Æminium Compilation
Theory in the Context of
the Plaid Language

Diploma Thesis by

Manuel Mohr

presented to Department of Informatics

Referee: Prof. Jonathan Aldrich, Ph.D.
Co-Referee: Prof. Dr.-Ing. Gregor Snelting
Supervisor: Dipl.-Inform. Sven Stork

September 1, 2010 – February 24, 2011
Due to the recent technology shift towards multicore architectures, writing parallel applications has become a necessity. However, many existing programming models make it very difficult to write correct and maintainable concurrent code. The Æminium project aims to solve these problems by using an approach based on access permissions. This thesis describes the integration of Æminium’s concepts into the Plaid programming language. Furthermore, the design of a code generation strategy and its implementation in the context of a prototype compiler for Æminium are discussed. The quality of the generated code is evaluated and initial performance measurements are conducted.
Declaration of Authorship

I hereby declare and confirm that this thesis is entirely the result of my own work except where otherwise indicated.

Heilbronn, February 24, 2011  Manuel Mohr


Heilbronn, den 24. Februar 2011  Manuel Mohr
This work was partially supported by the Portuguese Research Agency – FCT, through the CMU|Portugal program (R&D Project Æminium CMU-PT/SE/0038/2008).
## Contents

1 Introduction .......................................................... 1  
   1.1 Motivation ....................................................... 1  
   1.2 Overview ....................................................... 2  

2 Background .......................................................... 5  
   2.1 Plaid ............................................................. 5  
      2.1.1 Typestate .................................................. 5  
      2.1.2 Access Permissions ........................................ 8  
      2.1.3 Gradual Typing ............................................ 15  
      2.1.4 Concurrency by Default .................................. 16  
   2.2 Æminium .......................................................... 17  
      2.2.1 Making Implicit Dependencies Explicit ...................... 17  
      2.2.2 Unique and Immutable .................................... 18  
      2.2.3 Shared Permissions ....................................... 20  
      2.2.4 Data Groups ............................................... 21  
   2.3 Æminium Runtime ................................................ 25  
   2.4 Related Work ................................................... 26  

3 Integration .......................................................... 29  
   3.1 Dependency Analysis ............................................ 29  
   3.2 The Plaid Type System .......................................... 34  
      3.2.1 The Core Language Syntax .................................. 35  
      3.2.2 Plaid Types ............................................... 37  
      3.2.3 Permissions ............................................... 38  
      3.2.4 Subtyping ................................................ 39  
   3.3 Analyzing the Type-Annotated AST ............................... 40  
   3.4 Dependency Analysis Examples ................................... 45  
   3.5 Address Permissions ............................................ 48  
   3.6 Typing of Lambdas ............................................. 49  
   3.7 Borrowing for Local Variables .................................. 52  
   3.8 Java Interoperability .......................................... 54  
   3.9 Syntax Changes ................................................ 55  
   3.10 Tracking Data Groups ......................................... 61  
   3.11 Code Generation & Granularity ................................ 63  

