denote fully annotated version as '*' and the version that uses *Plaid*'s build-in default permission with '†'. The numbers show that, on average, in the fully annotated programs every 5^{th} line of code needs to be annotated. All those changes are solely caused by permission annotations to types.

Table 6.2: Annotation overhead in WebServer application

Filename	**SLOC**	**Modified SLOC**	**Type Annotations**	**Group Arguments**	**ÆMINIUM Constructs**
main.plaid*	6	1 (16.6%)	1	0	0
ServerSocket.plaid*	16	5 (31.5%)	6	0	0
Socket.plaid*	34	9 (26.5%)	11	0	0
WebServer.plaid*	171	32 (18.7%)	41	0	0
Σ*	**227**	**47 (20.7%)**	**59**	**0**	**0**
main.plaid†	6	0 (0%)	0	0	0
ServerSocket.plaid†	16	0 (0%)	0	0	0
Socket.plaid†	34	0 (0%)	0	0	0
WebServer.plaid†	171	0 (0%)	0	0	0
Σ†	**227**	**0 (0%)**	**0**	**0**	**0**

Leveraging *Plaid*'s default permission mechanism allows us to essentially write the whole webserver program without any additional permission annotations.

6.4 Case Study: Dictionary

The *Problem Based Benchmark Suite* (`http://www.cs.cmu.edu/~pbbs/`) defines several problems representing real-world tasks. We choose to developed a dictionary benchmark to evaluate the effectiveness of data groups. Our implementation is based on a hash map using separate chaining to handle collisions. We developed two versions: a *global* version which uses plain *shared* permissions for its internal data structures and a *fine* version in which every bucket has its own data group. The *global* version which uses plain *shared* permissions for which the mutual exclusion in the current implementation defaults to global locking strategy. The *fine* version uses shared permissions associated with data groups, enabling a more fine-grained locking approach, and yielding in a better scalability compared the *global* version.

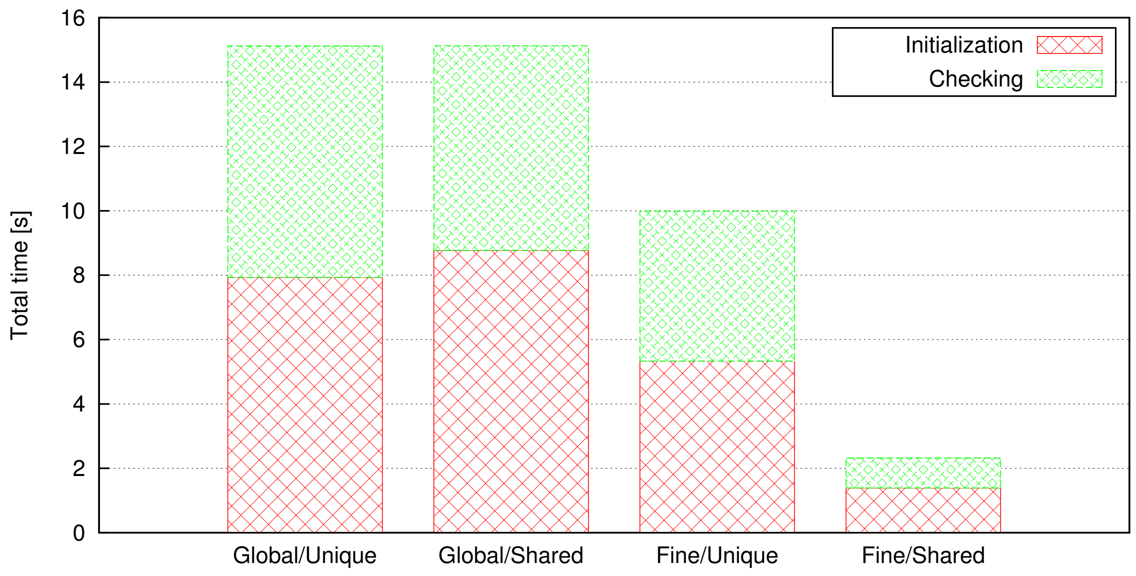
6.4.1 Code Generation Analysis

The dictionary by itself does not generate any parallelism by itself as all used operations (i.e., `add` and `contains`) are sequential by nature. The implementation allows parallel access if the dictionary is shared amongst different entities.

6.4.2 Performance Evaluation

Figure 6.5 on the facing page shows the average runtimes (based on 50 iterations of each benchmark) and standard deviation of the dictionary benchmarks. The benchmark first inserts the identity mapping for the numbers 2^0 to 2^{16} into the dictionary (initialization). Then we lookup every mapping to check for correctness (checking). We used a dictionary with 64 hash buckets. To avoid artificial patterns we randomized the sequence in which the numbers are inserted/checked with a constant seed to guarantee reproducibility. The first bar 'global/unique' ($15.1s$) represents the results of the *global* dictionary implementation with a *unique* permission to the dictionary. The linearity of the *unique* permission sequentiallizes all insert/check operations. In the second bar 'global/shared' ($15.1s$) we have a *shared* permission to the dictionary, which allows us to perform our operations in parallel. This case performs no better because each parallel operation must immediately synchronize on the entire *shared* dictionary structure, thus sequentializing all the accesses. The third bar 'fine/unique' ($10.0s$) uses the implementation which utilizes data groups for its internal representation. This scenario is faster than any of the cases using the global implementation, because of the use of fine-grained data groups, one for each bucket. The *unique* receiver permission allows us to get *exclusive* group permissions to the inner groups of the dictionary. This means we do not require protection to access data within those data groups and therefore we avoid unnecessary synchronization operations. The last case 'fine/shared' ($2.3s$) also allows the parallel execution of our operations. Because the implementation associates each bucket with its own data group, we achieve a very fine-grained protection mechanism which

Figure 6.5: Dictionary Performance Graph



	global/unique		global/shared		fine/unique		fine/shared	
	init	check	init	check	init	check	init	check
Average Runtime	7.9s	7.2s	8.8s	6.4s	5.3	4.6s	1.4s	0.9s
Standard Deviation	0.12s	0.10s	0.11s	1.9s	0.12	0.12	0.01s	0.01s

allows the parallel modification of disjoint parts of the dictionary. This results in a speedup of $6.5X$ compared to the 'global/shared' version.

6.4.3 Annotation Overhead

Tables 6.3 on the next page and 6.4 on the following page shows the annotation overhead measured for our dictionary implementations (global and fine) as described in Section 6.1. We denote fully annotated source code versions with '*' and the versions that use *Plaid*'s build-in default permissions with '†'. The numbers for the global implementation show that, on average, in the fully annotated case, every 4^{th} line needs to be changed. Using *Plaid*'s built-in default permissions significantly reduces the annotation overhead down to 3%. The numbers for the fine implementation are approximately the same for the fully annotated case, which requires every 4^{th} line to be changed. When employing *Plaid*'s default permission mechanism we still have to modify every 5^{th} line of code. While achieving some reduction the overall overhead is not as low as in the other cases. This overhead could not be reduced by *Plaid*'s default permission as the fine implementation associates its internally shared data with data groups. To inform the compiler that those shared data belong to a specific data group, we need to add additional information.

Table 6.3: Annotation overhead of *Global Dictionary* implementation.

Filename	**SLOC**	**Modified SLOC**	**Type Annotations**	**Group Arguments**	**ÆMINIUM Constructs**
Bucket.plaid*	42	9 (21.4%)	13	0	2
BucketList.plaid*	38	8 (21.1%)	9	0	1
GlobalHashmap.plaid*	38	16 (42.1%)	24	0	0
GlobalHashmapAddOperations.plaid*	9	1 (11.1%)	5	0	0
GlobalHashmapContainsOperations.plaid*	9	1 (11.1%)	5	0	0
GlobalHashmapOperations.plaid*	9	1 (11.1%)	3	0	0
package.plaid*	24	5 (20.8%)	6	0	0
Σ*	**169**	**41 (24.2%)**	**65**	**0**	**3**
Bucket.plaid†	42	2 (4.7%)	0	0	2
BucketList.plaid†	38	1 (2.6%)	0	0	1
GlobalHashmap.plaid†	38	2 (5.2%)	2	0	0
GlobalHashmapAddOperations.plaid†	9	0 (0%)	0	0	0
GlobalHashmapContainsOperations.plaid†	9	0 (0%)	0	0	0
GlobalHashmapOperations.plaid†	9	0 (0%)	0	0	0
package.plaid†	24	0 (0%)	0	0	0
Σ†	**169**	**5 (3%)**	**2**	**0**	**3**

Table 6.4: Annotation overhead of *Fine Dictionary* implementation.

Filename	**SLOC**	**Modified SLOC**	**Type Annotations**	**Group Arguments**	**ÆMINIUM Constructs**
Bucket.plaid*	100	24 (24%)	32	4	2
BucketList.plaid*	52	16 (30.7%)	18	2	0
FineHashmap.plaid*	38	16 (42.1%)	24	0	0
FineHashmapAddOperations.plaid*	14	1 (7.1%)	12	2	0
FineHashmapContainsOperations.plaid*	14	1 (7.1%)	12	2	0
FineHashmapOperations.plaid*	9	1 (11.1%)	5	0	0
package.plaid*	24	5 (20.8%)	6	0	0
Σ*	**251**	**71 (28.3%)**	**109**	**10**	**2**
Bucket.plaid†	100	24 (4.7%)	20	4	2
BucketList.plaid†	52	10 (19.2%)	8	2	0
FineHashmap.plaid†	38	2 (5.2%)	2	0	0
FineHashmapAddOperations.plaid†	14	4 (28.6%)	4	2	0
FineHashmapContainsOperations.plaid†	14	4 (28.6%)	4	2	0
FineHashmapOperations.plaid†	9	2 (22.2%)	3	0	0
package.plaid†	24	0 (0%)	0	0	0
Σ†	**251**	**46 (18.3%)**	**41**	**10**	**2**

Listing 6.2: `Integral` **code**

```
state Integral {
    method immutable Float64 compute(immutable Float64 x1, immutable Float64 x2) [immutable Integral this] {
        val immutable Float64 delta = x2 − x1;

        val immutable Boolean divide = delta.nativeLessThan(0.00000001);
        match ( divide ) {
            case True {
                val immutable Float64 f1 = this.f(x1);
                val immutable Float64 f2 = this.f(x2);
                val immutable Float64 combinedf = f1 + f2;
                val immutable Float64 avgf = combinedf / 2.0;
                val immutable Float64 area = avgf ∗ delta;

                area
            }
            default {
                val immutable Float64 combinedx = x1 + x2;
                val immutable Float64 middle = combinedx / 2.0;
                val immutable Float64 area1 = this.compute(x1, middle);
                val immutable Float64 area2 = this.compute(middle, x2);

                area1 + area2
            }
        }
    }

    @cheap
    method immutable Float64 f(immutable Float64 x) [immutable Integral this];
}
```

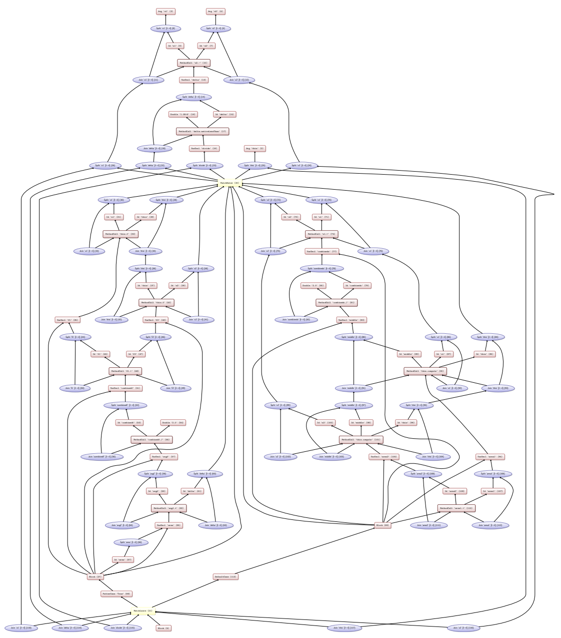
6.5 Case Study: Integral

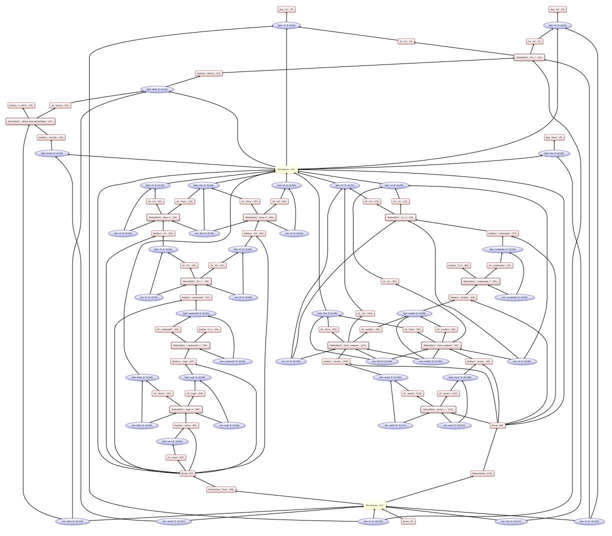
This case study evaluates ÆMINIUM capability to parallelize purely functional, highly computation intensive problems. We developed a small integral library which computes the integral of a user-defined function. The integral is computed by subdividing the overall interval into infinitesimal small intervals for which we calculate the approximate area, and then add up all fractions to compute the area of the whole integral.

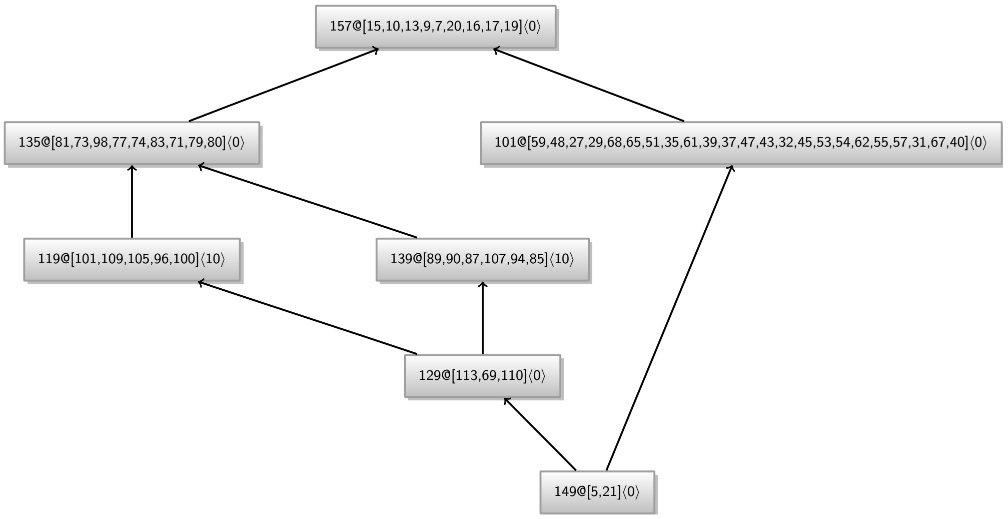
6.5.1 Code Generation Analysis

The main functionality of the integral case study is implemented in the `Integral.plaid` file shown in Figure 6.2. The `Integral` state has a `compute` method responsible for the computation of the integral and an abstract method `f` which users need to override in their implementation to implement the actual function. The `compute` method has two parameters defining the start and end of the interval. The method

Figure 6.6: Compiler generated dependency and task graphs of the `Integral.compute` method.


(a) Original Dependency Graph


(b) Æminiumified Dependency Graph



(c) Task Graph

first determines if the current interval is below a certain threshold or not. In the case that the interval is below the threshold the method approximates the area of the interval by computing its trapezoidal area using the user overridden method `f` and returns it. If the interval is above the threshold the method divides the interval and computes the integral of those subintervals recursively.

Table 6.5: `Integral` Method Overview

Name	Annotations	#Tasks	#Inlined	Description
`Integral.compute`	●	7	5	Sub-divides and computes integral recursivley.
`Integral.f`	`@cheap`	1	1	User defined function prototype.
`main`	●	1	1	Program entry point.
`Runner.stdout`	`@sequential`	1	1	Print messages to stdout.
`Runner.run`	●	1	1	Runs integral computation.
`SquareIntegral.f`	●	1	1	Implements square function (i.e., x^2).

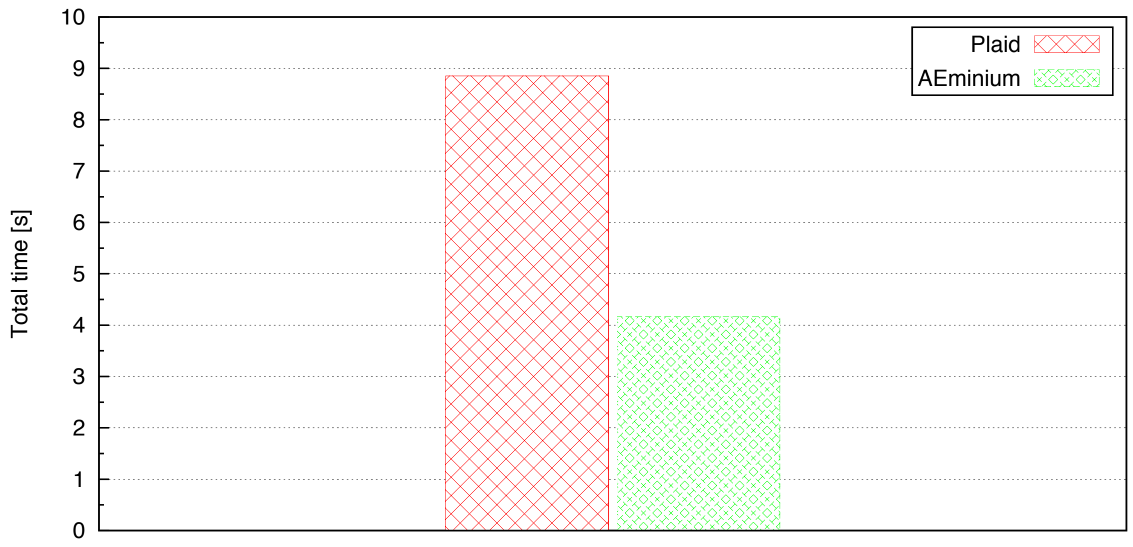
Figure 6.6a on the facing page shows original dependency graph of the `Integral.compute` method. The parallelized dependency graph, Figure 6.6b on the preceding page, allows the parallel execution of the recursive method calls and the method calls to the user-defined function `f`. We choose to annotate the user defined method with a `@cheap` annotation as we do not expect the computation of a function value to require significant computation effort. This allows the compiler to merge the two method calls to the user-defined function `f` into one task instead of generating parallel tasks for each of those method calls. The final task graph of the `Integral.compute` method is shown in Figure 6.6c on the facing page. The first task (157) represents the threshold computation and the begin of the match block. The long task (101) on the right hand side represents the `True` case which approximates the integral area for the given interval. Thanks to the cheap annotation on the user-defined function `f`, the task builder is able to reduce the whole case into a single task. The 'diamond' shaped task graph on the left represents the `False` case which recursively computes the interval. The top task (135) of the diamond represents the computation of the subintervals. The two edge tasks (119 and 139) represent the recursive method calls to compute the integral for the subintervals. The bottom task (129) of the diamond graph represents the accumulation of the sub interval area values.

Table 6.5 shows the methods in the integral case study along with the number of inferred and inlined tasks. The numbers show that all but the `Integral.compute` have been reduced into one task. The `Integral.compute` consists of seven tasks of which five are inlined by the code generator. The two task which have not been inlined consist of the two recursive method calls.