4 Implementation ...................................................... 67  
   4.1 The Plaid Runtime .............................................. 68  
   4.2 The Æminium Runtime .......................................... 71  
   4.3 Dependency Analysis ............................................ 73  
   4.4 Code Generation ................................................ 74  
   4.5 Granularity Management ......................................... 78
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Program consisting of two concurrent activities.</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Possible interleavings.</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Abstract state machine for BufferedReader.</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Unique access permission.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Immutable access permission.</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Shared access permission.</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Full access permission.</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Pure access permission.</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Legal permission split rules.</td>
<td>12</td>
</tr>
<tr>
<td>2.8</td>
<td>Permission flow graph for example from listing 2.5.</td>
<td>13</td>
</tr>
<tr>
<td>2.9</td>
<td>Legal rules for fractional permissions combining splitting and joining.</td>
<td>14</td>
</tr>
<tr>
<td>2.10</td>
<td>Permission flow graph demonstrating fractional permissions.</td>
<td>15</td>
</tr>
<tr>
<td>2.11</td>
<td>Æminium pipeline.</td>
<td>17</td>
</tr>
<tr>
<td>2.12</td>
<td>Permission flow inside the <code>histogramEqualization</code> method.</td>
<td>20</td>
</tr>
<tr>
<td>2.13</td>
<td>Different types of permissions in Æminium.</td>
<td>25</td>
</tr>
<tr>
<td>2.14</td>
<td>Work stealing.</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>General overview.</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Permission flow graphs for eager and lazy joining.</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Dependency graph for example from table 3.2.</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Dependency graph for example from table 3.3.</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Type system notation conventions.</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Linear dependencies.</td>
<td>63</td>
</tr>
<tr>
<td>3.7</td>
<td>Putting $A$, $B$ and $C$ into the same task reduces parallelism.</td>
<td>63</td>
</tr>
<tr>
<td>3.8</td>
<td>Putting $A$, $B$ and $C$ into the same task reduces parallelism.</td>
<td>64</td>
</tr>
<tr>
<td>3.9</td>
<td>Call graph for fib(4).</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Plaid compiler pipeline.</td>
<td>67</td>
</tr>
<tr>
<td>4.2</td>
<td>Æminium compiler pipeline.</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>Workflow inside the Æminium compiler.</td>
<td>68</td>
</tr>
<tr>
<td>4.4</td>
<td>Part of the Plaid runtime class hierarchy.</td>
<td>69</td>
</tr>
<tr>
<td>4.5</td>
<td>The PlaidScope class hierarchy.</td>
<td>70</td>
</tr>
<tr>
<td>4.6</td>
<td>Æminium runtime interfaces.</td>
<td>72</td>
</tr>
<tr>
<td>4.7</td>
<td>AST of <code>foo</code>’s body with dependencies.</td>
<td>74</td>
</tr>
<tr>
<td>4.8</td>
<td>Possible task partitioning.</td>
<td>75</td>
</tr>
<tr>
<td>4.9</td>
<td>Generated tasks upon execution of <code>foo</code>.</td>
<td>77</td>
</tr>
<tr>
<td>4.10</td>
<td>Lambda interface in the Plaid runtime.</td>
<td>78</td>
</tr>
<tr>
<td>4.11</td>
<td>Generated tasks upon execution of <code>bar</code>.</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>Task graph for the Fibonacci example from listing 5.1.</td>
<td>83</td>
</tr>
</tbody>
</table>
5.2 Task graph for main method. ............................. 84
5.3 Task graph for base case. ................................. 84
5.4 Task graph for recursing branch. .......................... 85
5.5 Speedup of Æminium code relative to sequential Plaid code. . 86
List of Tables

2.1 Access permission taxonomy. 10
3.1 Access operation compatibility. 31
3.2 Example 1. 33
3.3 Example 2. 34
1 Introduction

1.1 Motivation

Recent developments in computer architecture have brought multicore processors into the mainstream. Because it was no longer feasible to improve the performance of single core chip designs by increasing the clock frequency, chip vendors started to integrate multiple cores on a single chip. Thus, technology that once had only been used in the high performance computing sector was suddenly common even in low-end laptops. This radical change of hardware design did not leave the software development process unaffected.

In the world of single core processors, programmers could trust that their programs would automatically run faster on the next generation of processors, mainly because of the ever increasing clock frequency. However, with the advent of parallel platforms this is not true any more. As Sutter’s famous quote “The free lunch is over” [Sut05] expresses very concisely, programmers have to invest significant effort in order to fully utilize the resources of these hardware platforms. More specifically, applications need to distribute their work to keep multiple cores busy. This distribution is usually realized by the creation of multiple threads that each process a part of the overall task. But this is a far-reaching change in program structure and often requires the redesign of a substantial part of the program.

Moreover, multithreaded software is also prone to new kinds of defects. Among the most common bugs are race conditions. A program contains a race condition if it is possible that at least two threads access the same location in memory at the same time while at least one of the threads has the intent to modify the memory content. Figure 1.1 shows an example that illustrates the problem.

![Figure 1.1: Program consisting of two concurrent activities.](image-url)
Here, thread 1 increments the shared variable $X$ by one whereas thread 2 decrements it by 1. Assuming an initial value of 0 for $X$, the expected value of $X$ after both threads are finished is 0 again because the effects of the threads cancel each other out. However, as indicated by figure 1.1(b), the increment and decrement operations are not atomic. They consist of a suboperation that reads the value of $X$ and saves it to a thread-local variable $t_i$ and a second suboperation that writes the updated value to $X$. Because the individual suboperations of the threads can be interleaved in various fashions, the end result can differ and thus the program output is not deterministic any more. Figure 1.2 shows three possible interleavings and the associated results.

This small example demonstrates a multitude of problems with the current state of concurrent programming models. Firstly, the programmer has to explicitly introduce and manage the parallelism in their code. As indicated by the code snippet in figure 1.1(a), work has to be distributed to concurrent tasks manually. Additionally, threads must often be created and set up by calling library functions, which makes the resulting code substantially harder to understand.