6.5.2 Performance Evaluation

We evaluated the performance by computing the integral of the square function (i.e., $f(x) = x^2$) for the interval $[0, 1]$. We run the sequential *Plaid* and parallel ÆMINIUM version on our eight-core machine each 20 times. The average runtime and standard deviation of both cases are shown in Figure 6.7 on the following page. The *Plaid* version requires $8.9s$ while the ÆMINIUM version needs only $4.2s$. This results in a speedup of 2.1 meaning that ÆMINIUM was able to parallelize the program and achieve some performance improvements. But it also means that the ÆMINIUM version was only twice as fast on an eight core machine, which would suggest a speedup closer to eight. Our investigation revealed that the main source for this poor

Figure 6.7: Integral Performance Graph



	Plaid	ÆMINIUM
Average Runtime	8.6s	4.2s
Standard Deviation	0.19s	0.15s

performance lays in the *Plaid*'s object system. As described previously, *Plaid* does not support primitive types which means that value in *Plaid* is an object. This means that in this computation-heavy application we have to create a new object for every floating point value we compute. Our investigation showed that the this particular benchmark allocates more than 1.8 billion (1.8×10^9) floating point objects. This means that overall performance of out benchmark is limited the throughput of the virtual machine memory system. This result does not invalidate the ÆMINIUM approach, because the problem is an intrinsic limitation of the Plaid programming language and not of ÆMINIUM.

6.5.3 Annotation Overhead

Table 6.6 on the next page shows the annotation overhead measured for the *Plaid* integral applications as described in Section 6.1. We denote the fully annotated version of '*' and the version that uses *Plaid*'s build-in default permission with '†'. The numbers show that in the fully annotated program every 3^{rd} line of code needs to be annotated. All those changes are solely caused by permission annotations to types. Leveraging *Plaid*'s default permission mechanism makes it possible to write the whole integral program without any additional permission annotations.

Table 6.6: Annotation overhead of *Integral* implementation.

Filename	SLOC	Modified SLOC	Type Annotations	Group Arguments	ÆMINIUM Constructs
Integral.plaid*	30	13 (43.3%)	18	0	0
main.plaid*	17	1 (5.8%)	1	0	0
Runner.plaid*	8	4 (50%)	4	0	0
SquareIntegral.plaid*	6	1 (16.6%)	3	0	0
Σ^*	**61**	**19 (31.2%)**	**26**	**0**	**0**
Integral.plaid*	30	0 (0%)	0	0	0
main.plaid*	17	0 (0%)	0	0	0
Runner.plaid*	8	0 (0%)	0	0	0
SquareIntegral.plaid*	6	0 (0%)	0	0	0
$\Sigma^\dagger$	**61**	**0 (0%)**	**0**	**0**	**0**

6.6 Case Study: ForkJoin

Our Integral case study in Section 6.5 showed that ÆMINIUM is capable to parallelize purely functional programs while achieving performance benefits. Because of the memory issue we were not able to evaluate how well our implementation actually scales. We therefore conducted another case study, specifically designed to emulated the core principle of the Integral case study but without the memory issues. We decided to reuse the fork join benchmark we used in Section 6.2 to evaluate our baseline performance. This benchmark uses the same divide-and-conquer approach as the Integral case study, but only uses small integer values. The usage of small integer values is vital as Plaid, like Java, uses a singleton pattern to cache small integer object between -127 and 128. This avoid the memory explosion in the Integer case.

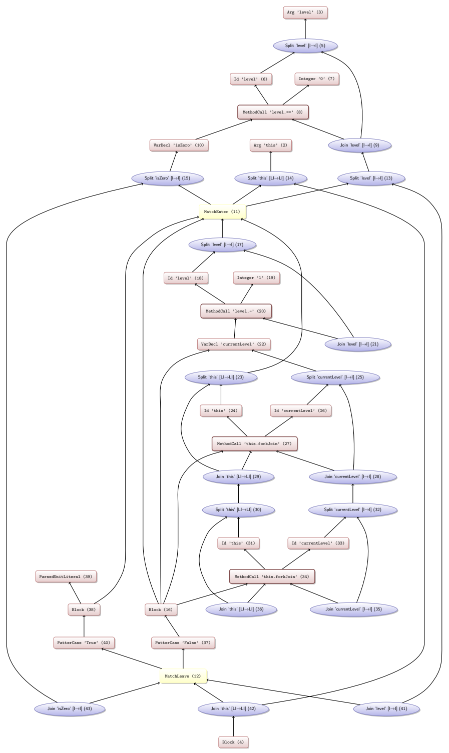
6.6.1 Code Generation Analysis

The generated code resembles closely the code generated for the Integral case study. This makes sense as the ForkJoin case study has been designed to emulate the Integral case study. Table 6.7 shows the method information for the ForkJoin case study. The main function `forkJoin`, like the `compute` function of the Integral, consists of seven tasks. In both cases five tasks can be inlined.

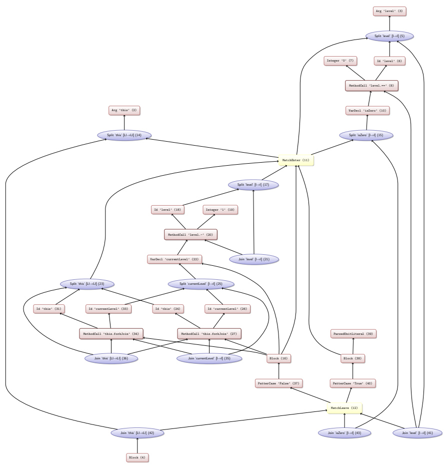
Table 6.7: `ForkJoin` Method Overview

Name	Annotations	#Tasks	#Inlined	Description
`FJ.compute`	●	7	5	Recursiveley calls itself until a certain depth is reached.
`main`	●	1	1	Program entry point.

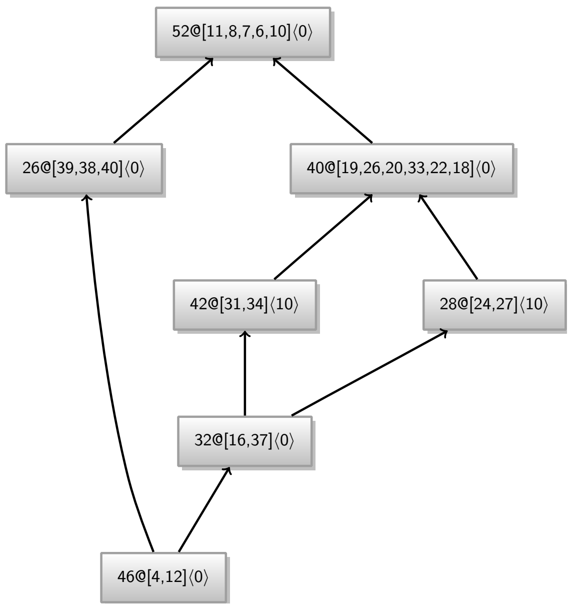
Figure 6.8: Compiler generated dependency and task graphs of the `FJ.forkJoin` **method.**



(a) Original Dependency Graph

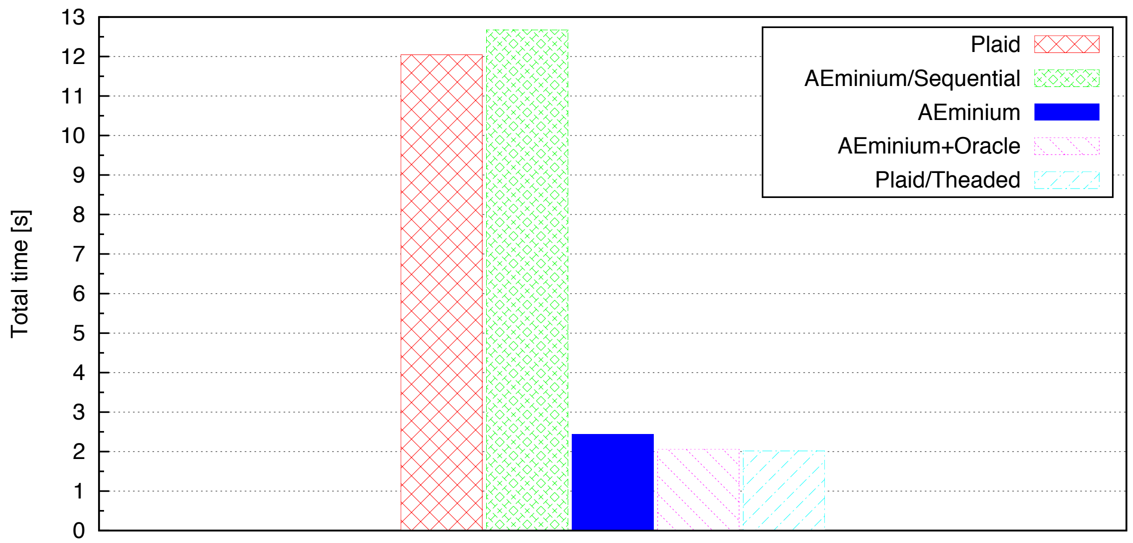


(b) Æminiumified Dependency Graph



(c) Task Graph

Figure 6.9: ForkJoin Performance Graph



	Plaid	ÆMINIUM (seq)	ÆMINIUM	ÆMINIUM (Oracle)	Plaid (threaded)
Average Runtime	12.1s	12.7s	2.4s	2.1s	2.0s
Standard Deviation	0.12s	0.41s	0.05s.	0.06s	0.05s

6.6.2 Performance Evaluation

We evaluated five different cases, (sequential) *Plaid*, (sequential) ÆMINIUM, (parallel) ÆMINIUM, (parallel) ÆMINIUM with oracle scheduling [11] and manually parallelized *Plaid* version using threads. We run every case 20 times on our eight-core machine and Figure 6.9 shows the average runtimes and standard deviations. We increased the input size to a reasonably high number to achieve a higher computation demand that can be parallelized. The sequential Plaid version took $12.0s$ to complete the benchmark. To measure the overhead introduced by our dynamic load balancing approach we used the ÆMINIUM parallelized version and manually modified the runtime system to let the `PARALLELIZE` method (cf. Section 5.4.4) always return false. This sequentializes the whole benchmark execution and allows us to determine the overhead we introduced by our code transformations. The sequential ÆMINIUM execution took $12.7s$ to complete the benchmark. This means that our overhead compared to the sequential *Plaid* version is about 5.2%. Note that the dynamic scheduling check is only inserted into methods which exhibit parallelism and not into methods which execute sequentially. The ÆMINIUM parallelized version with the dynamic load-balancing properly working needed on average $2.4s$. Comparing the sequential Plaid version with the ÆMINIUM version, this results in a speedup of 4.9 times. This means that our implementation provides much more speedup than the speedup of 2 measured for the Integral case. Despite the fact that the speedup significantly improved, our improved value is not as close to the perfect speedup of 8 as one would expect. Our investigation revealed that the main cause of this performance loss is due the creation of of tasks which represent too small computation, because as soon as an internal thread runs out of work it triggers the runtime to create more work. The runtime does not know the exact runtime costs of specific tasks as those might depend on the current input. Therefore the runtime might create too small tasks which cause too much overhead compared to the actual runtime of the task.

As mentioned in Section 5.4.3, one proposed solution to this problem is oracle scheduling [11]. The idea behind oracle scheduling is that the user supplies an additional function which is used by the runtime system to estimate the how many operations the original function will run for the given input size. For evaluation purposes we manually implemented the oracle scheduling in the compiler generated code. The ÆMINIUM version with the manual oracle scheduling took $2.1s$ to complete the benchmark. Compared to the sequential Plaid version this results in a speedup of 5.9. While yielding in higher performance we still do not achieve the optimal performance.

To gain a better idea about the baseline performance of *Plaid* we implemented a manually parallelized version of the program using threads. This version creates one thread per CPU and uses the sequential *Plaid* version to compute the appropriate sub-problem. Because the fork join benchmark represents perfectly balanced binary call tree, each thread has the same amount of computation. This version needed $2.0s$ to complete the benchmark resulting in a speedup of 6. This version represents the best possible case. There is exactly one thread per CPU and all threads have to perform the same amount of work (which eliminates potential load imbalances). This means that the oracle enabled version achieves almost the same performance as the optimal case and is only 2.4% slower compared to the optimal case. This also means that the ÆMINIUM version without the oracle scheduling is only 21.4% slower than the optimal case. The optimal version also shows that we are still losing scalability somewhere. The manually threaded version does not use our runtime and uses only *Java* threads and executes sequential Plaid code. Given that each thread executes purely functional code there is no reasons for any kind of inter-thread dependency. Our initial investigation revealed that this slow down is associated with the *Plaid* runtime system. To further investigate we manually removed step-by-step all Plaid specific features from the generated code until we had a plain Java version of the benchmark. This investigation isolated the performance issue to *Plaid*'s dynamic dispatch as one of the main sources for this scalability issue. *Plaid*'s object system is beyond the scope of this thesis and orthogonal to our problem as the ÆMINIUM approach is language agnostic and does not depend on *Plaid*'s object system. A future version of the *Plaid* runtime may address these issues.

6.6.3 Annotation Overhead

Table 6.8 on the facing page shows the annotation overhead measured for the *Plaid* fork join benchmark as described in Section 6.1. We denote the fully annotated version as '*' and the version that uses *Plaid*'s

Table 6.8: Annotation overhead of *ForkJoin* implementation.

Filename	SLOC	Modified SLOC	Type Annotations	Group Arguments	ÆMINIUM Constructs
package.plaid*	21	4 (19.1%)	5	0	0
Σ*	**21**	**4 (19.1%)**	**5**	**0**	**0**
package.plaid*	21	1 (4.7%)	1	0	0
Σ†	**21**	**1 (4.7%)**	**1**	**0**	**0**

Figure 6.10: Annotation Overhead over Java

Program	Total SLOC	Annotaded SLOC	Type Annotations	Group Arguments	ÆMINIUM Constructs
webserver*	227	47 (20.7%)	59	0	0
dictionary/global*	169	41 (24.2%)	65	0	3
dictionay/fine*	251	71 (28.3%)	109	10	2
integral*	61	19 (31.2%)	26	0	0
forkJoin*	21	4 (19.1%)	5	0	0
Total*	**729**	**182 (25.0%)**	**262**	**10**	**5**
webserver†	227	0 (0.0%)	0	0	0
dictionary/global†	169	5 (3.0%)	2	0	3
dictionary/fine†	251	41 (18.3%)	41	10	2
integral†	61	0 (0.0%)	0	0	0
forkJoin†	21	1 (4.7%)	1	0	0
Total†	**729**	**52 (7.1%)**	**44**	**10**	**5**

build-in default permission with '†'. The numbers show that in the fully annotated program every 5^{th} line of code needs to be annotated. All those changes are solely caused by permission annotations to types. Leveraging *Plaid*'s default permission mechanism allows us to reduce this to every 20^{th} line.

6.7 Reflection on Case Studies

Figure 6.10 shows the summary for all our case studies. The values marked with '$*$' are versions fully annotated and values marked with '†' are programs which use Plaid's default permission mechanism which allows omitting the annotations by specifying a default permission. The numbers show that type annotation are the most common source of overhead and that Plaid's default permissions help to reduce it substantially. The second important observation is that the more developers specify, the more performance the compiler can achieve. Figure 6.11 on the following page compares the percentage of SLOC that has to be changed (with the numbers for *Plaid*'s default permissions) against how much speedup has been achieved. As the

Figure 6.11: Comparison of Speedup against percentage of changed SLOCs.

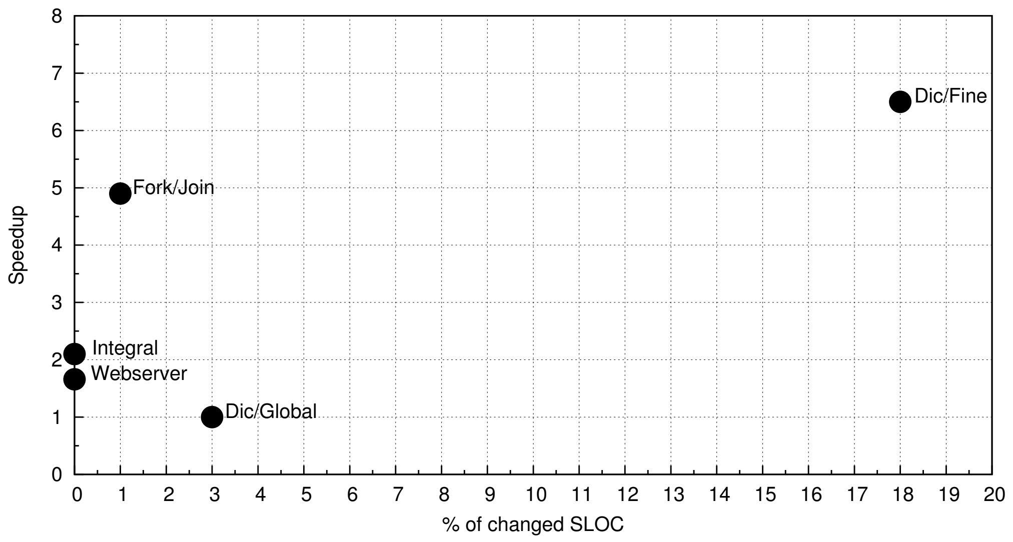


diagram shows for programs which only use unique and immutable permissions we can achieve speedup with little to no annotation overhead. When it comes to using shared data the amount of effort put into annotating the source code reflects how good the speedup is. Using only the bare minimum of annotation (i.e., Dic/Global) we do not achieve any performance improvements at all. When using more annotations (i.e., Dic/Fine) the overall speedup goes up. This means users can start with a simple version of a program and then incrementally add more annotations to increase the performance. It is worth pointing out that using Plaid's default permission approach we are able to extract concurrency in the webserver and integral example without the need for *any* additional annotations. Overall we achieve a reasonable 7.1% annotation overhead which is comparable to the 10.7% reported by DPJ [27]. Further improvements to our system (e.g., type inference) should allows us to further mitigate the programmers' burden. The reader should also take into account that access permission information in Plaid serves additional purposes (e.g., checking typestate). The following sections elaborate on specific observations we experienced during the evaluation of our implementation.

6.7.1 Readonly group permission

When we used data groups and group permissions in our case studies we encountered certain scenarios (e.g., in the dictionary case study to avoid synchronization the case of immutable permissions to the dictionary) in which it would have been useful to have a `readonly` group permission. The `readonly` group permission would allow concurrent read access to the objects of a data group (by guaranteeing there is no write present, similar to `immutable` permissions for Objects). Adding `readonly` group permissions to the language does not seem to impose a major change to the system and should merely be a matter of working out the appropriate language abstraction to split an `exclusive` permission to a `readonly` permission along with minor modifications to type checker to correctly check and compute dependencies.

6.7.2 Inference

Plaid's default permission support is a useful feature to reduce the amount of permission information that the programmer has to write. Adding automatic inference represents another opportunity to further reduce the annotation overhead. Inference seems particularly useful for local variable declarations and group parameters of method calls. In the case of variable declarations the programmer would just declare a new local variable using the `var` or `val` keyword and the variable name. The resulting type would be determined by the type of the initialization expression. The group parameters for methods calls could easily be omitted by the programmer and inferred from the type of the passed in parameters. If the automatic inference is not possible (e.g., the programmer passes in a unique reference which need to be split to the concrete shared permission) the programmer can always fall back to the manual annotation.

6.7.3 Polymorphism

One shortcoming we noticed during the implementation of the case studies was the lack of parametric polymorphism (i.e., generics) in the *Plaid* language. Prior to *Java 5* programmers had to use `Object` as the combined type for their data structures if they wanted them to be applicable to any type, and later use a dynamic state test to determine the actual type of the object. This approach works only partly in *Plaid* because every type in *Plaid* also contains a permission for which we do not have a generic abstraction. We therefore need to have a special version for each kind of permission, which results in code duplication. Some preliminary work on polymorphic permissions has been conducted by Nels Beckman [17].

6.7.4 Oracle Scheduling

We already discussed the usage of oracle scheduling to improve the overall parallelization strategy. In Section 6.6 we shows by manually implementing oracle scheduling in the compiler generated code that we can achieve close to optimal performance. We did not add oracle scheduling support to the source level at this point. The original proposal for oracle scheduleing, used user-defined methods to perform the runtime cost estimation. While this works well in a purely function setting (where oracle scheduling has been implemented and evaluated) it has some shortcomings in an object-oriented setting with dynamic dispatch. Also, using a more high-level declarative approach for specifying the runtime heuristic would be beneficial, both for programmer to specify and for the compiler to verify them.

6.7.5 Java Interoperability and Standard Library

Plaid targets the *Java Virtual Machine (JVM)* and was designed with *Java* interoperability in mind. While it is possible to create and use *Java* objects in the dynamic subset of *Plaid*, the current implementation of the type checker does not support checking of *Java* object creation and method calls. Therefore we have to wrap every *Java* object into a *Plaid* wrapper state and use the `@sequential` annotation to prevent type checking. This causes additional work for the programmer. An automatic approach to support *Java* code inside *Plaid* has been started but requires more effort before it can be completed.

6.7.6 Tooling Support

We did not only developed our case studies in *Plaid* but also wrote the whole *Plaid* compiler (with exception of the parser) in *Plaid* itself (cf. Section 5.3). The *Plaid* compiler uses our first prototype implementation to compile and execute. Out first prototype implementation does not support type checking. This means that users can use permission and type information in their code but the compiler will not check or enforce them. Despite the fact that those type annotations are not necessary in the code, most parts of the *Plaid* compiler are fully annotated. Having those annotations in the code, despite the fact that they are not checked or enforced, helped the readability and maintenance of the code. This becomes particularly obvious when realizing that the *Plaid* compiler comprises of almost 29 KLOC. The type annotations serves as crucial part of the documentation to quickly determine for instance what kind of object to pass into certain functions.

All our case studies use the new *Plaid* compiler which supports type checking. In our experience this was a big improvement in productivity. Some of the most common mistakes we made using the dynamic *Plaid* prototype compiler were the misspelling of variables and method names or passing the wrong type or number of arguments to functions. All of those issues are caught by the new compiler at compile time instead of run time. While the current *Plaid* compiler reflects a significant leap forward it is by no means free of flaws. The current compiler uses our initial prototype implementation which is several order of magnitude worse in every almost aspect: execution performance, memory footprint and generated code size. Currently some effort is made to make the new compiler self hosting, which eliminates many of those issues. Another criticism of the current compiler is the way errors are reported. *Plaid* supports a powerful type system which allows relatively detailed knowledge of what problems occur. Unfortunately the compiler does a rather poor job communicating this knowledge to the user, and sometimes provides rather cryptic error messages which only make sense when knowing the internal details of the compiler.

Aside from the compiler issues, the lack of any additional development tools provides an additional barrier. We developed several syntax highlighting plugins for various editors (e.g., *Emacs*, *TextWrangler*, etc). While simple syntax highlighting provided additional help it cannot replace the a fully featured development system. The development tool which we missed most was a source level debugger. The only way to debug *Plaid* programs at this point is to debug the generated *Java* code. This requires in depth knowledge of the compiler internals. Aside from that, another important *Integrated Development Environments (IDE)* feature we missed was a fully featured editor for the source code. By fully featured editor we mean an editor which not only supports syntax highlighting, but also code completion, background compilation and automatic indention. Other useful features which would improve productivity are refactoring, an incremental build system, a profiler, etc.

6.7.7 Programming and Parallelizing with Permissions

One of the main differences when writing programs in *Plaid* compared with *Java* is the use of the permissions. As described in Section 6.7.3, permissions add another dimension of flexibility. Instead of just thinking which objects are used the programmer also needs to think about which permissions are required. During the development of our case studies we had to refine our interfaces as the original anticipated permissions were not strong enough or were too strong for what we actually needed. While those changes seem annoying at first they actually reveal the power of the system. Because the compiler complains, for instance about not having strong enough permission to perform a specific operation, we can catch inconsistencies early on. Having to change the permissions on the interface of a state can lead to ripple effect because dependent code might not type check any longer because of the changes we made. Adapting this code to the changes might break other code and so on. In most cases those changes are solely changes to the required permissions and would not have been necessary in *Java* which has no permission information at all. While this implies that annotations can require more effort to accommodate changes it also shows a strength of the system, namely that it can check that the permissions invariants hold for the whole program.

When it comes to parallelizing with permissions we already established that it is technically feasible to use permission to parallelize code. A question remaining is how easy are permissions to use to parallelize code. In our experience the main issue we encountered is lack of feedback caused by the lack of available tooling (cf. Section 6.7.6). At this point the compiler reads in a *Plaid* file and generates *Java* files as output or lists all errors encountered. This means that the programmer does not know if and which functions actually have been parallelized. It has been our objective to automate as much as possible in ÆMINIUM and require as little as possible for programmers to understand what is happening internally. But sometimes having additional knowledge about what the compiler actually might be useful, in particular if the generated code does not behave as expected (e.g., the code does not get parallelized). As happened to us in a few instances, sometimes the compiler did not parallelize enough (e.g., we overlooked certain dependencies) or parallelized too much (e.g., a very simple operation for which we had forgotten to add the `cheap` annotation). In those situations we used a special debug flag to make the compiler generate dependency and task graphs and then manually mapped the source code onto the nodes. Having this information allowed us to understand quickly what the compiler did and why it did what it did. The mapping of source code to task and the graphical representation can be done automatically and should help to provide for a smoother programming experience. In particular having such a feature built into an IDE would have helped with most questions we encountered during our evaluation. Similarly useful would have been a profiler to identify hotspots.

6.8 General Limitations

By using automatic parallelization we abstract away any low-level parallel programming constructs from the developer. In some case this approach might be limiting and reduce the expressiveness of our system. For instance it is not possible to express wait-free algorithms with ÆMINIUM, because of there is not support for low-level operations such as compare-and-swap.

Another limitation of programs that can be expressed in ÆMINIUM comes from access permissions itself. While access permissions are a powerful abstraction, there are programs which are hard to express with access permissions. For instance, if access permissions cannot be joined back in the same scope in which the split happened it is very hard for a static type system to merge them back at a later point. An example of such a situation is when a programmer splits of a permission and stores the permission in a field in the heap.

The current system allows shared permissions only to be associated with a single data group. While have not encountered any concrete issues with this approach in general there are situations in which multiple data groups per shared permission could make sense. An example for such a multi data group scenario could be a matrix implementation in which we either want to group rows or columns together depending on the use case.

While the current system demonstrates the general feasibility of the approach, there are still a few usability issues that need to resolved before it can be used for every day developing (e.g., further simplifications to the language, proper tooling support such as debuggers and profilers, etc). Nonetheless the current work represents a good first step towards practicability.

7 Conclusion

We conclude this dissertation by revisiting our original claims and how we validated them. We outline some future work for further improving ÆMINIUM. We conclude this chapter by summarizing the whole thesis.

7.1 Thesis Statement and Hypothesis, Revisited

Our thesis statement claims:

> *"The flow of access- and group-permissions provides a powerful abstraction to capture common programming idioms while simultaneous enabling the safe extraction of efficient concurrency."*

We systematically evaluated this claim by breaking it into small hypothesis which we individually validated (cf. Chapter 6).

Hypothesis: Safety

To ensure safety we can develop and formalize an analysis that uses the access permission and data group permission flow throughout a program to compute data dependencies, which allow the concurrent execution of the program while guaranteeing the absence of race conditions.

Validation

We validated this hypothesis through the development of the core calculus called μÆMINIUM (cf. Chapter 4). μÆMINIUM consists of static type checking rules and concurrent-by-default small-step evaluation semantics and consists of data groups and permissions. We proofed the type soundness of the system stating that every correctly typed program is free of data races (cf. Appendix A). μÆMINIUM also uses the permission flow to compute the correct dependencies and to model a concurrent-by-default evaluation.

Hypothesis: Efficiency

Programs written in our concurrent-by-default approach can be parallelized automatically, achieving a better performance than their sequential counterparts (provided sufficient availability of parallelism in the application itself).

Validation

To validate this hypothesis we conducted several case studies and showed that ÆMINIUM is capable to extracting parallelism and achieves performance improvements. Our case studies also show that our generated load balancing code introduces a modest overhead of merely 5.2% compared to its sequential *Plaid* counter part. We also showed that ÆMINIUM parallelized code about 21.4% slower than an equivalent program that has been manually been parallelized.

Hypothesis: Practicality

We claim that access permissions and data group permissions provide a powerful abstraction for concurrent-by-default applications, which allows common program idioms to be captured.

Validation

We validated this hypothesis by conducting several case studies in different domains, ranging from I/O heavy operations, to essential data structures to computational heavy programs. We further reported on our experience developing those case studies using permissions. We found that using permission requires some additional overhead either because of additional code that needs to be written or mentally about thinking about the usage pattern of objects. Additionally we found that access permission are a powerful abstraction but can have some limitations in expressiveness. But because permissions are checked by the compiler, its is rather easy to find the violation as the compiler points them out. We demonstrated in Section 6.7 that Plaid's built-in default permission mechanism significantly reduces the require annotation overhead. Another advantage is the assurance that when the compiler accepts the program that every object is only accessed according to its available permission. That being said the system is still not ready for every day development. Further improvements, such as automatic permission inference and better tooling support, building upon our initial work will help to further improve every day usability.

7.2 Future Work

In our opinion additional effort to push ÆMINIUM to the next level should focus on two areas: improvements to the language and development of tooling. Besides implementing the remaining features of *Plaid* (e.g., lambda abstractions in the type system) we discussed the potential addition of `readonly` group permissions. Further improvements that would benefit the overall programming experience are polymorphism and type inference.

We also believe that developing decent tool support is of paramount importance. The *Plaid*/ÆMINIUM compiler contains a wealth of useful information that can be leveraged to reduce the programmers burden and increase his productivity. Standard IDE features ranging from better error messages to a good source editor to a debugger are essential in today's development process. Additional tools building on top of our unique features (e.g., type state, parallelization, etc) should be developed to provide the programmers with all the available information. Tools like sophisticated visualizations seem to be a good first target. When thinking about ÆMINIUM it would be particularly useful to have an easy way to visualize the generated task graph and have a visual mapping between the source code and the graphs.

More investigation and integration of ÆMINIUM with other language features needs to be done. In our proposed approach we did not address the synchronization on our programs, but we believe that the approach of Damiani et al. [41] provides an elegant solution to combine Plaid's typestate feature with ÆMINIUM's automatic parallelization approach. Damiani proposes to the state of an object, more precisely the changes of states, to implicitly synchronize programs. For example, if a certain piece of code want to remove an element from a list which is in the 'EMPTY' state, then the code will wait until the list changes its state to a state which indicates that it has elements which can be removed. Another language feature which we did not discuss in details is error handling mainly because Plaid itself does not have a clearly defined error model. Future work should investigate how to support errors in the ÆMINIUM approach. This investigation should first evaluate existing error approaches (such as exceptions, error code, etc) regarding their suitability for the concurrent-by-default programming style and might continue on in investigating new error models as necessary.

Access permissions still have some limitations in which programs can be expressed. Some of the problems with permissions are related to collections, polymorphism and recovering access permission stored in the heap. Future work needs to address those issue to further increase the applicability of access permissions.

7.3 Reflection: Concurrency-By-Default

We presented ÆMINIUM as a concrete example of a concurrent-by-default programming approach based on permissions. We demonstrated that ÆMINIUM work very well in a purely functional settings (as advocated by the functional community) and provides reasonable support for imperative programming with shared data. Only time will tell whether ÆMINIUM will be adopted by other people in future, but addressing its existing shortcomings (cf. Section 7.2 and Section 6.7) will play a vital role.

The core idea proposed by this thesis was a new programming paradigm called concurrent-by-default. While we proposed ÆMINIUM as as an concrete example, potential other approach exists and could proof more suitable than ÆMINIUM. Nonetheless all those approaches will share the overall idea to move away from sequential thinking when writing programs into concurrent-by-default setting. One of our goals was to develop as system that works for parallelism like a garbage collectors work for memory management. Our experience so far shows quite some similarities. On many cases the system just works as expected, but there occasional situation were the system does not behave as expected (e.g., does not parallelize code or the performance we get does not meet the performance we expected). In those situation a more thorough analysis is required. Having better tooling support and a more internalized shift towards a concurrent-by-default paradigm might help to easy this burden. Similar to garbage collectors, our approach is generally applicable but is not suitable in special purpose situations. Examples of such situations are realtime systems in which a specific timing needs to be guaranteed or low-level systems in which certain code needs to be executed on specific hardware resources.

We are convinced that a concurrent-by-default is a viable programming model for the future. While there are still cases in which a low-level programming model is ore desirable (e.g., for wait-free algorithms) we believe a concurrent-by-default approach to be more suitable for mainstream programming. Whether the permission based approach of ÆMINIUM will be the adopted solution or another approach following the concurrent-by-default paradigm does not matter as it is the core principle which matters.

7.4 Summary

In this thesis we proposed a new programming paradigm called: *concurrency-by-default.* In this new paradigm all parts of a program, to the extent of not violating dependencies, can be executed concurrently by default. Therefore the programmer no longer needs to reason about complicated and error prone ordering constraints. The programmer simply reasons about dependencies and leaves the execution and scheduling to the runtime system.

We presented ÆMINIUM, a new programming language, designed after the concurrency-by-default paradigm. ÆMINIUM uses access permissions and data groups to specify and verify dependencies. So far we successfully developed a core calculus for a representative subset of the ÆMINIUM language. After having evaluated a proof-of-concept prototype implementation in a master thesis [70] we implemented the full ÆMINIUM system inside the Plaid programming language (cf. Chapter 5). We evaluated our implementation through several case studies showing that ÆMINIUM is capable of extracting parallelism based on the permission flow while achieving performance numbers close to manually parallelized code. As shown in Chapter 6 our case studies demonstrated that the ÆMINIUM approach is applicable to the chosen examples and allows us to gain performance improvements. There are still several issues that need to be solved before a concurrent-by-default programming model can become mainstream. We consider ÆMINIUM a first step towards this goal, demonstrating that such a system is doable.

REFERENCES

[1] GNU Compiler Framework. http://gcc.gnu.org/.

[2] Intel Compiler. http://software.intel.com/en-us/intel-compilers/.

[3] ISO/IEC 9075 (ISO SQL Standard). http://www.iso.org/iso/catalogue_detail.htm?csnumber=34132.

[4] Microsoft Compilers. http://msdn.microsoft.com/en-us/visualc/default.aspx.

[5] Portland Group Compiler. http://www.pgroup.com/.

[6] Python implementation for the jvm. http://www.jython.org/.

[7] The python programming language. http://www.python.org/.

[8] Ruby implementation for the jvm. http://jruby.org/.

[9] T-Systems Cell Compiler. http://www.t-platforms.ru/en/tcell/cellcompiler.html.

[10] The Parallel URL Fetcher. http://puf.sourceforge.net/.

[11] ACAR, U. A., CHARGUÉRAUD, A., AND RAINEY, M. Oracle scheduling: controlling granularity in implicitly parallel languages. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011), OOPSLA '11, ACM, pp. 499–518.

[12] ALDRICH, J., BECKMAN, N. E., BOCCHINO, R., NADEN, K., SAINI, D., STORK, S., AND SUNSHINE, J. The Plaid Language: Typed Core Specification. Tech. Rep. CMU-ISR-12-103, Carnegie Mellon University, 2012.

[13] ALDRICH, J., SUNSHINE, J., SAINI, D., AND SPARKS, Z. Typestate-oriented programming. In *OOPSLA* (2009).

[14] ALLEN, E., CHASE, D., HALLETT, J., LUCHANGCO, V., MAESSEN, J., RYU, S., STEELE JR, G., AND TOBIN-HOCHSTADT, S. The Fortress language specification version 1.0. Tech. rep., Technical report, Sun Microsystems, Inc, 2008.

[15] ANDERSON, Z., GAY, D., ENNALS, R., AND BREWER, E. Sharc: checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 149–158.

[16] ARMSTRONG, J. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, July 2007.

[17] BECKMAN, N. *Types for Correct Concurrent API Usage*. PhD thesis, Carnegie Mellon University, 2010.

[18] BECKMAN, N. E., BIERHOFF, K., AND ALDRICH, J. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not. 43*, 10 (2008), 227–244.

[19] BECKMAN, N. E., KIM, Y. P., STORK, S., AND ALDRICH, J. Reducing stm overhead with access permissions. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming* (New York, NY, USA, 2009), IWACO '09, ACM, pp. 2:1–2:10.

[20] BEST, M. J., MOTTISHAW, S., MUSTARD, C., ROTH, M., FEDOROVA, A., AND BROWNSWORD, A. Synchronization via scheduling: techniques for efficiently managing shared state. *SIGPLAN Not. 47*, 6 (June 2011), 640–652.

[21] BIERHOFF, K., AND ALDRICH, J. Modular typestate checking of aliased objects. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications* (Montreal, Quebec, Canada, 2007), ACM, pp. 301–320.

[22] BLELLOCH, G. E. NESL: A Nested Data-Parallel Language (3.1). Tech. Rep. CMU-CS-95-170, Carnegie Mellon University, September 1995.

[23] BLELLOCH, G. E., CHATTERJEE, S., HARDWICK, J. C., SIPELSTEIN, J., AND ZAGHA, M. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, May 1993), pp. 102–111.

[24] BLELLOCH, G. E., AND GREINER, J. A provable time and space efficient implementation of nesl. In *ACM SIGPLAN International Conference on Functional Programming* (May 1996), pp. 213–225.

[25] BLOOM, B., FIELD, J., NYSTROM, N., ÖSTLUND, J., RICHARDS, G., STRNIŠA, R., VITEK, J., AND WRIGSTAD, T. Thorn: robust, concurrent, extensible scripting on the jvm. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2009), OOPSLA '09, ACM, pp. 117–136.

[26] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not. 30*, 8 (1995), 207–216.

[27] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel java. In *OOPLSA* (2009).

[28] BOCCHINO JR, R., HEUMANN, S., HONARMAND, N., ADVE, S., ADVE, V., WELC, A., AND SHPEISMAN, T. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL* (2011).

[29] BOEHM, H. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (Chicago, IL, USA, 2005), ACM, pp. 261–268.

[30] BOEHM, H.-J. Transactional Memory Should Be an Implementation Technique, Not a Programming Interface. Tech. Rep. HPL-2009-45, HP Laboratories, 2009.

[31] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: preventing data races and deadlocks. *OOPSLA 37* (November 2002), 211–230.

[32] BOYLAND, J. Checking interference with fractional permissions. In *SAS* (2003), Springer, pp. 55–72.

[33] BROOKES, S. D., HOARE, C. A. R., AND ROSCOE, A. W. A Theory of Communicating Sequential Processes. *J. ACM 31*, 3 (1984), 560–599.

[34] CHAFI, H., DEVITO, Z., MOORS, A., ROMPF, T., SUJEETH, A., HANRAHAN, P., ODERSKY, M., AND OLUKOTUN, K. Language Virtualization for Heterogeneous Parallel Computing. Tech. rep., 2010.

[35] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[36] CHAPMAN, B., JOST, G., AND VAN DER PAS, R. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Oct. 2007.

[37] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not. 40*, 10 (2005), 519–538.

[38] CHENG, G.-I., FENG, M., LEISERSON, C. E., RANDALL, K. H., AND STARK, A. F. Detecting data races in cilk programs that use locks. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 1998), ACM, pp. 298–309.

[39] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership types for flexible alias protection. In *OOPSLA* (1998).

[40] CRAIK, A., AND KELLY, W. Using Ownership to Reason about Inherent Parallelism in Object-Oriented Programs. In *Compiler Construction*, R. Gupta, Ed., vol. 6011 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 145–164.

[41] DAMIANI, F., GIACHINO, E., GIANNINI, P., AND DROSSOPOULOU, S. A type safe state abstraction for coordination in java-like languages. *Acta Informatica 45* (2008), 479–536.

[42] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[43] DEITZ, S. J. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, University of Washington, Feb. 2005.

[44] DUFFY, J., AND ESSEY, E. Running Queries On Multi-Core Processors. online, October 2007. `http://msdn.microsoft.com/en-us/magazine/cc163329.aspx`.

[45] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not. 33*, 5 (1998), 212–223.

[46] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 ed. Addison-Wesley Professional, Nov. 1994.

[47] GIRARD, J.-Y. Linear logic. *Theor. Comput. Sci. 50*, 1 (1987), 1–102.

[48] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[49] GREENHOUSE, A. *A Programmer-Oriented Approach to Safe Concurrency*. PhD thesis, Carnegie Mellon University, 2003.