Moreover, there is no automatic protection against data races. If the programmer does not correctly synchronize conflicting accesses to shared state, the program contains a race condition. In many programming models, there are neither static nor dynamic means to support the developer in preventing or finding race conditions.

### 1.2 Overview

The Æminium project aims to provide a solution to both problems. It frees the programmer from the task of creating threads and assigning work to them and places it in the hands of the compiler and an accompanying runtime system. Furthermore, it extends the notion of a well-typed program to also guarantee data race freedom. This means that well-typed Æminium programs do not expose nondeterministic behavior because of data races.
Æminium achieves this by utilizing access permissions. Access permissions are a way of providing static information about aliasing and access rights to the compiler and are therefore a source of valuable additional information. The Plaid programming language tightly integrates access permissions and thus is a natural fit for evaluating the Æminium approach.

One of the key concepts behind Æminium’s design is the decoupling of compiler and underlying hardware. Therefore, an additional layer in the form of a data flow oriented runtime system, called the Æminium runtime, is introduced. Its purpose is to execute parts of the program that have previously been identified by the compiler as independent with maximum concurrency.

The goal of this thesis is to investigate how the idea of the Æminium project can be realized in the context of Plaid. This encompasses the integration of Æminium’s concepts into Plaid as well as the design of a code generation strategy that targets the Æminium runtime library. A prototype implementation on the basis of the existing Plaid compiler is to be developed. This makes a first evaluation of the approach possible.

In chapter 2, the Plaid programming language, the ideas behind Æminium and the general design of Æminium’s runtime system are introduced. Moreover, the Æminium approach is compared to other projects in the broad area of automatic parallelization and safe concurrent programming.

In chapter 3, the problem of analyzing the dependencies between the expressions in the program is discussed. A general approach is presented which is then concretized and applied to the Plaid type system. Furthermore, the interaction of Plaid language elements with Æminium is investigated.

In chapter 4, more detailed information about the realization of an Æminium prototype compiler on the basis of the existing Plaid compiler is given. After taking a closer look at the involved runtime libraries, the implementation of the ideas presented in chapter 3 is described.

In chapter 5, a first evaluation of the prototype implementation is shown. An in-depth analysis of the code that is generated for a small example program is given as well as initial performance numbers. The results are then discussed and possible problems are highlighted.

In chapter 6, the results of the previous chapters are summarized. Following the summary, areas that have to be addressed in the future are presented.
2 Background

2.1 Plaid

Plaid [PG10b, PG10a] is a new general purpose programming language developed at Carnegie Mellon University whose characteristics are designed to facilitate the development of component-based and concurrent software. The key features of Plaid include

- typestate;
- access permissions;
- concurrency by default and
- gradual typing.

Each of these concepts will be discussed in more detail in the following sections.

2.1.1 Typestate

The motivation behind making typestate [SY86] a central language concept is the observation that state is fundamental to modeling the world. Engineers think in states and state transitions and use state machines to visualize and reason about object behavior, but conventional programming languages provide little to no support for expressing state machines in actual code. Often the only way to implement a state machine is to encode the current state explicitly using some form of an integer field inside the object or implicitly using information about whether certain fields are set to valid values or not\(^1\). State transitions and the checking for the correct state must be handled manually which is error-prone and difficult to update in the case new states or operations are added.

Giving the programmer direct means of expressing stateful designs in a programming language is becoming more and more important as software development activity is shifting from writing entirely new code to reusing previously developed software components. Such components almost always define usage protocols which must be followed to guarantee the correct function of objects. A usage protocol is a valid sequence of method calls. It can, for example, state that certain method calls are only allowed if the object is in a certain state and another method must be called to transition the object to that specific state. A file class, for example, only allows calls to \texttt{open()} if the object is

\(^{1}\text{Nullable fields are often used for this purpose.}\)
in state \texttt{Closed}, and an iterator class only allows calls to \texttt{next()} if there are non-visited elements left in the collection. Another concrete example is Java’s \texttt{BufferedReader} class, which allows the user to mark the current position in the stream, using \texttt{mark()}, and then later reset the stream to the marked position using \texttt{reset()}. Figure 2.1 shows a state machine for this portion of BufferedReader’s behavior.