[50] GREENHOUSE, A., AND BOYLAND, J. An object-oriented effects system. In *ECOOP' 99 — Object-Oriented Programming*, R. Guerraoui, Ed., vol. 1628 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1999, pp. 668–668.

[51] HALL, M. W., ANDERSON, J. M., AMARASINGHE, S. P., MURPHY, B. R., LIAO, S.-W., BUGNION, E., AND LAM, M. S. Maximizing multiprocessor performance with the suif compiler. *Computer 29*, 12 (1996), 84–89.

[52] HEWITT, C., BISHOP, P., AND STEIGER, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence* (San Francisco, CA, USA, 1973), IJCAI'73, Morgan Kaufmann Publishers Inc., pp. 235–245.

[53] HINDMAN, B., AND GROSSMAN, D. Atomicity via source-to-source translation. In *Proceedings of the 2006 workshop on Memory system performance and correctness* (New York, NY, USA, 2006), MSPC '06, ACM, pp. 82–91.

[54] HOEFLINGER, J. P. *Extending OpenMP to Clusters*. Intel Corporation. `http://www.intel.com`.

[55] HUNT, G. C., AND LARUS, J. R. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 37–49.

[56] IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight Java: a minimal core calculus for Java and GJ. In *OOPSLA* (2001).

[57] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 59–72.

[58] JONES, S. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[59] KOBAYASHI, N. A new type system for deadlock-free processes. In *CONCUR 2006–Concurrency Theory*. Springer, 2006, pp. 233–247.

[60] LARUS, J., AND RAJWAR, R. *Transactional Memory*, 1 ed. Morgan & Claypool Publishers, 2007.

[61] LEA, D. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (New York, NY, USA, 2000), JAVA '00, ACM, pp. 36–43.

[62] LEINO, K. R. M. Data groups: specifying the modification of extended state. In *Proc. ACM SIGPLAN conference on OOPSLA* (New York, NY, USA, 1998), pp. 144–153.

[63] LOVEMAN, D. B. High performance fortran. *IEEE Parallel Distrib. Technol. 1*, 1 (1993), 25–42.

[64] MANSON, J., PUGH, W., AND ADVE, S. V. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2005), ACM, pp. 378–391.

[65] MCKEEMAN, W. M. Peephole optimization. *Commun. ACM 8* (July 1965), 443–444.

[66] MEIJER, E., BECKMAN, B., AND BIERMAN, G. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), ACM, pp. 706–706.

[67] MESSAGE PASSING INTERFACE FORUM. *MPI: A Message Passing Interface Standard*, Juni 1995. `http://www.mpi-forum.org`.

[68] MESSAGE PASSING INTERFACE FORUM. *MPI-2: Extensions to the Message-Passing Interface*, Juli 1997. `http://www.mpi-forum.org`.

[69] MICROSOFT CORPORATION. *Axum Programmer's Guide*, 2009. `http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx`.

[70] MOHR, M. Aeminium compilation theory in the context of the plaid language. Master's thesis, Karlsruhe Institute of Technology (KIT), Feb. 2011.

[71] MOORE, K. F., AND GROSSMAN, D. High-level small-step operational semantics for transactions. In *POPL* (2008).

[72] NADEN, K., BOCCHINO, R., ALDRICH, J., AND BIERHOFF, K. A type system for borrowing permissions. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2012), POPL '12, ACM, pp. 557–570.

[73] NYSTROM, N., CLARKSON, M., AND MYERS, A. Polyglot: An extensible compiler framework for java. In *Compiler Construction*, G. Hedin, Ed., vol. 2622 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003, pp. 138–152.

[74] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 1099–1110.

[75] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Program Interface*, May 2008. `http://openmp.org`.

[76] PIERCE, B. C. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[77] PIERCE, B. C. *Advanced Topics in Types and Programming Languages*. The MIT Press, Dec. 2004.

[78] PLAID-GROUP. Plaid Google Code Repository. `http://code.google.com/p/plaid-lang/`.

[79] QI, X., AND MYERS, A. C. Masked types for sound object initialization. In *Principles of Programming Languages* (2009).

[80] REISTAD, B., AND GIFFORD, D. K. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM conference on LISP and functional programming* (New York, NY, USA, 1994), LFP '94, ACM, pp. 65–78.

[81] SARKAR, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[82] SRINIVASAN, S., AND MYCROFT, A. Kilim: Isolation-typed actors for java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 104–128.

[83] STORK, S., MARQUES, P., AND ALDRICH, J. Concurrency by default: using permissions to express dataflow in stateful programs. In *Onward!* (2009).

[84] STROM, R. E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. 12*, 1 (1986), 157–171.

[85] SUNSHINE, J., NADEN, K., STORK, S., ALDRICH, J., AND TANTER, E. First-class state change in plaid. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011), OOPSLA '11, ACM, pp. 713–732.

[86] SUTTER, H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal 30*, 3 (2005), 16–20.

[87] TERAUCHI, T. Checking race freedom via linear programming. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), ACM, pp. 1–10.

[88] WILLIAMS, S., SHALF, J., OLIKER, L., KAMIL, S., HUSBANDS, P., AND YELICK, K. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers* (New York, NY, USA, 2006), ACM, pp. 9–20.

[89] YANG, Y., GRINGAUZE, A., WU, D., AND ROHDE, H. Detecting data race and atomicity violation via Typestate-Guided static analysis. Tech. Rep. MSR-TR-2008-108, Microsoft Research, Aug. 2008.

[90] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.

Proof of Formal System

A.1 Judgements

Name	Form	Description
Evaluation Judgement	$\mu \vert \delta \vert \Psi \vert \mathcal{G} \vdash e \mapsto e' \dashv \mu_\delta \vert \Psi' \vert \mathcal{G}'$	Given the store (μ), the runtime permissions (δ), the group access tokens (Ψ), the data group configuration ($\mathcal{G}$) the expression e steps to e' and produces a differential store (μ_δ), an updated set of group access tokens (Ψ') and an updated data group configuration ($\mathcal{G}'$).
Typechecking Judgement	$\Gamma \vert \Sigma \vert \Delta \vdash_C e : T \vert \mathcal{G}$	Given the typing context Γ, the store typing Σ, the permission context Δ, the expression e checks in the context of class C with type T and has data group configuration $\mathcal{G}$.
Object Typing	$\mu \vert \Gamma \vert \Sigma \vdash o : T$	Given the heap μ, the typing context Γ and the store typing context Σ the object the location o refers to has type T.
Program State Well-Formedness	$\Gamma \vert \Sigma \vert \Delta \vdash_{wf} (\mu \vert \delta \vert \Psi \vert \mathcal{G} \vert e)$	cf. Definition *Well-Formed Program State*
Type Well-Formedness	$\Gamma \vdash T ok$	Given the typing context Γ we check verify that the given type T corresponds to an existing type and that the number of provided group parameters corresponds with required number of group parameters.
Sub-Typing	$\Gamma \vdash T <: T'$	Checks that T is a proper sub-type if T'.

A.2 Sub-Typing

Standard sub-typing rules. Extended to cover data group parameters.

$$\text{ST-CLASS}\quad \frac{\texttt{class}\ C\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends}\ D\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\}}{\Gamma \vdash C\langle\overline{gr_D},\overline{gr_C}\rangle <: D\langle\overline{gr_D}\rangle}$$

$$\text{ST-REFL}\quad \frac{}{\Gamma \vdash C\langle\overline{gr_C}\rangle <: C\langle\overline{gr_C}\rangle}$$

$$\text{ST-TRANS}\quad \frac{\Gamma \vdash C\langle\overline{gr_C}\rangle <: D\langle\overline{gr_D}\rangle \qquad \Gamma \vdash D\langle\overline{gr_D}\rangle <: E\langle\overline{gr_E}\rangle}{\Gamma \vdash C\langle\overline{gr_C}\rangle <: E\langle\overline{gr_E}\rangle}$$

$$\text{ST-BOTTOM}\quad \frac{}{\Gamma \vdash \bot <: C\langle\overline{gr}\rangle}$$

A.3 Definitions

Definition 1 (Program State) A program state is a quintuple of the form $(\mu|\delta|\Psi|\mathcal{G}|e)$, consisting of a store (μ), a runtime permission context (δ), a group access token context (Ψ) of available tokens, a data group configuration $(\mathcal{G})$ and an expression (e).

Definition 2 (Stuck) An program state $(\mu|\delta|\Psi|\mathcal{G}|e)$ is stuck if e is not a value and:

- $(\mu|\delta|\Psi|\mathcal{G}|e)$ does not take a step (i.e. $(\mu|\delta|\Psi|\mathcal{G}|e) \not\rightarrow (\mu'|\delta'|\Psi'|\mathcal{G}'|e')$ for all $e', \mu', \delta', \Psi', \mathcal{G}'$)
- $(\mu|\delta|\Psi|\mathcal{G}|e)$ does not wait for resources to become available (i.e,. it is waiting for the corresponding group access token to become available)

Definition 3 (Unique Allocation) If multiple expressions simultaneously allocate new objects, then every creation site will get a unique object reference.

Definition 4 (Store Typing) A store μ is said to be *well typed*, written $\Gamma|\Sigma \vdash \mu$, if:

- $dom(\Sigma) = dom(\mu)$
- $\forall o \in dom(\mu) : \mu|\Gamma|\Sigma \vdash o : \Sigma(o)$

OBJ-TYPING

$$\frac{\langle o_r \mapsto obj\rangle \in \mu \qquad obj = C[\overline{f = v_f}] \qquad CT(C) = \texttt{class}\ C\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends}\ E\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \\ fields(C) = \overline{T_f\ f} \qquad \Sigma(o_r) = C\langle\overline{o.gn}\rangle \qquad \overline{\Sigma(v_f) <: [^{\overline{o.gn}}/_{\overline{\alpha,\beta}}][^{o_r}/_{this}]T_f}}{\mu|\Gamma|\Sigma \vdash o_r : C\langle\overline{o.gn}\rangle}$$

Definition 5 (Well-Formed Program State) A program state is well typed, written as $\cdot|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$, if :

- $\cdot|\Sigma|\Delta \vdash e : T\ |\mathcal{G}$
- $\Gamma|\Sigma \vdash \mu$
- If $o.gn \in \delta$ then there exists the corresponding $o.gn : gp \in \Delta$
- $\mu \neq \texttt{race}$
- $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
- $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

Lemma 1 (Weakening) If $\Gamma' \subseteq \Gamma$, $\Sigma' \subseteq \Sigma$ and $\Delta' \subseteq \Delta$ then $\Gamma'|\Sigma'|\Delta' \vdash e : T\ |\mathcal{G}$ implies $\Gamma|\Sigma|\Delta \vdash e : T\ |\mathcal{G}$. Proof straightforward through standard induction.

Lemma 2 (Store Monotonicity) If $\Gamma|\Sigma \vdash \mu$ and $\Sigma \subseteq \Sigma'$ then $\Gamma|\Sigma' \vdash \mu$.

Proof: By induction on typing derivation.

Lemma 3 (Substitution) If $\Gamma, x : T_x, \Gamma'|\Sigma|\Delta \vdash e : T_e \,|\mathcal{G}_e$ and $\Gamma|\Sigma|\Delta \vdash r : T_r \mid \bullet$ and $T_r <: T_x$ then $\Gamma, [^r/_x]\Gamma'|\Sigma|[^r/_x]\Delta \vdash [^r/_x](e : T_e \,|\mathcal{G}_e)$ (with $[^r/_x](e : T_e \,|\mathcal{G}_e)$ being the capture-avoiding substitution).

Proof: By induction on a derivation of the statement $\Gamma, x : T_x, \Gamma'|\Sigma|\Delta \vdash e : T_e \,|\mathcal{G}_e$.

Lemma 4 (Inversion)

- If $\Gamma|\Sigma|\Delta \vdash$ `let` $x = e_1$ `in` $e_2 : T_2|\mathcal{G}$ then $\Delta = \Delta_1, \Delta_R$ and $\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2)$ and $\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1 \,|\mathcal{G}_1$ for some T_1 and $\Gamma, x{:}T_1|\Sigma|\Delta_1, \Delta_R \vdash e_2 : T_2 \,|\mathcal{G}_2$
- If $\Gamma|\Sigma|\Delta \vdash r.f_i : T_i|\mathcal{G}$ then $\Gamma|\Sigma|\Delta \vdash r : D\langle\overline{gr}\rangle$ for some D and $\overline{gr}$ and $\mathcal{G} = \{gr_0\}$ and $T_i\ f_i \in fields(D)$ and $\Delta \vdash gr_o : gp$ with $gp \in \{exclusive, protected\}$.
- If $\Gamma|\Sigma|\Delta \vdash r_v.f_i := r_v : T_v|\mathcal{G}$ then $\Gamma|\Sigma|\Delta \vdash r_v : D\langle\overline{gr}\rangle$ for some D and $\overline{gr}$ and $\mathcal{G} = \{gr_0\}$ and $T_i\ f_i \in fields(D)$ and $\Gamma|\Sigma|\Delta \vdash r_v : T_v \,|\bullet$ with $T_v <: T_i$ and $\Delta \vdash gr_o : gp$ with $gp \in \{exclusive, protected\}$.
- If $\Gamma|\Sigma|\Delta \vdash$ `new` $C\langle\overline{o_g.gn_g}\rangle : T \,|\mathcal{G}$ then $CT(D) =$ `class` $D\langle\overline{\alpha}, \overline{\beta}\rangle$ `extends` $E\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\}$ and $\Gamma|\Sigma \vdash \overline{gr : \mathbb{G}}$ and $T = [^{\overline{o_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}]C\langle\overline{\alpha}, \overline{b}\rangle$ and $\mathcal{G} = \bullet$.
- If $\Gamma|\Sigma|\Delta \vdash r.m\langle\overline{gr}\rangle(\overline{r_p}) : T \,|\mathcal{G}$ then $\Gamma|\Sigma \vdash r : T_r, \overline{p : T_p}, \overline{gr : \mathbb{G}}$ and $\Delta \vdash \overline{gr : gp}$ and $T_r = D\langle\overline{gr_D}\rangle$ and $CT(D) =$ `class` $D\langle\overline{\alpha}, \overline{\beta}\rangle$ `extends` $E\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\}$ and $mdecl(D, m) = T_{result}\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x})\{\ e\ \}$ and $\overline{T_p} <: [^{\overline{gr},\overline{gr_D}}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]T_x$ and $T_r <: [^{\overline{gr},\overline{gr_D}}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]D\langle\overline{\alpha}, \overline{\beta}\rangle$ and $T = [^{\overline{gr},\overline{gr_D}}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]T_{result}$ and $\mathcal{G} = \{\overline{gr}\}$.
- If $\Gamma|\Sigma|\Delta \vdash$ `unpackGroupsOf` r `in` e then $\Gamma|\Sigma \vdash r : C\langle\overline{gr}\rangle$ and $\Delta = \Delta', (gr_0 : qp)$ and $groupDecls(C) = \overline{gn}$ and $\Gamma, (\overline{r.gn : \mathbb{G}})|\Sigma|\Delta', (\overline{r.gn : qp'}) \vdash e : T \mid \mathcal{G}_e$ and $G = (\{gr_o, \overline{r.gn}\} \oplus \mathcal{G}_e)$.
- If $\Gamma|\Sigma|\Delta \vdash_C$ `atomic` $\langle gr\rangle\ e : T \mid \mathcal{G}$ then $\mathcal{G} = (\{gr\} \oplus \mathcal{G}_e)$ and $\Delta = \Delta', (gr : shared)$ and $\Gamma|\Sigma \vdash gr : \mathbb{G}$ and $\Gamma|\Sigma|(\Delta', gr : protected) \vdash_C e : T \mid \mathcal{G}_e$
- If $\Gamma|\Sigma|\Delta \vdash_C$ `inatomic` $\langle gr\rangle\ e : T \mid \mathcal{G}$ then $\mathcal{G} = (\{gr\} \oplus \mathcal{G}_e)$ and $\Delta = \Delta', (gr : shared)$ and $\Gamma|\Sigma \vdash gr : \mathbb{G}$ and $\Gamma|\Sigma|(\Delta', gr : protected) \vdash_C e : T \mid \mathcal{G}_e$
- If $\Gamma|\Sigma|\Delta \vdash_C$ `share` $\langle\overline{gr}\rangle$ `between` $e_1 \parallel e_2 : \bot \mid \mathcal{G}$ then $\{\overline{gp}\} \subseteq \{exclusive, shared\}$ and $\Delta = \Delta_1, \Delta_2, \Delta_r, (\overline{gr : gp})$ and $\Gamma|\Sigma|(\Delta_1, \overline{gr : shared}) \vdash_C e_1 : T_1 \,|\mathcal{G}_1$ and $\Gamma|\Sigma|(\Delta_2, \overline{gr : shared}) \vdash_C e_2 : T_2 \,|\mathcal{G}_2$ and $\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2)$.

Proof: Immediate from the definition of the typing relation.

Lemma 5 (Progress) If $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$ (i.e. a well-formed program state) then either:

- e is a value and $\mathcal{G} = \bullet$
- $\mu|\delta|\Psi|\mathcal{G} \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'$ for some $e', \mu_\delta, \Psi', \mathcal{G}'$
- e stops execution with null-dereference, meaning that the expression e contains a sub expression of the form `null`$.f$.
- e is waiting for resource to become available

Lemma 6 (Preservation) If $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$ with $\Gamma|\Sigma|\Delta \vdash e : T \,|\mathcal{G}$ and $\mu|\delta|\Psi|\mathcal{G} \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'$ and $\mu' = [\mu_\delta]\mu$ then there exists:

- $\Sigma' \supseteq \Sigma$
- T'

such that:

- $\Gamma|\Sigma'|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ with $\Gamma|\Sigma'|\Delta \vdash e' : T' \ |\mathcal{G}'$ and $T' <: T$

Theorem 1 (Type-Safety) If $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$ and $(\mu|\delta|\Psi|\mathcal{G} \vdash e) \mapsto^* (\mu'|\delta'|\Psi'|\mathcal{G}'|e')$ then $\Gamma|\Sigma'|\Delta \vdash_{wf}$ $(\mu'|\delta'|\Psi'|\mathcal{G}'|e')$ and *not* stuck.