![Figure 2.1: Abstract state machine for BufferedReader.](image)

An error will occur should the programmer try to reset the stream before it has been marked. Consider the following Java code:

```java
import java.io.*;
public class MarkReset {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new FileReader("test.file"));
            reader.reset();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

**Listing 2.1:** Simple protocol violation.

In this example, the intent of the \texttt{reset()} call was to reset the stream to the most recent mark—however, \texttt{mark()} is not called before which means that the reset call violates the usage protocol. Upon execution of the program, an \texttt{IOException} carrying the message “Stream not marked” is thrown.

In general, if dynamic ways of managing the state of objects like roles or the state design pattern are used, usage protocol compliance cannot be checked at compile-time but only at runtime. In this case, protocol violations typically result in an exception being thrown, just like in the example in listing 2.1. By contrast, an object’s typestate is statically trackable. This enables the compiler to assist the programmer in finding bugs related to the violation of usage protocols at compile-time. Using Plaid’s typestate mechanism [ASSS09], the BufferedReader usage protocol can be modeled like this:
In this Plaid snippet, two states called MarkedReader and UnmarkedReader are defined which represent the two states in figure 2.1. States look a lot like classes in conventional object-oriented programming languages and just like them can contain method and field definitions. Plaid uses the keyword var to declare a mutable variable that can be reassigned later whereas val indicates that the variable cannot be reassigned. Method definitions always start with the keyword method. Note that even functions in the global namespace are thought of as being part of a global object and are therefore also called methods and defined with the method keyword. In the following, “global method” and “global function” will be used interchangeably.

The bracket notation after the method definitions expresses the state transition and as such the pre and post conditions of the methods. For example, a call to mark() in state UnmarkedReader transitions the reader from unmarked to marked. In the case of methods, the first specified state transition always refers to the receiver object this. The state transition for mark() could also have been written as [UnmarkedReader >> MarkedReader this].

Note that the availability of fields and methods is linked to the state which is a major difference between working with typestate and using a conventional approach. For example, the reset() method is only available for objects in state MarkedReader. While typestate makes it possible for objects to change their state, this does not mean that their object identity itself changes. If the programmer tries to call reset() on an object whose state is still UnmarkedReader, this is a compile time error because as an object’s typestate is tracked statically, the compiler knows that the UnmarkedReader state does not define a reset() method. Similarly, the markPosition field only exists in the MarkedReader state and trying to access it in the other state is a compile time error as well.

In a conventional programming language like Java, this explicit modeling of states would not have been possible. Here, the programmer could have modeled the state implicitly by using the markPosition field as an indicator which state the object currently is in. In that case, some special value like −1 indicates the state UnmarkedReader and a value ≥ 0 indicates that mark() has been called. However, the disadvantages of this are obvious, as markPosition can be accessed and possibly changed even if the object is in state UnmarkedReader. The same reasoning applies to the methods and their specified state transitions. Without the typestate mechanism, the programmer needs to add checks for protocol compliance herself, i.e. check for markPosition ≠ −1 at the start of reset(). As mentioned before, all these checks are performed at runtime as opposed to the static checking which is possible with typestate support.
2.1.2 Access Permissions

While statically tracking the state of objects provides a lot of additional information at compile time, it is extremely difficult to do without further measures. The main problem is aliasing, i.e. the existence of multiple references to the same object. In the case of typestate tracking, aliasing makes it extremely hard to guarantee a certain state for the referenced object.

```
1 method void foo(A x) {
2     bar();
3 }
```

Listing 2.3: Aliasing problem.

Consider listing 2.3 where state A for x cannot be guaranteed any more after the call to bar finished because an alias to x could have been stored inside a global variable. The function bar could have used that global reference to alter the state of the object that is also referenced by x. One way out of this dilemma is to restrict aliasing which of course also restricts the developer and limits their flexibility while writing programs.

The idea of access permissions [BA07] is to combine access control and aliasing information into one concept. A permission is able to track how the current reference is allowed to access the referenced object and also contains information about how the object might be accessed by other references. Working with access permissions requires each reference to be associated with such an access permission. For sake of brevity, “unique reference” is used as an equivalent to “unique permission to a reference”. The different permissions will be discussed in more detail in the following paragraphs. In the figures, solid lines represent references that grant read/write access and dashed lines represent references that only grant non-modifying access to the object.

Unique

A unique reference [Boy03] to an object guarantees that at this moment in time, this is the only reference to that object. Therefore, the owner has exclusive access to this particular object. Figure 2.2 illustrates the situation. In the case of unique, there is only one reference which grants full read/write access.

![Figure 2.2: Unique access permission.](image-url)
Immutable

An immutable reference [Boy03] to an object guarantees that at this moment in time, no reference with read/write permission to the object exists. This means that the user’s reference itself does not grant modifying access. Note that, like shown in figure 2.3, immutable does not restrict the number of immutable references to an object that can exist at the same time.

\[\text{me} \rightarrow \text{Object} \rightarrow \text{them}_1 \rightarrow \ldots \rightarrow \text{them}_n\]

Figure 2.3: Immutable access permission.

Shared

A shared reference [DF04] to an object grants modifying access to the object while making no further claims about other possible references to the object. It is thus possible that other references with read/write permission, like other shared references, to the same object exist. Like figure 2.4 shows, shared allows an arbitrary number of other references with or without write access to the object.

\[\text{me} \rightarrow \text{Object} \rightarrow \text{them}_1 \rightarrow \ldots \rightarrow \text{them}_n\]

Figure 2.4: Shared access permission.

Full

A full reference [Bie06] to an object provides modifying access to the object, but in contrast to shared, it guarantees that no other references exist to that object which grant read/write access. As figure 2.5 illustrates, full does not restrict the number of other references but requires all of them to be read-only.

\[\text{me} \rightarrow \text{Object} \rightarrow \text{them}_1 \rightarrow \ldots \rightarrow \text{them}_n\]

Figure 2.5: Full access permission.
Pure

A *pure* reference [Bie06] to an object provides read-only access to the object and, as shown in figure 2.6, does not restrict the number and quality of other references. It is thus an access permission with very weak guarantees.

![Diagram of a pure access permission](image)

**Figure 2.6:** Pure access permission.

None

A *none* reference to an object provides neither read nor write access to the object. This may seem useless on the first glance but as the reference still points to a location in memory, having *none* references can be useful. As a *none* permission still points to a specific object, it can for example be used to get a particular object out of collection of *unique* objects like demonstrated in [ASSS09].

Obviously, not all of the access permission types are compatible with each other. An *immutable* reference to an object for example can never coexist with a reference that allows write access. Table 2.1 sums up the properties of the permission types again and additionally gives an overview of the compatibility amongst access permissions.

<table>
<thead>
<tr>
<th>Permission kind</th>
<th>This reference</th>
<th>Other references</th>
<th>Compatible permission types</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique</td>
<td>read/write</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>full</td>
<td>read/write</td>
<td>read-only</td>
<td>pure, none</td>
</tr>
<tr>
<td>shared</td>
<td>read/write</td>
<td>read-only</td>
<td>shared, pure, none</td>
</tr>
<tr>
<td>immutable</td>
<td>read-only</td>
<td>read-only</td>
<td>immutable, pure, none</td>
</tr>
<tr>
<td>pure</td>
<td>read-only</td>
<td>read/write</td>
<td>full, shared, immutable, pure, none</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>read/write</td>
<td>unique, full, shared, immutable, pure, none</td>
</tr>
</tbody>
</table>

**Table 2.1:** Access permission taxonomy.

One way to think about access permissions is to consider them resources that can be *consumed* and *produced*. Therefore, access permission lend themselves very well to being modeled with *linear logic* [Gir87]. Linear logic can be used to reason about resources within the logic itself. For example, in linear logic the usual implication $A \Rightarrow B$ is replaced by linear implication $A \rightarrow B$ which consumes its input $A$ and produces the output $B$. Thus the input is not available any more after it has been transformed. This is an important difference
between linear logic and classical logic. In classical logic, it is possible to conclude \( A \land B \) from \( A \) and \( A \Rightarrow B \) whereas linear logic only allows to conclude \( B \) from \( A \) and \( A \rightarrow B \) because \( A \) is consumed in the process.

Just as in linear logic where, once a resource has been consumed, it is no longer available, access permissions are consumed upon using them. Otherwise permissions could be freely duplicated and the guarantees regarding other references to the same object would not hold. For example, duplicating a \texttt{unique} permission immediately violates the uniqueness guarantee. Thus, a \texttt{unique} permission to an object is consumed as soon as the referenced object is accessed and a new access permission needs to be produced by the operation. If no new permission were produced, access to the object would be lost. This is also reflected by the way method signatures in Plaid are defined. Because access permissions play a central role in Plaid, they are integrated in the type system. Hence, the type of an object reference in Plaid is always a tuple and consists of a permission and the actual object type of the referenced object.