A.4 Proofs

A.4.1 Proof Type-Safety

Proof (Type-Safety) by induction on $(\mu|\Psi_{\mathcal{F}}|\Psi_{\mathcal{L}}|\mathcal{G}|e) \mapsto^* (\mu|\Psi'_{\mathcal{F}}|\Psi'_{\mathcal{L}}|\mathcal{G}'|e')$:

Case $[(\mu|\Psi_{\mathcal{F}}|\Psi_{\mathcal{L}}|\mathcal{G}|e) \mapsto (\mu|\Psi_F|\Psi_{\mathcal{L}}|\mathcal{G}|e)]$:
by progess lemma: e $(= e')$ either is value | e takes a step | e causes a null-dereference
$\therefore$ $(\mu|\Psi_{\mathcal{F}}|\Psi_{\mathcal{L}}|\mathcal{G}|e)$ is not stuck

Case $\left[\dfrac{\begin{array}{c}\mu|\Psi_{\mathcal{F}}|\Psi_{\mathcal{L}}|\mathcal{G} \vdash e \mapsto e_1 \dashv (\mu_{\delta_{\mathcal{R}}}, \mu_{\delta_{\mathcal{W}}})|\Psi_{\mathcal{F}1}|\Psi_{\mathcal{L}1}|\mathcal{G}_1 \\ \mu_1 = [\mu_{\delta_{\mathcal{W}}}]\mu \qquad (\mu_1|\Psi_{\mathcal{F}1}|\Psi_{\mathcal{L}1}|\mathcal{G}_1|e_1) \mapsto^* e' \dashv \mu'|\Psi'_{\mathcal{F}}|\Psi'_{\mathcal{L}}|\mathcal{G}'\end{array}}{(\mu|\Psi_{\mathcal{F}}|\Psi_{\mathcal{L}}|\mathcal{G}|e) \mapsto^* (\mu'|\Psi'_{\mathcal{F}}|\Psi'_{\mathcal{L}}|\mathcal{G}'|e')}\right]$:

by preservation lemma: $\Gamma|\Sigma|\Delta \vdash_{prog} (\mu_1|\Psi_{\mathcal{F}_1}|\Psi_{\mathcal{L}_1}|\mathcal{G}_1|e_1)$
by IH on $(\mu_1|\Psi_{\mathcal{F}_1}|\Psi_{\mathcal{L}_1}|\mathcal{G}_1|e_1)$: $(\mu'|\Psi'_{\mathcal{F}_1}|\Psi'_{\mathcal{L}}|\mathcal{G}'|e')$ not stuck

■

A.4.2 Proof of Preservation

Proof (Preservation) by induction on $\mu|\delta|\Psi|\mathcal{G} \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'$

Case E-FIELD-READ :
$\therefore e = o_r.f_i$
$\therefore e' = o_v$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|o_v)$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash o_v : T' \ |\mathcal{G}'$ with $T' <: T$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|o_r.f_i)$
by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash o_r.f : T \mid \mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$T_i\ f_i \in fields(D)$
$o_r : D\langle\overline{gr}\rangle$
$\mathcal{G} = \{gr_0\}$
$gr_o : gp \in \Delta$ with $gp \in \{exclusive, protected\}$

WLOG: let $o'.gn' = gr_0$

by E-FIELD-READ:

$o'.gn' \in \delta$
$\Psi' = \Psi$
$\mathcal{G}' = \bullet$
$\mu_\delta = \langle o_r \mapsto D[\overline{f = v_f}]\rangle$

WLOG: let $\Sigma' = \Sigma$

by STORE-TYPING: $o_v : T_v \in \Sigma$ with $T_v <: T_i$
by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash o_v : T_v \mid \bullet$ (TS1.1)
by CONSTRUCTION: $\mu_\delta \neq \texttt{race} \implies \mu' = [\mu_\delta]\mu \neq \texttt{race}$ (TS1.2)
by CONSTRUCTION: $\mu' = [\mu_\delta]\mu = \mu$ (TS1.3)
by E-FIELD-READ: δ, Δ do not change (TS1.4)
by E-FIELD-READ,AS5,AS6: $\Psi = \Psi'$ (TS1.5, TS1.6)
by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|o_v)$ (TS1)

Case E-Let-12 :
$\therefore e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$
$\therefore e' = \texttt{let}\ x = e_1'\ \texttt{in}\ e_2'$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2')$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : T\ |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2)$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$\Delta = \Delta_1, \Delta_R$

$\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2$

$\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1\ |\mathcal{G}_1$ for some T_1

$\Gamma, x{:}T_1|\Sigma|\Delta_1, \Delta_R \vdash e_2 : T\ |\mathcal{G}_2$

by E-LET-12:

let $\delta_1 = \delta \cap requiredPerms(\mathcal{G}_1)$ be the (sub-)set of permissions that are required by e_1

$\therefore o.gn \in \delta_1 \implies o.gn : gp \in \Delta_1$

$\mu \neq \texttt{race}$ and is well typed with respect to Σ

let $\Psi = \Psi_1, \Psi_2$ with $requiredTokens(e_1) \subseteq \Psi_1$ and $requiredTokens(e_2) \subseteq \Psi_2$

$\therefore o.gn@L \in \Psi_1 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_1$

$\therefore (o.gn@U \in \Psi_1 \vee o.gn@\_ \notin \Psi_1) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_1$

$\therefore o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_2$

$\therefore (o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_2$

$\Gamma|\Sigma|\Delta \vdash e_1 : T_1\ |\mathcal{G}_1$ is well typed

$\therefore\ \Gamma|\Sigma|\Delta_1 \vdash_{wf} (\mu|\delta_1|\Psi_1|\mathcal{G}_1|e_1)$

by IH: on $\Gamma|\Sigma_1|\Delta_1 \vdash_{wf} (\mu_1|\delta_1|\Psi_1|\mathcal{G}_1|e_1)$ with $\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1\ |\mathcal{G}_1$ where $\mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e_1' \dashv \mu_{\delta_1}|\Psi_1'|\mathcal{G}_1'$

for some $\mathcal{G}_1', \Sigma \subseteq \Sigma_1$

(IS1.1-1) $\Gamma|\Sigma_1|\Delta_1 \vdash e_1' : T_1\ |\mathcal{G}_1'$

(IS1.2-1) $\mu_1 \neq \texttt{race}$

(IS1.3-1) μ_1 is well typed with respect to Σ_1

(IS1.4-1) $o.gn \in \delta_1 \implies o.gn : gp \in \Delta_1$

(IS1.5-1) $(o.gn@U \in \Psi_1 \vee o.gn@\_ \notin \Psi_1) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_1'$

(IS1.6-1) $o.gn@L \in \Psi_1 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_1'$

by E-LET-12:

let $\delta_2 = (\delta - \delta_1)$ be the sub-set of permissions that did not go to e_1

$\therefore o.gn \in \delta_2 \implies o.gn : gp \in \Delta_2$

$\mu \neq \texttt{race}$ and is well typed with respect to Σ

let $\Psi = \Psi_1, \Psi_2$ with $requiredTokens(e_1) \subseteq \Psi_1$ and $requiredTokens(e_2) \subseteq \Psi_2$

$\therefore o.gn@L \in \Psi_1 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_1$

$\therefore (o.gn@U \in \Psi_1 \vee o.gn@\_ \notin \Psi_1) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_1$

$\therefore o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_2$

$\therefore (o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_2$

$\Gamma|\Sigma|\Delta \vdash e_2 : T_2\ |\mathcal{G}_2$ is well typed

$\therefore\ \Gamma|\Sigma|\Delta_2 \vdash_{wf} (\mu|\delta_2|\Psi_2|\mathcal{G}_2|e_2)$

by IH: on $\Gamma|\Sigma_2|\Delta_2 \vdash_{wf} (\mu_2|\delta_2|\Psi_2|\mathcal{G}_2|e_2)$ with $\Gamma|\Sigma|\Delta_1 \vdash e_2 : T_1\ |\mathcal{G}_1$ where $\mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e_2' \dashv \mu_{\delta_2}|\Psi_2'|\mathcal{G}_2'$

for some $\mathcal{G}_2', \Sigma \subseteq \Sigma_1$

(IS1.1-2) $\Gamma|\Sigma_2|\Delta_2 \vdash e_2' : T\ |\mathcal{G}_2'$

(IS1.2-2) $\mu_2 \neq \texttt{race}$

(IS1.3-2) μ_2 is well typed with respect to Σ_2

(IS1.4-2) $o.gn \in \delta_2 \implies o.gn : gp \in \Delta_2$

(IS1.5-2) $(o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \in e_2'$

(IS1.6-2) $o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \in e_2'$

by UNIQUE ALLOCATE:

$\Sigma_1 = \Sigma \cup \Sigma_1'$ and $\Sigma_2 = \Sigma \cup \Sigma_2'$ with $dom(\Sigma_1') \cap dom(\Sigma_2') = \bullet$

$\therefore$ let $\Sigma' = \Sigma \cup \Sigma_1 \cup \Sigma_2$

$dom(\Psi_1') \cap dom(\Psi_2') = \bullet$

by E-LET-12: $\Psi' = \Psi_1', \Psi_2'$ with $dom(\Psi_1') \cap dom(\Psi_2)'$

by IS1.5-1, IS1.5-2:

$(o.gn@U \in \Psi' \vee o.gn@\_ \notin \Psi') \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e'$ (TS1.5) by IS1.6-1, IS1.6-2:

$o.gn@L \in \Psi' \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e'$ (TS1.6)

by E-LET-12: $\delta = \delta_1, \delta_2$

$\therefore \nexists o_1, o_2 : o_1 \in dom(\mu_{\delta_1}) \wedge o_2 \in dom(\mu_{\delta_2}) \wedge \Sigma'(o_1) = C\langle o'.gn' \ldots\rangle \wedge \Sigma'(o_2) = D\langle o'.gn' \ldots\rangle$

$\therefore dom(\mu_{\delta_1}) \cap dom(\mu_{\delta_2}) = \bullet$

$\therefore \mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2} \neq \texttt{race}$

$\therefore \mu' = [\mu'_\delta]\mu \neq \texttt{race}$ (TS1.3)

by IS1.3-1,IS1.3-2:

μ_1 is well typed with respect to Σ_1

μ_2 is well typed with respect to Σ_2

$\therefore \mu'$ well typed with respect to Σ' (TS1.2)

by E-LET-12: δ, Δ does not change

$o.gn \in \delta \implies o.gn : gp \in \Delta$ (TS1.4)

by T-LET-12: $\Gamma|\Sigma'|\Delta \vdash \texttt{let}\ x = e'_1\ \texttt{in}\ e'_2 : T\ |\mathcal{G}'$ (TS1.1)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: (TS1)

Case E-LET-1 :
Proof is a sub-case of case E-LET-12, without the e_2 sub-expression step.

Case E-LET-2 :
Proof is a sub-case of case E-LET-12, without the e_1 sub-expression step.

Case E-LET-VALUE :

$\therefore e = \texttt{let}\ x = v\ \texttt{in}\ e_2$

$\therefore e' = [^v/_x]e_2$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|[^v/_x]e_2)$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash [^v/_x]e_2 : T\ |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{let}\ x = v\ \texttt{in}\ e_2)$
by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{let}\ x = v\ \texttt{in}\ e_2 : T\ |\mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$\Delta = \Delta_1, \Delta_R$
$\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2$
$\Gamma|\Sigma|\Delta_1 \vdash v : T_1\ |\mathcal{G}_1$ for some T_1
$\Gamma|\Sigma|\Delta_1, \Delta_R \vdash e_2 : T_2\ |\mathcal{G}_2$

by rule E-LET-VALUE:

$\mathcal{G}' = [^v/_x]\mathcal{G}_2$
$\Psi' = \Psi$
$\mu_\delta = \bullet$

by SUBSTITUTION:

$\Gamma, x : T_1, \Gamma'|\Sigma|\Delta_1, \Delta_R \vdash e_2 : T_2\ |\mathcal{G}_2 \implies \Gamma, [^v/_x]\Gamma'|\Sigma|[^v/_x]\Delta_1, \Delta_R \vdash [^v/_x](e_2 : T_2\ |\mathcal{G}_2)$ (TS1.1)

by E-LET-VALUE: $\mu_\delta = \bullet$
$\therefore \mu' = [\mu_\delta]\mu = \mu \neq \texttt{race}$ (TS1.3)

WLOG: let $\Sigma' = \Sigma$
by AS2: $\mu' = \mu$ is well typed with respect $\Sigma' = \Sigma$ (TS1.2)

by E-LET-VALUE: neither $\delta\Delta$ changes (TS1.4)

by AS5,AS6: $\Psi' = \Psi$ (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.5: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-FIELD-ASSIGN :
$\therefore e = v_r.f_i := o_v$
$\therefore e' = o_v$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|o_v)$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash o_v : T \; |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\, \langle o.gn \rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\, \langle o.gn \rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|v_r.f_i := o_v)$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash v_r.f_i := o_v : T \; |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\, \langle o.gn \rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\, \langle o.gn \rangle \ldots \in e$

by INVERSION:

$T_i\ f_i \in fields(D)$

$o_r : D\langle \overline{gr} \rangle$

$\mathcal{G} = \{gr_0\}$

$gr_o : gp \in \Delta$ with $gp \in \{exclusive, protected\}$

$o_v : T_v$ with $T_v : T_i$

WLOG: let $o'.gn' = gr_0$

by E-FIELD-ASSIGN:

$o'.gn' \in \delta$

$o'.gn'@\_ \in \Psi$

$\Psi' = \Psi$

$\mathcal{G}' = \bullet$

$\mu_\delta = \langle o_r \mapsto D[\overline{f = v_f}] \rangle$

WLOG: let $\Sigma' = \Sigma$

by STORE-TYPING: $o_v : T_v \in \Sigma$ with $T_v <: T_i$

by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash o_v : T_v \; |\bullet$ (TS1.1)

by CONSTRUCTION: $\mu_\delta \neq \texttt{race} \implies \mu' = [\mu_\delta]\mu \neq \texttt{race}$ (TS1.2)

by CONSTRUCTION: $\mu' = [\mu_\delta]\mu = \mu$ (TS1.3)

by E-FIELD-ASSIGN: δ, Δ do not change (TS1.4)

by E-FIELD-ASSIGN,AS5,AS6: $\Psi = \Psi'$ (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-NEW :
$\therefore e = \texttt{new}\ C\langle\overline{v_g.gn_g}\rangle$
$\therefore e' = o_{new}$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|o_{new})$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash o_{new} : T\ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{new}\ C\langle\overline{v_g.gn_g}\rangle)$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{new}\ C\langle\overline{v_g.gn_g}\rangle : T\ |\mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:
$T = [^{o_g.gn_g}/_{\overline{\alpha},\overline{\beta}}]C\langle\overline{\alpha},\overline{b}\rangle$
$\mathcal{G} = \bullet$

by E-NEW:
$o'.gn' \in \delta$
$o'.gn'@\_ \in \Psi$
$groupDecls(C) = \overline{gn_c}$ $\qquad \Psi' = \Psi, \overline{o_{new}.gn_c@U}$
$\mathcal{G}' = \bullet$
$\mu_\delta = \langle o_{new} \mapsto C[\overline{f = \texttt{null}}]\rangle$

by E-NEW:
$\mu_\delta \neq \texttt{race} \implies \mu' = [\mu_\delta]\mu$ (TS1.3)
δ, Δ do not change (TS1.4)

WLOG: let $\Sigma' = \Sigma, o_{new} : T$
by E-TRANS-N: $\mu' = [\mu_\delta]\mu$ is well typed with respect to Σ' (TS1.2)

by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash o_{new} : T\ |\mathcal{G}'$ (TS1.1) by E-NEW,AS5,AS6: $\Psi' = \Psi, \{o_{new}.gn_c@U\}$

newly added access tokens are in unlocked state

$\therefore$ `atomic` $\rightarrow$ `inatomic` transmission could have happened so far (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-CALL :

$\therefore e = v_r.m\langle\overline{v_g.gn_g}\rangle(\overline{v_p})$

$\therefore e' = [\overline{v_g.gn_g}/_{\overline{\alpha},\overline{\beta}}][\overline{v_p}/\overline{x}][v_r/_{this}]e$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|[\overline{v_g.gn_g}/_{\overline{\alpha},\overline{\beta}}][\overline{v_p}/\overline{x}][v_r/_{this}]e_b)$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash [\overline{v_g.gn_g}/_{\overline{\alpha},\overline{\beta}}][\overline{v_p}/\overline{x}][v_r/_{this}]e_b : T \;|\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq$ `race`

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists$ `inatomic` $\langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|v_r.m\langle\overline{v_g.gn_g}\rangle(\overline{v_p}))$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash v_r.m\langle\overline{v_g.gn_g}\rangle(\overline{v_p}) : T \;|\mathcal{G}$

(AS2) $\mu \neq$ `race`

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists$ `inatomic` $\langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \ldots \in e$

by INVERSION:

$\Gamma|\Sigma \vdash r : T_r, \overline{p : T_p}, \overline{gr : \mathbb{G}}$

$\Delta \vdash \overline{gr : gp}$

$T_r = D\langle\overline{gr_D}\rangle$

$CT(D) =$ `class` $D\langle\overline{\alpha},\overline{\beta}\rangle$ `extends` $E\langle\overline{\alpha}\rangle\{\overline{G}\;\overline{F}\;\overline{M}\}$

$\underline{mdecl(D,m) = T_{result}}\; m\langle\overline{gp\;\gamma}\rangle(\overline{T_x\;x})\{\; e\;\}$

$\overline{T_p} <: [\overline{gr},\overline{gr_D}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]T_x$

$T_r <: [\overline{gr},\overline{gr_D}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]D\langle\overline{\alpha},\overline{\beta}\rangle$

$T = [\overline{gr},\overline{gr_D}/_{\overline{\gamma},\overline{\alpha},\overline{\beta}}]T_{result}$

$\mathcal{G} = \{\overline{gr}\}$

by E-CALL:

$\overline{vg_g.gn_g} \in \delta$
$mbody(C, m) = \overline{\alpha}.\overline{x}.e_b \times \mathcal{G}_e$
$\mathcal{G}' = [\overline{v_g.gn_g}/_{\overline{\alpha},\overline{\beta}}][\overline{v_p}/_{\overline{x}}][v_r/_{this}]\mathcal{G}_e$
$\Psi' = \Psi$
$\mu_\delta = \bullet$

WLOG: let $\Sigma' = \Sigma$

by SUBSTITUTION: $\Gamma', \overline{x : T_x}, \texttt{this} : T_r, \overline{gr} : \mathbb{G}, \Gamma'|\Sigma|\Delta \vdash e : T_e \;|\mathcal{G}_e$
$\Rightarrow \Gamma, [\overline{v_g.gn_g}/_{\overline{\alpha},\overline{\beta}}][\overline{v_p}/_{\overline{x}}][v_r/_{this}]\Gamma'|\Sigma|[\overline{v_g.gn_g}/_{\overline{\alpha},\overline{\beta}}][\overline{v_p}/_{\overline{x}}][v_r/_{this}]\Delta \vdash [\overline{v_g.gn_g}/_{\overline{\alpha},\overline{\beta}}][\overline{v_p}/_{\overline{x}}][v_r/_{this}](e : T_{result} \;|\mathcal{G}_e)$
(TS1.1)

by E-CALL: $\mu_\delta = \bullet \implies \mu' = [\mu_\delta]\mu = \mu$

μ' is well typed with respect to Σ' (TS1.2)
$\mu' \neq Race$ (TS1.3)

by E-CALL: δ, Δ do not change (TS1.4)

by E-CALL, AS5, AS6: $\texttt{inatomic} \notin e_b$ because it is a runtime only construct (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-UNPACKGROUPSOF-REPLACE :
$\therefore e = \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e_{sub}$
$\therefore e' = \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e'_{sub}$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|\ \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e'_{sub})$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e'_{sub} : T' \;|\mathcal{G}'$ for some $T' <: T$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|$ `unpackGroupsOf` v_r `in` $e_{sub})$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash$ `unpackGroupsOf` v_r `in` $e_{sub} : T\ |\mathcal{G}$
(AS2) $\mu \neq$ `race`
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists$ `inatomic` $\langle o.gn\rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \ldots \in e$

by INVERSION:
$\Gamma|\Sigma \vdash v_r : D\langle\overline{gr}\rangle$
$\Delta = \Delta', (gr_0 : qp)$
$groupDecls(D) = \overline{gn}$
$\Delta'' = \Delta', (\overline{v_r.gn : qp'})$
$\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash e : T \mid \mathcal{G}_e$
$\mathcal{G} = (\{gr_o, \overline{r.gn}\} \oplus \mathcal{G}_e)$

by E-UNPACKGROUPSOF-REPLACE:
$\mathcal{G} = (\{v'.gn, \overline{v_r.gn}\} \oplus \mathcal{G}_e)$
$\delta = \delta', v'.gn$
$\delta'' = \delta', \overline{v_r.gn}$

Sub-Case T-UNPACKGROUPSOF-SHARED :
$qp \in \{shared, protected\} \Rightarrow qp' = shared$

Sub-Case T-UNPACKGROUPSOF-EXCLUSIVE :
If $qp \in \{exclusive\} \Rightarrow qp' = exclusive$

by IH: on $\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash_{wf} (\mu|\delta''|\Psi|\mathcal{G}_e|e_{sub})$ with $\Gamma|\Sigma|\Delta'' \vdash e_{sub} : T\ |\mathcal{G}_e$ where $\mu|\delta''|\Psi|\mathcal{G}_e \vdash e_{sub} \mapsto e'_{sub} \dashv \mu_\delta|\Psi'|\mathcal{G}'_e$
for some $\mathcal{G}'_2, \Sigma \subseteq \Sigma', T' <: T$
(IS1.1) $\Gamma|\Sigma'|\Delta'' \vdash e'_{sub} : T'\ |\mathcal{G}'_e$
(IS1.2) $\mu' \neq$ `race`
(IS1.3) μ' is well typed with respect to Σ'
(IS1.4) $o.gn \in \delta'' \implies o.gn : gp \in \Delta''$
(IS1.5) $(o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists$ `inatomic` $\langle o.gn\rangle \in e'_{sub}$
(IS1.6) $o.gn@L \in \Psi_2 \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \in e'_{sub}$

by IS1.2, IS1.3, IS1.5, IS1.6: (TS1.2, TS1.3, TS1.5, TS1.6)

by T-UNPACKGROUPSOF-EXCLUSIVE, T-UNPACKGROUPSOF-SHARED, E-UNPACKGROUPSOF-REPLACE:

$\Gamma|\Sigma'|\Delta \vdash$ `unpackGroupsOf` v_r `in` $e'_{sub} : T' \,|\mathcal{G}'$ (TS1.1)

by IS1.4, E-UNPACKGROUPSOF-REPLACE:

$o.gn \in \delta \implies o.gn : gp \in \Delta$ (TS1.1)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-UNPACKGROUPSOF-NONE :

$\therefore e =$ `unpackGroupsOf` v_r `in` e_{sub}

$\therefore e' =$ `unpackGroupsOf` v_r `in` e'_{sub}

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|$ `unpackGroupsOf` v_r `in` $e'_{sub})$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash$ `unpackGroupsOf` v_r `in` $e'_{sub} : T \,|\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq$ `race`

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists$ `inatomic` $\langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|$ `unpackGroupsOf` v_r `in` $e_{sub})$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash$ `unpackGroupsOf` v_r `in` $e_{sub} : T \,|\mathcal{G}$

(AS2) $\mu \neq$ `race`

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists$ `inatomic` $\langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \ldots \in e$

by INVERSION:

$\Gamma|\Sigma \vdash v_r : C\langle\overline{gr}\rangle$

$\Delta = \Delta', (gr_0 : qp)$

$groupDecls(C) = \overline{gn}$

$\Delta'' = \Delta', (\overline{v_r.gn : qp'})$

$\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash e : T \mid \mathcal{G}_e$

$G = (\{gr_o, \overline{r.gn}\} \oplus \mathcal{G}_e)$

WLOG: let $v'.gn = gr_0$

by E-UNPACKGROUPSOF-NONE:

$gr_0 \notin \delta$
$\mathcal{G} = (\{v'.gn\} \oplus \mathcal{G}_e)$
$\mathcal{G}' = (\{v'.gn\} \oplus \mathcal{G}'_e)$

by IH: on $\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash_{wf} (\mu|\delta''|\Psi|\mathcal{G}_e|e_{sub})$ with $\Gamma|\Sigma|\Delta'' \vdash e_{sub} : T \ |\mathcal{G}_e$ where $\mu|\delta''|\Psi|\mathcal{G}_e \vdash e_{sub} \mapsto e'_{sub} \dashv \mu_\delta|\Psi'|\mathcal{G}'_e$

for some $\mathcal{G}'_2, \Sigma \subseteq \Sigma'$
(IS1.1) $\Gamma|\Sigma'|\Delta'' \vdash e'_{sub} : T \ |\mathcal{G}'_e$
(IS1.2) $\mu' \neq \texttt{race}$
(IS1.3) μ' is well typed with respect to Σ'
(IS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta'$
(IS1.5-2) $(o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\ \texttt{inatomic}\ \langle o.gn \rangle \in e'_{sub}$
(IS1.6-2) $o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \in e'_{sub}$

by IS1.2, IS1.3, IS1.5, IS1.6: (IS1.2, IS1.3, IS1.5, IS1.6)

by T-UNPACKGROUPSOF-EXCLUSIVE, T-UNPACKGROUPSOF-SHARED, E-UNPACKGROUPSOF-NONE:

$\Gamma|\Sigma'|\Delta \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e'_{sub} : T \ |\mathcal{G}'$ (IS1.1)

by IS1.4, E-UNPACKGROUPSOF-NONE:

$o.gn \in \delta \implies o.gn : gp \in \Delta$ (IS1.1)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-UNPACKGROUPSOF-VALUE :

$\therefore e = \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ v$
$\therefore e' = v$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|v)$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash v : T \ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e_{sub})$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e_{sub} : T\ |\mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:
$\Gamma|\Sigma \vdash v_r : C\langle\overline{gr}\rangle$
$\Delta = \Delta', (gr_0 : qp)$
$groupDecls(C) = \overline{gn}$
$\Delta'' = \Delta', (\overline{v_r.gn : qp'})$
$\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash e : T\ |\ \mathcal{G}_e$
$G = (\{gr_o, \overline{r.gn}\}\} \oplus \mathcal{G}_e)$

by E-UNPACKGROUPSOF-VALUE:
$\mathcal{G}' = \bullet$
$\mu'_\delta = \bullet$
$\Psi' = \Psi$
WLOG: let $\Sigma' = \Sigma$

by STORE-TYPING: $v : T \in \Sigma$

by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash v : T\ |\bullet$ (TS1.1)

by E-UNPACKGROUPSOF-VALUE:
$\mu_\delta = \bullet \implies \mu' = [\mu_\delta]\mu \neq \texttt{race}$ (TS1.3)
$\mu' = \mu$ is well typed with respect to $\Sigma' = \Sigma$ (TS1.2)

by AS1.4, AS1.5, AS1.6,E-UNPACKGROUPSOF-VALUE: d, Δ, Ψ do not change (TS1.4, TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-ATOMIC-STEP1 :
Follows the reasoning as the *E-UnpackGroupsOf-None* case, by allowing the sub-expression to execute code that does not depend on the aotmic permission.

Case E-ATOMIC-STEP2 :
Analog to case *E-Atomic-Step1.* Despite the fact that we have the necessary permission the data group access token indicate the already another atomic block is executing. Therefore only allow sub-expression only to execute code that does not depend on the atomic permission.

Case E-ATOMIC-INATOMIC :
$\therefore e = \texttt{atomic}\ \langle gr \rangle\ e_{sub}$
$\therefore e' = \texttt{inatomic}\ \langle gr \rangle\ e'_{sub}$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|\ \texttt{inatomic}\ \langle gr \rangle\ e_{sub})$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash\ \texttt{inatomic}\ \langle gr \rangle\ e_{sub} : T\ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{atomic}\ \langle gr \rangle\ e_{sub})$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{atomic}\ \langle gr \rangle\ e_{sub} : T\ |\mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$

by INVERSION:
$\mathcal{G} = (\{gr\} \oplus \mathcal{G}_e)$
$\Delta = \Delta', (gr : shared)$
$\Gamma|\Sigma \vdash gr : \mathbb{G}$
$\Gamma|\Sigma|(\Delta', gr : protected) \vdash_C e_{sub} : T\ |\ \mathcal{G}$

by E-ATOMIC-INATOMIC:
$\Psi = \Psi'', gr@U$
$\Psi' = \Psi'', gr@L$
$\mathcal{G}' = \mathcal{G}$
$\mu_\delta = \bullet$
WLOG: let $\Sigma' = \Sigma$

by AS2, E-ATOMIC-INATOMIC: $\mu_\delta = \bullet \implies \mu' = [\mu_\delta]\mu$

$\mu' = \mu \neq \texttt{race}$ (TS1.3)

$\mu' = \mu$ is well typed with respect to $\Sigma' = \Sigma$ (TS1.2)

by T-INATOMIC:

$\Gamma|\Sigma'|\Delta \vdash \texttt{inatomic}\ \langle gr\rangle\ e_{sub} : T\ |\mathcal{G}'$ (TS1.1)

by E-ATOMIC-INATOMIC: δ, Δ do not change (TS1.4)

by AS1,AS5,AS6:

$gr@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle gr\rangle \ldots \in e$

$gr@\_ \notin \Psi'' \implies \nexists\, \texttt{inatomic}\ \langle gr\rangle \ldots \in e_{sub}$

$gr@L \in \Psi' \implies \nexists\, \texttt{inatomic}\ \langle gr\rangle \ldots \in e'$ (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-INATOMIC-STEP1 :
Follows a similar logic as *E-Atomic-Step2*. In this case the all permissions are passed to the sub-expression let the sub-expression take a step.

Case E-INATOMIC-VALUE :
$\therefore e = \texttt{inatomic}\ \langle gr\rangle\ v$
$\therefore e' = v$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|v)$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash v : T\ |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{inatomic}\ \langle gr\rangle\ e_{sub})$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash \texttt{inatomic}\ \langle gr\rangle\ e_{sub} : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$\mathcal{G} = (\{gr\} \oplus \mathcal{G}_e)$
$\Delta = \Delta', (gr : shared)$
$\Gamma|\Sigma \vdash gr : \mathbb{G}$
$\Gamma|\Sigma|(\Delta', gr : protected) \vdash_C v : T \mid \mathcal{G}$

by E-INATOMIC-VALUE:

$gr \in \delta$
$\Psi = \Psi'', gr@L$
$\Psi' = \Psi''', gr@U$
$\mu_\delta = 0$
$\mathcal{G}' = \bullet$

WLOG: let $\Sigma' = \Sigma$

by STORE-TYPING: $v : T \in \Sigma$

by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash v : T \mid\bullet$ (TS1.1)

by E-INATOMIC-VALUE:

$\mu_\delta = \bullet \implies \mu' = [\mu_\delta]\mu \neq \texttt{race}$ (TS1.3)
$\mu' = \mu$ is well typed with respect to $\Sigma' = \Sigma$ (TS1.2)

by AS1.4 E-UNPACKGROUPSOF-VALUE: d, Δ, Ψ do not change (TS1.4)

by AS1,AS5,AS6:

$gr@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle gr \rangle \ldots \in e$
$gr@U \in \Psi' \implies \nexists\ \texttt{inatomic}\ \langle gr \rangle \ldots \in e'$ (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-SPLIT-12 :
$\therefore e =$ $\texttt{share}\ \langle \overline{gr} \rangle\ \texttt{between}\ e_1 \parallel e_2$
$\therefore e' =$ $\texttt{share}\ \langle \overline{gr} \rangle\ \texttt{between}\ e_1' \parallel e_2'$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|\ \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e_1' \parallel e_2')$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash\ \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e_1' \parallel e_2' : T\ |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e_1 \parallel e_2)$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e_1 \parallel e_2 : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$\{\overline{gp}\} \subseteq \{exclusive, shared\}$

$\Delta = \Delta_1, \Delta_2, \Delta_r, (\overline{gr : gp})$

$\Gamma|\Sigma|(\Delta_1, \overline{gr : shared}) \vdash_C e_1 : T_1\ |\mathcal{G}_1$

$\Gamma|\Sigma|(\Delta_2, \overline{gr : shared}) \vdash_C e_2 : T_2\ |\mathcal{G}_2$

$\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2)$

by E-SPLIT-12:

$\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2)$

$\delta_1 = \delta \cap required(\mathcal{G}_1)$ the sub-set of permission that are required by e_1

$\delta_2 = \delta \cap required(\mathcal{G}_2)$ the sub-set of permission that are required by e_2

$\Psi = \Psi_1, \Psi_2$ with $requiredTokens(e_1) \subseteq \Psi_1$ and $requiredTokens(e_2) \subseteq \Psi_2$

$\Psi' = \Psi_1', \Psi_2'$

$\mathcal{G}' = (\mathcal{G}_1' \parallel \mathcal{G}_2')$

by ASSUMPTION:

$\Gamma|\Sigma|\Delta_1 \vdash_{wf} (\mu|\delta_1|\Psi_1|\mathcal{G}_1|e_1)$

$\Gamma|\Sigma|\Delta_2 \vdash_{wf} (\mu|\delta_2|\Psi_2|\mathcal{G}_2|e_2)$

by IH: on $\Gamma|\Sigma_1|\Delta_1 \vdash_{wf} (\mu_1|\delta_1|\Psi_1|\mathcal{G}_1|e_1)$ with $\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1 \ |\mathcal{G}_1$ where $\mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e_1' \dashv \mu_{\delta_1}|\Psi_1'|\mathcal{G}_1'$

for some $\mathcal{G}_1', \Sigma \subseteq \Sigma_1$

(IS1.1-1) $\Gamma|\Sigma_1|\Delta_1 \vdash e_1' : T_1 \ |\mathcal{G}_1'$

(IS1.2-1) $\mu_1 \neq \texttt{race}$

(IS1.3-1) μ_1 is well typed with respect to Σ'

(IS1.4-1) $o.gn \in \delta_1 \implies o.gn : gp \in \Delta_1$

(IS1.5-1) $(o.gn@U \in \Psi_1 \vee o.gn@\_ \notin \Psi_1) \implies \nexists\, \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e_1'$

(IS1.6-1) $o.gn@L \in \Psi_1 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e_1'$

by IH: on $\Gamma|\Sigma_2|\Delta_2 \vdash_{wf} (\mu|\delta_2|\Psi_2|\mathcal{G}_2|e_2)$ with $\Gamma|\Sigma|\Delta_1 \vdash e_2 : T_1 \ |\mathcal{G}_1$ where $\mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e_2' \dashv \mu_{\delta_2}|\Psi_2'|\mathcal{G}_2'$

for some $\mathcal{G}_2', \Sigma \subseteq \Sigma_1$

(IS1.1-2) $\Gamma|\Sigma_2|\Delta_2 \vdash e_2' : T \ |\mathcal{G}_2'$

(IS1.2-2) $\mu_2 \neq \texttt{race}$

(IS1.3-2) μ_2 is well typed with respect to Σ'

(IS1.4-2) $o.gn \in \delta_2 \implies o.gn : gp \in \Delta_2$

(IS1.5-2) $(o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\, \texttt{inatomic}\ \langle o.gn \rangle \in e_2'$

(IS1.6-2) $o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \in e_2'$

by UNIQUE ALLOCATE:

$\Sigma_1 = \Sigma \cup \Sigma_1'$ and $\Sigma_2 = \Sigma \cup \Sigma_2'$ with $dom(\Sigma_1') \cap dom(\Sigma_2') = \bullet$

$\therefore$ let $\Sigma' = \Sigma \cup \Sigma_1 \cup \Sigma_2$

$dom(\Psi_1') \cap dom(\Psi_2') = \bullet$

by E-SPLIT-12: $\Psi' = \Psi_1', \Psi_2'$

by IS1.5-1, IS1.5-2:

$(o.gn@U \in \Psi' \vee o.gn@\_ \notin \Psi') \implies \nexists\, \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e'$ (TS1.5)

by IS1.6-1, IS1.6-2:

$o.gn@L \in \Psi' \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e'$ (TS1.6)

by E-SPLIT-12: $\delta = \delta_1, \delta_2$

$\therefore \nexists o_1, o_2 : o_1 \in dom(\mu_{\delta_1}) \wedge o_2 \in dom(\mu_{\delta_2}) \wedge \Sigma'(o_1) = C\langle o'.gn' \ldots \rangle \wedge \Sigma'(o_2) = D\langle o'.gn' \ldots \rangle$

$\therefore dom(\mu_{\delta_1}) \cap dom(\mu_{\delta_2}) = \bullet$

$\therefore \mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2} \neq \texttt{race}$

$\therefore \mu' = [\mu_\delta']\mu \neq \texttt{race}$ (TS1.3) by IS1.2-1,IS1.2-2:

μ_1 is well typed with respect to Σ_1
μ_2 is well typed with respect to Σ_2
$\therefore \mu'$ well typed with respect to Σ' (TS1.2)

by E-SPLIT-12: δ, Δ does not change
$o.gn \in \delta \implies o.gn : gp \in \Delta$ (TS1.4)

by T-SPLIT-12: $\Gamma|\Sigma'|\Delta \vdash e' : T\ |\mathcal{G}'$ (TS1.1)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: (TS1)

Case E-SPLIT-1 :
Follows the same approach as case *E-Split-12* with the difference that the evaluation of e_2 is not considered.

Case E-SPLIT-2 :
Follows the same approach as case *E-Split-12* with the difference that the evaluation of e_1 is not considered.

Case E-SPLIT-VALUE :
Follows the same approach as case *E-UnpackGroupsOf-Value*.

■

A.4.3 Proof of Progress

Proof (Progress) by induction on $\Gamma|\Sigma|\Delta \vdash_C e : T\ |\mathcal{G}$.

Case T-UNPACKGROUPSOF-EXCLUSIVE :
$e =$ `unpackGroupsOf` r `in` e_1
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Case e_1 is value :
by E-UNPACKGROUPSOF-VALUE: $\mu|\delta|\Psi|\mathcal{G} \vdash$ `unpackGroupsOf` r `in` $v_1 \mapsto v_1 \dashv \bullet|\Psi|\bullet$
$\therefore e \mapsto e'$ takes a step

Sub-Case e_1 takes a step (with s) :
by E-UNPACKGROUPSOF-REPLACE:
$\mu|\delta|\Psi|\mathcal{G} \vdash$ `unpackGroupsOf` r `in` $e_1 \mapsto$ `unpackGroupsOf` r `in` $e_1' \dashv \mu_\delta|\Psi'|G'$
$\therefore e \mapsto e'$ takes a step

Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-UNPACKGROUPSOF-SHARED :
Symmetric to the T-UNPACKGROUPSOF-EXCLUSIVE case.

Case T-SPLIT :
$e = \texttt{split}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2$
by GRAMMAR: $r_i = \texttt{null}$ or $r_i = o$

Sub-Case $\exists i : r_i = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $\forall i : r_i \neq \texttt{null}$:
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Sub-Case e_1 is value :
by IH: e_2 is value | e_2 takes a step | e_2 stops with null-dereference | e_2 waits for resources

Sub-Sub-Sub-Case e_2 is value :
by E-SPLIT-VALUE:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{split}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ v_1 \parallel v_2 \mapsto \texttt{null} \dashv \bullet|\Psi|\bullet \therefore e \mapsto e'$ takes a step

Sub-Sub-Sub-Case $e_2 \mapsto e_2'$ takes a step :
by E-SPLIT-2:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{split}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto \texttt{split}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2' \dashv \mu_\delta|\mathcal{G}';|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Sub-Case e_2 stops with null-dereference :
Then e stops with null-dereference.

Sub-Sub-Sub-Case e_2 waits for resources :
Then e waits for resources.

Sub-Sub-Case $e_1 \mapsto e_1'$ takes a step :
by E-SPLIT-1:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{split}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto \texttt{split}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1' \parallel e_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-ATOMIC :
$e =$ `atomic` $\langle r.gn \rangle\ e_1$
by GRAMMAR: $r =$ `null` or $r = o$

Sub-Case : $r =$ `null` :
Then e stops with null-dereference.

Sub-Case $r \neq$ `null` :
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Sub-Case e_1 is value :
by E-ATOMIC-INATOMIC:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{atomic}\ \langle r.gn \rangle\ v_1 \mapsto \texttt{inatomic}\ \langle r.gn \rangle v_1 \dashv \bullet|\Psi'|\bullet$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-INATOMIC :
$e =$ `inatomic` $\langle r.gn \rangle\ e_1$
by GRAMMAR: $r =$ `null` or $r = o$
by T-INATOMIC: $\mathcal{G} = (\{r.gn\} \oplus \mathcal{G}_e)$

Sub-Case $r =$ `null` :
Then e stops with null-dereference.

Sub-Case $r \neq$ `null` :
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Sub-Case e_1 takes a step (with $\mu|\delta|\Psi \backslash \{r.gn@L\}|\mathcal{G}_e \vdash e_1 \to e_1' \dashv \mu_\delta|\Psi'|\mathcal{G}_e'$) :
by E-INATOMIC-STEP:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{inatomic}\ \langle r.gn \rangle\ e_1 \mapsto \texttt{inatomic}\ \langle r.gn \rangle\ e_1' \dashv \mu_\delta|\Psi', r.gn@L|(\{r.gn\} \oplus \mathcal{G}_e')$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case e_1 is value :
by E-INATOMIC-VALUE:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{inatomic}\ \langle r.gn \rangle\ v_1 \mapsto v_1 \dashv \bullet|\Psi'|\bullet$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-LET :
$e =$ `let` $x = e_1$ `in` e_2
by T-LET: $\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2)$
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Case e_1 is value :
by IH: e_1 is value $\implies G_1 = \bullet$
by E-LET-VALUE:
$\mu|\delta|\Psi|\mathcal{G} \vdash$ `let` $x = v_1$ `in` $e_2 \mapsto [^{v_1}/_{e_2}] \dashv \bullet|\Psi|\bullet$
$\therefore e \mapsto e'$ takes a step

Sub-Case $e_1 \mapsto e_1'$ takes a step :
by T-LET: $requiredTokens(e) = requiredTokens(e_1) \cup requireTokens(e_2)$
by T-LET: $requiredTokens(e) \subseteq \Psi$
let $\Psi_1 = \Psi$
by E-LET-1:
$\mu|\delta|\Psi|\mathcal{G} \vdash$ `let` $x = e_1$ `in` $e_2 \mapsto$ `let` $x = e_1$ `in` $e_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-REFERENCE :
$e = r$
by GRAMMAR: $r =$ `null` or $r = o$

Sub-Case $r =$ `null` :
Then e stops with null-dereference.