```plaintext
1 method void modify(unique Object >> unique Object x);
2 method void main() {
3 val unique Object o = new Object;
4 modify(o);
5 modify(o);
6 }
```

\textbf{Listing 2.4:} Method signature in Plaid showing the resource-like nature of access permissions.

In listing 2.4, a method \texttt{modify} is defined which takes a \texttt{unique} reference to its argument and gives it back after the method body has been executed. This is expressed by the pre-condition \( \gg \) post-condition syntax which specifies the pre-conditions and the post-conditions of the method. In the body of the \texttt{main} method, a local variable \texttt{o} of type \texttt{unique Object} is defined and then initialized with a newly constructed object. The assignment is valid because at this point in time, the reference returned by the \texttt{new} expression clearly is the only reference to the new object. After the initialization, the previously defined \texttt{modify} method is called. Following the resource interpretation of access permissions, the call consumes the \texttt{unique} permission to \texttt{o} and hands it to the called method. After \texttt{modify} returns, a \texttt{unique} permission to \texttt{o} is produced as specified by \texttt{modify}’s method signature. Because the permission is recovered, it is possible to call \texttt{modify} again. By convention, the signature of a method that does not change its argument’s permission is often abbreviated by leaving out the part after the \( \gg \) sign. Hence, the \texttt{modify} method could also have been written method void modify(unique Object x).

Consider a different example, shown in listing 2.5.

```plaintext
1 method void read(immutable Object >> immutable Object x);
2 method void pushOnStack(immutable Object >> none Object x);
3 method void main() {
4 val unique Object o = new Object;
5 pushOnStack(o);
6 read(o);
7 }
```

\textbf{Listing 2.5:} Splitting example.
Here, two methods are defined. \texttt{read}, that does not modify its argument, takes an \texttt{immutable} permission and returns an \texttt{immutable} permission. And \texttt{pushOnStack} which saves the reference to its argument in a stack data structure.

Passing a \texttt{unique} permission to \texttt{o} to \texttt{pushOnStack}, as is done in \texttt{main}, is intuitively no problem because a \texttt{unique} permission is stronger than an \texttt{immutable} permission. A \texttt{unique} reference guarantees that no other references exist whatsoever, thereby automatically also satisfying \texttt{immutable}'s requirement that no other reference with modifying access exists. However, because capturing the reference by putting it in a data structure consumes the access permission, \texttt{pushOnStack} just returns a \texttt{none} permission as specified by the post condition in the method signature. Hence, the following call to \texttt{read} is illegal because \texttt{read} requires an \texttt{immutable} permission where only a \texttt{none} permission is available. Intuitively, this program should work because the call to \texttt{pushOnStack} does not require a \texttt{unique} permission to \texttt{o} at all. If it were somehow possible to convert the \texttt{unique} permission into two \texttt{immutable} permissions and use one of these for each method call, the program would be valid. But it is unclear how this conversion can be modeled while preserving the guarantees of the different types of access permissions.

The answer to this question is called \textit{permission splitting}. As demonstrated before, some permissions are intuitively stronger than others; for example \texttt{unique} is stronger than \texttt{immutable}. Instead of passing a \texttt{unique} permission as an \texttt{immutable} in listing 2.5, the \texttt{unique} permission is split into two \texttt{immutable} permissions. One of those permissions remains at the call site while the other permission is passed to the called method. It is important that splitting needs to preserve the guarantees that are associated with the permissions. For example, it is illegal to split a \texttt{unique} into two \texttt{unique} permissions or a \texttt{full} permission into a \texttt{full} and an \texttt{immutable}.

\begin{align*}
\texttt{unique}(x) & \Rightarrow \texttt{full}(x) / \texttt{pure}(x) \\
\texttt{unique}(x) & \Rightarrow \texttt{immutable}(x) / \texttt{immutable}(x) \\
\texttt{full}(x) & \Rightarrow \texttt{shared}(x) / \texttt{shared}(x) \\
\texttt{full}(x) & \Rightarrow \texttt{shared}(x) / \texttt{pure}(x) \\
\texttt{full}(x) & \Rightarrow \texttt{immutable}(x) / \texttt{immutable}(x) \\
\texttt{immutable}(x) & \Rightarrow \texttt{immutable}(x) / \texttt{pure}(x) \\
\texttt{immutable}(x) & \Rightarrow \texttt{immutable}(x) / \texttt{immutable}(x) \\
\texttt{shared}(x) & \Rightarrow \texttt{shared}(x) / \texttt{pure}(x) \\
\texttt{shared}(x) & \Rightarrow \texttt{shared}(x) / \texttt{shared}(x) \\
\texttt{pure}(x) & \Rightarrow \texttt{pure}(x) / \texttt{pure}(x) \\
\Pi(x) & \Rightarrow \Pi(x) / \texttt{none}(x)
\end{align*}

\textbf{Figure 2.7:} Legal permission split rules.

Figure 2.7 lists some legal permission split rules. The variable \(\Pi\) stands for an arbitrary permission type.
A good visualization of the access permissions in a program is a permission flow graph like the one shown in figure 2.8. The nodes of the flow graph are the operations in the program that consume and produce permissions; for example function calls. When a permission split happens, this is indicated by a special split node which is inserted into the permission flow. Incoming edges of a node represent the permissions that are consumed by the operation and outgoing edges represent the produced permissions. All edges are annotated with the permission type and the name of the reference that this permission is associated with.

If permissions are split during the execution of a program, this immediately raises the question of whether the original permission can somehow be recovered later. Listing 2.6 gives an example where permission restoration is necessary for the program to compile.

In this example, a new object is created. Subsequently, the object is passed to a method which performs a read-only operation on it followed by a call that modifies the object. Following the permission splitting mechanism, the unique permission is split into two immutable permissions before read is called. After the call finished, however, the permission system is confronted with the problem of having two immutable permissions where one unique permission is needed for the modify call. It is intuitively clear that combining both immutable permissions back into one unique permission would make sense in this case.

In the following, this combination operation will be called permission joining. But although joining might intuitively make sense here, it is not always allowed. The reason for this is that an immutable permission can be split into an arbitrary number of immutable permissions, as can be seen when looking...
at the split rules in figure 2.7. Hence, it is unsound to allow the reconstruction of a unique permission out of two immutable permissions because, without further measures, it is unknown how often those permissions have been split in the meantime. If three immutable references to same object are created via splitting and two of them are recombined into a unique reference, the resulting situation violates the permission guarantees.

One way to deal with this problem is fractional permissions [Boy03]. Fractional permissions are like regular access permissions but are additionally annotated with a fraction that keeps track of how often a certain permission has been split. By definition, a unique permission carries a full fraction represented by one (1). If a permission is split, its associated fraction is divided by two and distributed amongst the resulting permissions. To join two permissions, their fractions are added and used as the fractional value for the resulting permission. This allows the definition of combined splitting and joining rules which work as follows.

\[
\begin{align*}
\text{unique}(x, 1) & \iff \text{full}(x, 1/2) / \text{pure}(x, 1/2) \\
\text{unique}(x, 1) & \iff \text{immutable}(x, 1/2) / \text{immutable}(x, 1/2) \\
\text{full}(x, \beta) & \iff \text{immutable}(x, \beta/2) / \text{immutable}(x, \beta/2) \\
\text{immutable}(x, \beta) & \iff \text{immutable}(x, \beta/2) / \text{pure}(x, \beta/2) \\
\text{immutable}(x, \beta) & \iff \text{immutable}(x, \beta/2) / \text{immutable}(x, \beta/2) \\
\text{shared}(x, \beta) & \iff \text{shared}(x, \beta/2) / \text{pure}(x, \beta/2) \\
\text{shared}(x, \beta) & \iff \text{shared}(x, \beta/2) / \text{shared}(x, \beta/2) \\
\text{pure}(x, \beta) & \iff \text{pure}(x, \beta/2) / \text{pure}(x, \beta/2) \\
\Pi(x, \beta) & \iff \Pi(x, \beta) / \text{none}(x, 0)
\end{align*}
\]

**Figure 2.9:** Legal rules for fractional permissions combining splitting and joining.

Figure 2.9 shows legal rules that combine splitting and joining, as indicated by the double arrows. The variable \(\beta\) in a rule indicates that the rule applies to any fractional value. Through the introduction of fractions, it becomes possible to solve the problem demonstrated in listing 2.6. The permission flow graph illustrating the splitting and joining of permissions is shown in figure 2.10.

Note that this example requires that immutable Object >> immutable Object x implicitly means that exactly the same fraction is returned. This convention is called borrowing. With fractional permissions, this could be made clearer by extending the syntax to allow the expression of the fraction part of a reference in a method signature like immutable<k> Object >> immutable<k> Object x. The parameter k requires both immutable references to have exactly the same fractional part.

If borrowing is enforced for all methods or if there exists some way of marking a method parameter as borrowed, permission joining becomes possible even in a permission system without fractions. However, to preserve soundness, the system must differentiate between a borrowed and a non-borrowed permission.
2.1. Plaid

Figure 2.10: Permission flow graph demonstrating fractional permissions.

on a fundamental level. Not all operations are allowed using a reference that is annotated with a borrowed permission. For example, borrowed permissions must not be passed to methods that do not borrow the permission.

2.1.3 Gradual Typing

Gradual typing [Sie] is a type system that allows the programmer to mix dynamically and statically typed code. Such a type system is flexible enough to enable the programmer to remove type annotations from statically typed program parts and still get a valid program. There has been much discussion about whether static or dynamic type checking is the better choice for developing software. While the question can certainly not be answered in general, there are certain advantages to both approaches.

Static type checking catches certain types of bugs earlier, thereby greatly assisting the developer in avoiding finding bugs late in the development cycle. A typical situation is the application of a binary operation $\oplus$ to two operands whose types are incompatible. In this case, the difference between dynamic and static checking is that in the static case the program will not compile, whereas in the dynamic case a runtime error will be raised upon execution of the binary operation. If such a bug is located in a code path that is executed very rarely and a dynamic type system is used, the fault can linger in the code unnoticed by the programmer for a long time. In the worst case, the fault is not discovered during testing and the program fails in production use. Additionally, static type systems support the optimization phases of the compiler by giving it enough information about variables to exploit specialized functional or storage units which might be available on the current architecture.
On the other hand, dynamic type systems are generally regarded as more flexible when it comes to modifying the program to react to changed requirements. In a statically typed setting, the programmer has to change the program into a form which is accepted by the type checker first. Furthermore, dynamic type checking facilitates certain situations where variable types depend on runtime information, for example when assigning the return value of a user entry to a variable.

Keeping this in mind, it seems like a good idea to try to support both static and dynamic type checking and let the user decide what is most appropriate given the current situation. A *gradual type checker* can handle programs where parts have been annotated with types and other parts have not.