Sub-Case $r \neq$ `null` :
The e is value.

Case T-FIELD-READ :
$\therefore e = r.f_i$
by GRAMMAR: $r =$ `null` or $r = o$
by T-FIELD-READ: $\Gamma|\Sigma \vdash r : D\langle\overline{gr}\rangle$
$\therefore \mu(r) = D[\overline{f = v}]$

Sub-Case $r =$ `null` :
Then e stops with null-dereference.

Sub-Case $r \neq$ `null` :

Sub-Sub-Case $r.gn \in \delta$:
by T-FIELD-READ: $\Gamma|\Sigma \vdash r : D\langle gr\rangle$ and $fields(D) = \overline{T_f f}$
by E-FIELD-READ:
$\mu|\delta|\Psi|\mathcal{G} \vdash r.f_i \mapsto v_i \dashv \mu_\delta|\Psi|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case $r.gn \notin \delta$:
Then e is waiting for resources.

Case T-FIELD-ASSIGN :
$\therefore e = r.f_i := v$
by GRAMMAR: $r = \texttt{null}$ or $r = o$
by T-FIELD-ASSIGN: $\Gamma|\Sigma \vdash r : D\langle\overline{gr}\rangle$
$\therefore \mu(r) = D[\overline{f = r_f}]$

Sub-Case $r = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $r \neq \texttt{null}$:

Sub-Sub-Case $r.gn \in \delta$:
by E-FIELD-READ:
$\mu|\delta|\Psi|\mathcal{G} \vdash r.f_i := r_v \mapsto v_i \dashv \mu_\delta|\Psi|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case $r.gn \notin \delta$:
Then e is waiting for resources.

Case T-NEW :
$e = \texttt{new}\ C\langle\overline{r.gn}\rangle()$
by GRAMMAR: $r_i = \texttt{null}$ or $r_i = o$

Sub-Case $\exists i : r_i = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $\forall i : r_i \neq \texttt{null}$:
by E-NEW:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{new}\ C\langle\overline{r.gn}\rangle() \mapsto o \dashv \mu_\delta|\Psi'|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Case T-CALL :
$e = r_r.m\langle r_g.gn\rangle(r_p)$
by GRAMMAR: $r_i \in \{r_r, \overline{r_g}\} \implies r_i = \texttt{null}$ or $r_i = o$

Sub-Case $\exists i : r_i = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $\forall i : r_i \neq \texttt{null}$:

Sub-Sub-Case $\exists r_g \notin \delta$:
Then e waits for resources.

Sub-Sub-Case $\forall r_g \in \delta$:
by E-CALL:
$\mu|\delta|\Psi|\mathcal{G} \vdash r_r.m\langle r_g.gn\rangle(r_p) \mapsto [^{\overline{r_g.gn}}/_{\overline{\alpha}}][^{\overline{r_p}}/_{\overline{x}}][^{r_r}/_{this}]e_b \dashv \bullet|\Psi|\mathcal{G}_b$
$\therefore e \mapsto e'$ takes a step

■

B.1 Base Line Performance Data

Plaid		Java		Java Script	
global	**member**	**global**	**member**	**global**	**member**
3.375	5.879	1.059	1.182	13.390	13.245
3.415	5.903	1.054	1.148	13.243	13.198
3.356	5.795	1.054	1.163	13.488	13.865
3.313	5.877	1.093	1.156	13.500	13.512
3.420	5.890	1.063	1.141	13.581	13.182
3.294	5.813	1.079	1.149	13.730	13.432
3.361	5.797	1.095	1.171	13.508	13.458
3.350	5.971	1.121	1.139	13.720	13.279
3.386	6.013	1.095	1.163	13.555	13.184
3.354	5.876	1.087	1.150	13.445	13.228
3.365	5.852	1.069	1.149	13.306	13.196
3.412	5.863	1.083	1.158	13.438	14.002
3.370	5.872	1.080	1.161	13.302	13.155
3.344	5.928	1.086	1.170	13.408	13.111
3.351	5.887	1.054	1.140	13.284	13.637
3.342	5.901	1.062	1.204	13.706	13.280
3.451	5.833	1.041	1.145	13.578	13.218
3.389	5.802	1.042	1.150	13.739	13.186
3.368	5.878	1.091	1.172	13.443	13.285
3.379	5.791	1.049	1.133	13.816	13.269
3.369	**5.871**	**1.073**	**1.157**	**13.509**	**13.346**

B.2 Webserver Performance Data

Plaid	**ÆMINIUM**	**Java (Sequential)**	**Java (Parallel)**
46.40	38.10	50.78	30.27
46.18	38.64	48.31	32.51
49.82	40.13	50.15	30.45
47.47	42.12	49.21	31.19
51.81	33.79	49.09	31.91
48.14	38.60	50.10	30.31
49.48	34.73	50.04	30.69
52.79	38.46	45.08	31.27
51.47	42.22	51.45	30.91
52.24	31.99	46.41	30.49
51.48	34.81	48.54	32.67
47.32	36.29	47.07	32.98
50.98	42.23	48.60	31.36
46.06	37.42	47.48	30.89
47.69	35.98	48.43	31.23
47.82	42.09	44.72	30.80
49.45	31.93	47.47	31.34
47.48	32.00	46.29	31.67
49.37	41.84	48.77	30.57
49.41	34.24	51.95	32.07
49.14	**37.38**	**48.50**	**31.28**

B.3 Dictionary Performance Data

global/unique		global/shared		fine/unique		fine/shared	
init	**check**	**init**	**check**	**init**	**check**	**init**	**check**
7.989	7.059	8.849	8.004	5.266	4.487	1.379	0.944
7.724	7.060	8.722	7.958	5.355	4.684	1.400	0.940
7.916	7.251	8.641	7.987	5.378	4.737	1.405	0.946
7.667	7.149	8.693	8.024	5.282	4.509	1.376	0.939
8.088	7.300	8.734	8.176	5.364	4.925	1.387	0.944
8.108	7.248	8.882	4.250	5.212	4.402	1.386	0.942
7.946	7.075	8.691	8.019	5.315	4.606	1.396	0.947
7.860	7.203	9.175	4.128	5.434	4.676	1.374	0.938
7.855	7.096	8.557	7.928	5.394	4.667	1.394	0.945
8.104	7.203	8.845	7.902	5.336	4.711	1.387	0.941
7.983	7.239	8.847	3.905	5.423	4.713	1.374	0.931
7.965	7.149	8.723	7.997	5.273	4.468	1.412	0.943
8.109	7.009	8.866	4.127	5.403	4.478	1.399	0.937
7.755	7.079	8.697	7.856	5.475	4.718	1.398	0.935
7.860	7.349	8.736	8.030	5.239	4.553	1.367	0.939
7.978	7.136	8.726	7.794	5.229	4.687	1.380	0.939
7.908	6.872	8.826	3.996	5.409	4.669	1.386	0.939
7.950	7.190	8.784	7.893	5.244	4.526	1.395	0.937
8.028	7.259	8.689	4.088	5.417	4.765	1.375	0.932
7.802	7.126	8.833	7.871	5.147	4.492	1.392	0.936
7.769	7.265	8.859	4.305	5.562	4.661	1.407	0.942
7.955	7.253	8.727	7.954	5.384	4.701	1.393	0.941
8.047	7.135	8.831	7.912	5.187	4.475	1.418	0.949
7.935	7.253	8.581	7.947	5.315	4.486	1.391	0.932
8.125	7.263	8.964	4.323	5.480	4.714	1.392	0.924
8.036	7.045	8.912	4.095	5.667	4.693	1.375	0.960
7.647	7.203	8.670	7.856	5.378	4.711	1.366	0.930
7.830	7.164	8.821	4.170	5.208	4.365	1.385	0.939
8.033	7.200	8.828	4.055	5.396	4.548	1.420	0.946
7.906	7.308	8.796	4.035	5.473	4.601	1.421	0.945
8.091	7.335	8.995	4.163	5.463	4.673	1.384	0.945
7.920	7.174	8.674	7.934	5.439	4.710	1.390	0.946
8.027	7.124	8.972	4.165	5.076	4.478	1.381	0.935
8.105	7.236	8.689	7.871	5.324	4.607	1.383	0.950
7.926	7.173	8.765	7.845	5.200	4.509	1.410	0.930
7.784	7.189	8.815	8.015	5.463	4.683	1.382	0.928
7.944	7.305	8.665	3.988	5.157	4.417	1.406	0.946
8.084	7.286	8.782	7.973	5.151	4.509	1.379	0.930
7.925	7.285	8.787	4.048	5.515	4.762	1.383	0.927
7.969	7.172	8.749	7.917	5.334	4.469	1.384	0.937
7.850	7.202	8.867	4.261	5.400	4.648	1.395	0.930
7.873	7.091	8.815	4.153	5.219	4.486	1.405	0.932
7.945	7.327	8.683	8.077	5.405	4.697	1.421	0.948
7.993	7.285	8.606	7.938	5.218	4.451	1.376	0.937
7.908	7.189	8.789	7.889	5.432	4.734	1.400	0.943
7.661	7.202	8.649	7.916	5.123	4.511	1.414	0.929
8.016	7.196	8.790	7.860	5.238	4.448	1.390	0.938
7.928	7.332	8.821	5.194	5.299	4.680	1.380	0.940
7.820	7.243	8.819	3.970	5.143	4.503	1.389	0.934
8.052	7.401	8.741	4.077	5.356	4.739	1.380	0.926
7.933	7.197	8.779	6.356	5.332	4.602	1.391	0.938

B.4 Integral Performance Data

Plaid	ÆMINIUM
8.755	4.318
8.624	4.292
9.062	4.087
9.157	4.107
8.777	4.306
8.905	3.995
8.752	4.291
8.597	4.289
8.577	4.200
9.325	3.791
8.849	3.989
8.857	4.106
8.670	4.295
8.795	4.192
9.012	4.309
9.139	4.330
8.760	3.990
8.738	3.989
8.895	4.198
8.847	4.098
8.855	4.159

B.5 ForkJoin Performance Data

Plaid	ÆMINIUM (sequential)	ÆMINIUM	ÆMINIUM +Oracle	Plaid/Threaded
11.902	12.597	2.494	2.118	1.960
12.000	12.994	2.532	1.976	2.021
11.968	12.584	2.405	2.167	2.027
12.041	13.609	2.418	2.012	2.060
11.870	12.094	2.399	2.017	2.020
12.103	13.206	2.507	2.089	2.008
11.979	12.797	2.500	1.975	2.092
11.989	12.696	2.428	1.995	2.011
12.073	12.501	2.396	2.110	1.962
11.981	12.297	2.398	2.098	2.031
12.036	12.375	2.398	2.197	1.983
11.960	13.592	2.506	2.097	1.985
12.263	12.280	2.503	2.021	2.105
11.889	12.382	2.423	2.109	2.024
12.202	12.783	2.403	2.035	1.902
12.122	12.295	2.404	2.105	2.047
11.996	12.602	2.412	2.017	1.938
12.107	12.386	2.404	1.985	2.042
12.243	12.477	2.522	2.043	2.054
12.253	12.883	2.395	2.080	1.964
12.049	12.671	2.442	2.062	2.012

Source Code

C.1 Base Line Examples

C.1.1 Plaid

Listing C.1: Plaid's Base Line Benchmarks

```
package plaid.examples.baseline.plaid;

state BaseLine {
    method void forkJoin(immutable Integer level) [ immutable BaseLine this ] {
        val immutable Boolean isZero = level == 0;

        match ( isZero ) {
            case False {
                val immutable Integer nextLevel = level  1;
                this.forkJoin(nextLevel);
                this.forkJoin(nextLevel);
            }
            default { unit }
        }
    }
}


method void forkJoin(immutable Integer level) {
    val immutable Boolean isZero = level == 0;

    match ( isZero ) {
        case False {
            val immutable Integer nextLevel = level  1;
            forkJoin(nextLevel);
            forkJoin(nextLevel);
        }
        default { unit }
    }
}
```

C.1.2 Java

Listing C.2: Java's Member Base Line Benchmark

```
package plaid.examples.baseline.java;

public class ClassBaseLine {

        public void forkJoin(int level){
                boolean isZero = level == 0;
                if ( isZero == false ) {
                        int nextLevel = level  1;
                        this.forkJoin(nextLevel);
                        this.forkJoin(nextLevel);
                } else {}
        }

        public static void main(String[] args) {
                ClassBaseLine cbl = new ClassBaseLine();
                long begin = System.nanoTime();
                cbl.forkJoin(28);
                long end = System.nanoTime();
                long delta = end  begin;
                double divider = 1000∗1000∗1000.0;
                double result = delta / divider;
                System.out.printf(”%.3f\n”, result);
        }

}
```

Listing C.3: Java's Global Base Line Benchmark

```
package plaid.examples.baseline.java;

public class StaticBaseLine {

        public static void forkJoin(int level) {
                boolean isZero = level == 0;
                if ( isZero == false ) {
                        int nextLevel = level  1;
                        forkJoin(nextLevel);
                        forkJoin(nextLevel);
                } else {}
        }

        public static void main(String[] args) {
                long begin = System.nanoTime();
                forkJoin(28);
                long end = System.nanoTime();
                long delta = end  begin;
                double divider = 1000∗1000∗1000.0;
                double result = delta / divider;
                System.out.printf(”%.3f\n”, result);
        }

}
```

C.1.3 Java Script

Listing C.4: Java Script Member Base Line Benchmark

```
function BaseLine() {

};

BaseLine.prototype.forkJoin = function (level) {
    var isZero = level == 0;
    if ( isZero == false ) {
        var nextLevel = level  1;
        this.forkJoin(nextLevel);
        this.forkJoin(nextLevel);
    }
};

var begin = new Date();
var bl = new BaseLine();
bl.forkJoin(28);
var end = new Date();
console.log(”” + ((end.getTime()begin.getTime())/1000.0));
```

Listing C.5: Java Script Global Base Line Benchmark

```
function forkJoin(level) {
    var isZero = level == 0;
    if ( isZero == false ) {
        var nextLevel = level  1;
        forkJoin(nextLevel);
        forkJoin(nextLevel);
    }
}

var begin = new Date();
forkJoin(28)
var end = new Date();
console.log(”” + ((end.getTime()begin.getTime())/1000.0));
```

C.2 Webserver Example

Listing C.6: Webserver `main.plaid`

```
package plaid.examples.webserver.plaid;

method void main() {
    val immutable WebServer ws = new WebServer;
    ws.run();
}
```

Listing C.7: Webserver `WebServer.plaid`

```
package plaid.examples.webserver.plaid;

state WebServer {

        @sequential
        method immutable String getRoot() [immutable WebServer this] {
        java.lang.System.getProperty(”user.dir”) +”/www/”;
        }

        method void run() [immutable WebServer this] {
                val unique ServerSocket ss = new ServerSocket;

                this.acceptLoop(ss);
        }

        method void acceptLoop(unique ServerSocket serverSocket) [immutable WebServer this] {
                val unique Socket client = serverSocket.accept();
                this.serveClient(client);
                this.acceptLoop(serverSocket);
        }

        @sequential
        method immutable Boolean fileExists(immutable String path) [immutable WebServer this] {
                val file = java.io.File.new(this.getRoot() + path);

                file.exists() && file.isFile() && file.canRead()
        }

        method void serveClient(unique Socket client) [immutable WebServer this] {

                // get request string
                val immutable ?String request = client.readLine();
                match (request) {
                    case String {

                    val immutable Boolean isGet = request.toLowerCase().startsWith(”get␣”);
                        match ( isGet ) {
                                case True {
                                        // compute path
                                        val immutable Integer requestLength = request.length();
                                        val immutable String reqSuffix = request.substring(4, requestLength);
                                        val immutable Integer indexSpace = reqSuffix.indexOf(”␣”);
                                        val immutable Integer indexParam = reqSuffix.indexOf(”?”);
                          val immutable Boolean possitiveIndexSpace = indexSpace >= 0;
                          val immutable Boolean possitiveIndexParam = indexParam >= 0;
                          val immutable Integer index = match (possitiveIndexSpace) {
                              case True {
                                  match (possitiveIndexParam) {
                                      case True {
                                          val immutable Boolean spaceSmallerThanParam = indexParam >= indexSpace;
                                          match ( spaceSmallerThanParam ) {
                                              case True { indexSpace }
                                              default { indexParam }
                                          }
                                      }
                                      default { indexSpace }
                                  };
                              }
                              default { indexParam }
                          };
                          val immutable Boolean possitiveIndex = index >= 0;
                                          match ( possitiveIndex ) {
                                                  case True {
                                                          val immutable String path = reqSuffix.substring(0, index);
                                                          this.transferFile(client, path);
                                                  }
                                                  default {
                                                          this.transferFile(client, reqSuffix);
                                                  }
                                          }
                                }
                                case False { }
                        };
                }
                default { /∗ connection closed ∗/ }
        };

                client.close();
        }

        @sequential
        method void transferData(unique Socket client, immutable String path) [immutable WebServer this] {
            val immutable Boolean fileExists = this.fileExists(path);
                match ( fileExists ) {
                        case True {
                val file = java.io.File.new(this.getRoot() + path);
                client.copyFileToSocket(file);
                        }
                        default {
                                client.writeLine("<html><body><h1>404␣File␣not␣found</h1></body></html>");
                        }
                }
        }

        method void transferFile(unique Socket client, immutable String path) [immutable WebServer this] {
            val immutable Boolean isIndex = path == "/";
                match ( isIndex ) {
                        case True {
                                this.transferHeader(client, "index.html");
                                this.transferData(client, "index.html");
                        }
                        default {
                                this.transferHeader(client, path);
                                this.transferData(client, path);
                        }
                };
                client.flush();
            }

            method void transferHeader(unique Socket client, immutable String path) [immutable WebServer this] {
                val immutable Boolean fileExists = this.fileExists(path);
                    match ( fileExists ) {
                        case True {
                            client.writeLine("HTTP/1.1␣200␣Script␣output␣follows");

                            // HTML
                            val immutable Boolean isHTML = path.endsWith(".html");
                            match ( isHTML ) {
                                case True { client.writeLine("ContentType:␣text/html;␣charset=UTF8"); }
                                default { /∗ nop ∗/ }
                            };

                            // CSS
                            val immutable Boolean isCSS = path.endsWith(".css");
                            match ( isCSS ) {
                                case True { client.writeLine("ContentType:␣text/css;␣charset=UTF8"); }
                                default { /∗ nop ∗/ }
                            };

                            // Java Script
                          val immutable Boolean isJS = path.endsWith(".js");
                            match ( isJS ) {
                                case True { client.writeLine("ContentType:␣text/javascript;␣charset=UTF8"); }
                                default { /∗ nop ∗/ }
                            };

                    // Jpeg
                    val immutable Boolean isJPEG = path.endsWith(".jpg") || path.endsWith(".jpeg");
                    match ( isJPEG ) {
                        case True { client.writeLine("ContentType:␣image/jpeg"); }
                        default { /∗ nop ∗/ }
                    };

                    // PNG
                    val immutable Boolean isPNG = path.endsWith(".png");
                    match ( isPNG ) {
                        case True { client.writeLine("ContentType:␣image/png"); }
                        default { /∗ nop ∗/ }
                    };

                    // GIF
                    val immutable Boolean isGIF = path.endsWith(".gif");
                    match ( isGIF ) {
                        case True { client.writeLine("ContentType:␣image/gif"); }
                        default { /∗ nop ∗/ }
                    };

                            // close connection
```

```
                        client.writeLine("Connection:␣close");

                        // separator to content
                        client.writeLine("");
                    }
                    default {
                        client.writeLine("HTTP/1.1␣404␣File␣not␣found");
                        client.writeLine("");
                    }
            }
        }
}
```

Listing C.8: Webserver `ServerSocket.plaid`

```
package plaid.examples.webserver.plaid;

state ServerSocket {
    @sequential
    val ss = java.net.ServerSocket.new(8000);

    @sequential
    method unique Socket accept() [ unique ServerSocket this ] {
        val client = this.ss.accept();

        val inputReader = java.io.InputStreamReader.new(client.getInputStream());
        val reader = java.io.BufferedReader.new(inputReader);

        new Socket { val socket = client; val inputReader = reader; val outputStream = client.getOutputStream(); }
    }
}
```

Listing C.9: Webserver `Socket.plaid`

```
package plaid.examples.webserver.plaid;

state Socket {
    val socket;
    val inputReader;
    val outputStream;

    @sequential
    method immutable ?String readLine() [unique Socket this] {
        this.inputReader.readLine();
    }

    @sequential
    method void writeLine(immutable String msg) [unique Socket this] {
        this.outputStream.write((msg + "\n").getBytes());
    }

    @sequential
    method void flush() [unique Socket this] {
        this.outputStream.flush();
    }

    @sequential
    method void close() [unique Socket this] {
       this.socket.close();
    }

    @sequential
    method void copyFileToSocket(file) [immutable Socket this] {
        val is = java.io.BufferedInputStream.new(java.io.FileInputStream.new(file));
        plaid.examples.webserver.java.Webserver.copyFileToSocket(this.outputStream, is);
        is.close();
    }
}
```

C.3 Integral Example

Listing C.10: Integral `main.plaid`

```
package plaid.examples.integral.plaid;

method void main() {
    val unique Runner runner = new Runner;

    runner.run();
}
```

Listing C.11: Integral `Integral.plaid`

```
package plaid.examples.integral.plaid;

state Integral {
    method immutable Float64 compute(immutable Float64 x1, immutable Float64 x2) [immutable Integral this] {
        val immutable Float64 delta = x2  x1;

        val immutable Boolean divide = delta.nativeLessThan(0.00000001);
        match ( divide ) {
            case True {
                val immutable Float64 f1 = this.f(x1);
                val immutable Float64 f2 = this.f(x2);
                val immutable Float64 combinedf = f1 + f2;
                val immutable Float64 avgf = combinedf / 2.0;
                val immutable Float64 area = avgf * delta;

                area
            }
            default {
                val immutable Float64 combinedx = x1 + x2;
                val immutable Float64 middle = combinedx / 2.0;
                val immutable Float64 area1 = this.compute(x1, middle);
                val immutable Float64 area2 = this.compute(middle, x2);

                area1 + area2
            }
        }
    }

    @cheap
    method immutable Float64 f(immutable Float64 x) [immutable Integral this];
}
```

Listing C.12: Integral `SquareIntegral.plaid`

```
package plaid.examples.integral.plaid;


state SquareIntegral case of Integral {
    @cheap
    method immutable Float64 f(immutable Float64 x) [immutable SquareIntegral this] {
        x ∗ x
    }
}
```

Listing C.13: Integral `Runner.plaid`

```
package plaid.examples.integral.plaid;

state Runner {
    @sequential
    method void stdout(immutable Float64 area) {
        val formater = java.text.DecimalFormat.new("#.####");

        java.lang.System.out.println(formater.format(area));
    }

    method void run() [unique Runner this] {
        val immutable SquareIntegral si = new SquareIntegral;
        val immutable Float64 area = si.compute(0.0, 1.0);

        this.stdout(area);
    }
}
```

C.4 Dictionary Example

Listing C.14: Dictionary `Hashmap.plaid`

```
package plaid.examples.lib.hashmap;

state Hashmap {
    method void addShared(immutable Hashable key, immutable Object value) [ local shared Hashmap this ];
    method void addUnique(immutable Hashable key, immutable Object value) [ unique Hashmap this ];
    method immutable Boolean containsShared(immutable Hashable obj) [ local shared Hashmap this ];
    method immutable Boolean containsUnique(immutable Hashable obj) [ unique Hashmap this ];
}
```

Listing C.15: Dictionary `Hashable.plaid`

```
package plaid.examples.lib.hashmap;

state Hashable case of Object {
    method immutable Integer hash() [ immutable Hashable this ] ;
}
```

C.4.1 Global Dictionary

Listing C.16: Global Dictionary `GlobalHashmap.plaid`

```
package plaid.examples.lib.hashmap.global;

import plaid.examples.lib.hashmap.Hashable;
import plaid.examples.lib.hashmap.Hashmap;

import plaid.arrays.GlobalSharedArray;

state GlobalHashmap case of Hashmap {
    val immutable Integer bucketCount;
    val immutable GlobalHashmapOperations addOps;
    val immutable GlobalHashmapOperations containsOps;
    val unique GlobalSharedArray buckets;

    override method void addUnique(immutable Hashable key, immutable Object data) [ unique GlobalHashmap this ] {
        val immutable Integer index = key.hash() % this.bucketCount;
        val immutable GlobalHashmapOperations bops = this.addOps;
        this.buckets.doUniqueData2(index, bops, key, data);
    }

    override method void addShared(immutable Hashable key, immutable Object data) [ local shared GlobalHashmap this ] {
        val immutable Integer index = key.hash() % this.bucketCount;
        val immutable GlobalHashmapOperations bops = this.addOps;
        this.buckets.doLocalSharedData2(index, bops, key, data);
    }
    override method immutable Boolean containsUnique(immutable Hashable key) [ unique GlobalHashmap this ] {
        val immutable Integer index = key.hash() % this.bucketCount;
        val immutable GlobalHashmapOperations bops = this.containsOps;
        this.buckets.doUniqueData1(index, bops, key);
    }

    override method immutable Boolean containsShared(immutable Hashable key) [ local shared GlobalHashmap this ] {
        val immutable Integer index = key.hash() % this.bucketCount;
        val immutable GlobalHashmapOperations bops = this.containsOps;
        this.buckets.doLocalSharedData1(index, bops, key);
    }

}
```

Listing C.17: Global Dictionary `GlobalHashmapOperations.plaid`

```
package plaid.examples.lib.hashmap.global;

import plaid.arrays.AbstractGlobalSharedArrayOperations;

state GlobalHashmapOperations case of AbstractGlobalSharedArrayOperations {
    method shared Object initialize(immutable Integer index) [ immutable GlobalHashmapOperations this ] {
        new Bucket
    }
}
```

Listing C.18: Global Dictionary `GlobalHashmapAddOperations.plaid`

```
package plaid.examples.lib.hashmap.global;

state GlobalHashmapAddOperations case of GlobalHashmapOperations {

    @sequential
    override method immutable Boolean doSharedData2(shared ?Object obj, immutable Object key, immutable Object value)
                                                    [ immutable GlobalHashmapAddOperations this ]{
        obj.addShared(key, value);
    }
}
```

Listing C.19: Global Dictionary `GlobalHashmapContainsOperations.plaid`

```
package plaid.examples.lib.hashmap.global;

state GlobalHashmapContainsOperations case of GlobalHashmapOperations {

    @sequential
    override method immutable Boolean doSharedData1(shared ?Object obj, immutable Object key)
                                                    [ immutable GlobalHashmapContainsOperations this ]{
        obj.containsShared(key);
    }
}
```

Listing C.20: Global Dictionary `Bucket.plaid`

```
package plaid.examples.lib.hashmap.global;

import plaid.examples.lib.hashmap.Hashable;

state Bucket case of Object {
    // bucket list
    var shared ?BucketList bucketList = unit;

    override method immutable Boolean addShared(immutable Hashable key, immutable Object value) [ shared Bucket this] {
        atomic {
            val shared ?BucketList head = this.bucketList;

            match ( head ) {
                case BucketList {
                    // check for existing entry
                    val immutable Boolean found = head.containsShared(key.hash());
                    match ( found ) {
                        case False {
                            val shared BucketList newHead = new BucketList;
                            newHead.keyHash = key.hash();
                            newHead.value = value;
                            newHead.next = head;
                            this.bucketList = newHead;
                            new True
                        }
                        default {
                            new False
                        }
                    }
                }
                default {
                    // add first element
                    val shared BucketList newHead = new BucketList;
                    newHead.keyHash = key.hash();
                    newHead.value = value;
                    this.bucketList = newHead;
                    new True
                }
            }
        }
    }

    override method immutable Boolean containsShared(immutable Hashable key) [ shared Bucket this] {
        atomic {
            val shared ?BucketList head = this.bucketList;

            match ( head ) {
                case BucketList {
                    // check for existing entry
                    head.containsShared(key.hash());
                }
                default { new False }
            }
        }
    }
}
```

Listing C.21: Global Dictionary `BucketList.plaid`

```
package plaid.examples.lib.hashmap.global;

state BucketList {

    var shared ?BucketList next = unit;

    @sequential
    var immutable Object value = unit;
    var immutable Integer keyHash = 0;

    method immutable Boolean containsShared(immutable Integer objHash) [ shared BucketList this ] {
        atomic {
            val immutable Integer thisHash = this.keyHash;
            val immutable Boolean found = thisHash == objHash;

            match ( found ) {
                case False {
                    val shared ?BucketList next = this.next;
                    match ( next ) {
                        case BucketList {
                            //printLine("[BucketList] search rest of list.");
                            next.containsShared(objHash)
                        }
                        default {
                            //printLine("[BucketList] end of list.");
                            found
                        }
                    }
                }
                default {
                    //printLine("[BucketList] found element.");
                    found
                }
            }
        }
    }

}
```

Listing C.22: Global Dictionary `package.plaid`

```
package plaid.examples.lib.hashmap.global;

import plaid.arrays.GlobalSharedArray;
import plaid.arrays.makeGlobalSharedArray;
import plaid.examples.lib.hashmap.Hashmap;

/*****************************************************************************
 ** factory methods
 *****************************************************************************/

@sequential
method unique Hashmap makeGlobalHashmap(immutable Integer order) {
    val immutable Integer bucketCount = 1 << order;
    val immutable GlobalHashmapOperations addOps = new GlobalHashmapAddOperations;
    val immutable GlobalHashmapOperations containsOps = new GlobalHashmapContainsOperations;
    val unique GlobalSharedArray sa = makeGlobalSharedArray(order);
    sa.initialize(addOps);
    new GlobalHashmap {
        val immutable Integer bucketCount = bucketCount;
        val immutable GlobalHashmapOperations addOps = addOps;
        val immutable GlobalHashmapOperations containsOps = containsOps;
        val unique GlobalSharedArray buckets = sa;
    }
}
```

C.4.2 Fine Dictionary

Listing C.23: Fine Dictionary `FineHashmap.plaid`

```
package plaid.examples.lib.hashmap.fine;

import plaid.examples.lib.hashmap.Hashable;
import plaid.examples.lib.hashmap.Hashmap;

import plaid.arrays.SharedArray;

state FineHashmap case of Hashmap {
    val immutable Integer bucketCount;
    val immutable FineHashmapOperations addOps;
    val immutable FineHashmapOperations containsOps;
    val unique SharedArray buckets;

    override method void addUnique(immutable Hashable key, immutable Object data) [ unique FineHashmap this ] {
        val immutable Integer index = key.hash() % this.bucketCount;
        val immutable FineHashmapOperations bops = this.addOps;
        this.buckets.doUniqueData2(index, bops, key, data);
    }

    override method void addShared(immutable Hashable key, immutable Object data) [ local shared FineHashmap this ] {
        val immutable Integer index = key.hash() % this.bucketCount;
        val immutable FineHashmapOperations bops = this.addOps;
        this.buckets.doLocalSharedData2(index, bops, key, data);
    }

    override method immutable Boolean containsUnique(immutable Hashable key) [ unique FineHashmap this ] {
        val immutable Integer index = key.hash() % this.bucketCount;
        val immutable FineHashmapOperations bops = this.containsOps;
        this.buckets.doUniqueData1(index, bops, key);
    }

    override method immutable Boolean containsShared(immutable Hashable key) [ local shared FineHashmap this ] {
        val immutable Integer index = key.hash() % this.bucketCount;
        val immutable FineHashmapOperations bops = this.containsOps;
        this.buckets.doLocalSharedData1(index, bops, key);
    }

}
```

Listing C.24: Fine Dictionary `FineHashmapOperations.plaid`

```
package plaid.examples.lib.hashmap.fine;

import plaid.arrays.AbstractSharedArrayOperations;

state FineHashmapOperations case of AbstractSharedArrayOperations {
    method shared<owner> Object initialize<group exclusive owner>(immutable Integer index) [ immutable FineHashmapOperations this ] {
        new Bucket<owner>
    }
}
```

Listing C.25: Fine Dictionary `FineHashmapAddOperations.plaid`

```
package plaid.examples.lib.hashmap.fine;

state FineHashmapAddOperations case of FineHashmapOperations {

    @sequential
    override method immutable Boolean doExclusiveData2<group exclusive owner>
                    (shared<owner> ?Object obj, immutable Object key, immutable Object value)
                    [ immutable FineHashmapAddOperations this ] {
        obj.addExclusive<owner>(key, value);
    }

    @sequential
    override method immutable Boolean doSharedData2<group shared owner>
                    (shared<owner> ?Object obj, immutable Object key, immutable Object value)
                    [ immutable FineHashmapAddOperations this ]{
        obj.addShared<owner>(key, value);
    }
}
```

Listing C.26: Fine Dictionary `FineHashmapContainsOperations.plaid`

```
package plaid.examples.lib.hashmap.fine;

state FineHashmapContainsOperations case of FineHashmapOperations {
    @sequential
    override method immutable Boolean doExclusiveData1<group exclusive owner>
                            (shared<owner> ?Object obj, immutable Object key)
                            [ immutable FineHashmapContainsOperations this ] {
        obj.containsExclusive<owner>(key);
    }

    @sequential
    override method immutable Boolean doSharedData1<group shared owner>
                            (shared<owner> ?Object obj, immutable Object key)
                            [ immutable FineHashmapContainsOperations this ]{
        obj.containsShared<owner>(key);
    }
}
```

Listing C.27: Fine Dictionary `Bucket.plaid`

```
package plaid.examples.lib.hashmap.fine;

import plaid.examples.lib.hashmap.Hashable;

state Bucket<group Owner> case of Object {
    // bucket list
    var shared<Owner> ?BucketList<Owner> bucketList = unit;

    override method immutable Boolean addExclusive<group exclusive owner>
                            (immutable Hashable key, immutable Object value) [ shared<owner> Bucket<owner> this] {
        val shared<owner> ?BucketList<owner> head = this.bucketList;

        match ( head ) {
            case BucketList<owner> {
                // check for existing entry
                val immutable Boolean found = head.containsExclusive<owner>(key.hash());
                match ( found ) {
                    case False {
                        val shared<owner> BucketList<owner> newHead = new BucketList<owner>;
                        newHead.keyHash = key.hash();
                        newHead.value = value;
                        newHead.next = head;
                        this.bucketList = newHead;
                        new True
                    }
                    default {
                        new False
                    }
                }
            }
            default {
                // add first element \
                val shared<owner> BucketList<owner> newHead = new BucketList<owner>;
                newHead.keyHash = key.hash();
                newHead.value = value;
                this.bucketList = newHead;
                new True
            }
        }
    }

    override method immutable Boolean addShared<group shared owner>
                            (immutable Hashable key, immutable Object value) [ shared<owner> Bucket<owner> this] {
        atomic<owner> {
            val shared<owner> ?BucketList<owner> head = this.bucketList;

            match ( head ) {
                case BucketList<owner> {
                    // check for existing entry
                    val immutable Boolean found = head.containsProtected<owner>(key.hash());
                    match ( found ) {
                        case False {
                            val shared<owner> BucketList<owner> newHead = new BucketList<owner>;
                            newHead.keyHash = key.hash();
                            newHead.value = value;
                            newHead.next = head;
                            this.bucketList = newHead;
                            new True
                        }
                        default {
                            new False
                        }
                    }
                }
                default {
                    // add first element
                    val shared<owner> BucketList<owner> newHead = new BucketList<owner>;
                    newHead.keyHash = key.hash();
                    newHead.value = value;
                    this.bucketList = newHead;
                    new True
                }
            }
        }
    }

    override method immutable Boolean containsExclusive<group exclusive owner>
                            (immutable Hashable key) [ shared<owner> Bucket<owner> this] {
        val shared<owner> ?BucketList<owner> head = this.bucketList;

        match ( head ) {
            case BucketList<owner> {
                // check for existing entry
                head.containsExclusive<owner>(key.hash());
            }
            default { new False }
        }
    }
    override method immutable Boolean containsShared<group shared owner>
                    (immutable Hashable key) [ shared<owner> Bucket<owner> this] {
        atomic<owner> {
            val shared<owner> ?BucketList<owner> head = this.bucketList;

            match ( head ) {
                case BucketList<owner> {
                    // check for existing entry
                    head.containsProtected<owner>(key.hash());
                }
                default { new False }
            }
        }
    }
}
```

Listing C.28: Fine Dictionary `BucketList.plaid`

```
package plaid.examples.lib.hashmap.fine;

state BucketList <group Owner>{

    var shared<Owner> ?BucketList<Owner> next = unit;

    @sequential
    var immutable Object value = unit;
    var immutable Integer keyHash = 0;

    method immutable Boolean containsExclusive<group exclusive Owner>
                    (immutable Integer objHash) [ shared<Owner> BucketList<Owner> this ] {
        val immutable Integer thisHash = this.keyHash;
        val immutable Boolean found = thisHash == objHash;

        match ( found ) {
            case False {
                val shared<Owner> ?BucketList<Owner> next = this.next;
                match ( next ) {
                    case BucketList<Owner> {
                        next.containsExclusive<Owner>(objHash)
                    }
                    default {
                        found
                    }
                }
            }
            default {
                found
            }
        }
    }

    method immutable Boolean containsProtected<group protected Owner>
                    (immutable Integer objHash) [ shared<Owner> BucketList<Owner> this ] {
```

```
            val immutable Integer thisHash = this.keyHash;
            val immutable Boolean found = thisHash == objHash;

            match ( found ) {
                case False {
                    val shared<Owner> ?BucketList<Owner> next = this.next;
                    match ( next ) {
                        case BucketList<Owner> {
                            next.containsProtected<Owner>(objHash)
                        }
                        default {
                            found
                        }
                    }
                }
                default { found }
            }
        }
}
```

Listing C.29: Fine Dictionary `package.plaid`

```
package plaid.examples.lib.hashmap.fine;

import plaid.arrays.SharedArray;
import plaid.arrays.makeSharedArray;
import plaid.examples.lib.hashmap.Hashmap;

/*****************************************************************************
 ** factory methods
 *****************************************************************************/

@sequential
method unique Hashmap makeFineHashmap(immutable Integer order) {
    val immutable Integer bucketCount = 1 << order;
    val immutable FineHashmapOperations addOps = new FineHashmapAddOperations;
    val immutable FineHashmapOperations containsOps = new FineHashmapContainsOperations;
    val unique SharedArray sa = makeSharedArray(order);
    sa.initialize(addOps);
    new FineHashmap {
        val immutable Integer bucketCount = bucketCount;
        val immutable FineHashmapOperations addOps = addOps;
        val immutable FineHashmapOperations containsOps = containsOps;
        val unique SharedArray buckets = sa;
    }
}
```

C.5 ForkJoin Example

Listing C.30: Integral `main.plaid`

```
package plaid.examples.forkJoin;

state FJ {
    method void forkJoin(immutable Integer level) [local immutable FJ this] {
        val immutable Boolean isZero = level == 0;

        match ( isZero ) {
            case False {
                val immutable Integer currentLevel = level  1;
                this.forkJoin(currentLevel);
                this.forkJoin(currentLevel);
            }
            case True { unit }
        }
    }
}

method void main() {
    val unique FJ fj = new FJ;
    fj.forkJoin(29);
}
```