```plaintext
method void incT (immutable Integer x) {
    x + 1;
}
method void inc (dyn x) {
    x + 1;
}
state S {
}
method void main () {
    incT (new S);
    inc (new S);
}
```

Listing 2.7: Gradual typing in Plaid.

Listing 2.7 shows an example of gradually typed Plaid code. Both versions of the `inc` method contain the same code: they both apply the binary + operator to their argument `x` with the other operand being the constant 1. The first `inc` method, however, does not require its argument to be of a certain type as expressed by the keyword `dyn`\(^2\). The second method, `incT`, does specify a type annotation for its argument.

While type checking the main method, the type checker will report a type error for the call to `incT` but not for the call to `inc` although both contain the same code. This is because when checking the call to `incT`, the type checker can use the information from `incT`'s signature and knows that the argument has to be of type `immutable Integer`. Here, the argument has an object type that is not equal to `immutable Integer` and the type checker will therefore report an error. For `inc`, the type checker does not have static type information about `x`, so it will defer type checking to runtime.

### 2.1.4 Concurrency by Default

The concurrency by default paradigm [SMA09] enables the programmer to express dependencies between operations in the program using access permissions instead of forcing them to explicitly specify the execution order. This

\(^2\) Actually, even `dyn` can be omitted here. The type is automatically assumed to be `dyn` in this case.
dependency information can then be leveraged by the compiler to automatically extract concurrency from the program. As this approach is a project independent from Plaid, it is covered in more depth in the following sections.

2.2 Àeminium

The idea of the Àeminium project [SMA09, SAM10] is to exploit additional information available to the compiler in the form of access permissions to automatically parallelize code. Instead of using the sequential order in which code is written to implicitly express dependencies, permissions are used to make those dependencies explicit. This makes it possible to declare concurrent execution to be the default, to the extent permitted by the dependencies in the program. Thus unlike in Plaid, the motivation for introducing access permissions in the language is not to support the compiler in tracking the type-state of objects but to enable the programmer to express dependencies between operations.

Although Àeminium’s ideas represent the foundations of one of the key features of Plaid, it is an independent project from Plaid. The Àeminium approach can be applied to any programming language that builds access permissions into the language. As shown in figure 2.11, Àeminium’s general design includes two major components: the compiler and the runtime system. The compiler’s task is to analyze the permission flow inside the program and to compute the dependencies between different parts of the program. Those parts are then packed into tasks which each encapsulate a certain part of the functionality of the program. Together with their inter-task dependencies, the set of tasks forms a data flow graph which is handed to the runtime for execution. The runtime is responsible for executing the individual tasks with the maximum amount of concurrency allowed by the assigned task dependencies and the currently available hardware resources. Àeminium’s runtime system is discussed in further detail in section 2.3.

2.2.1 Making Implicit Dependencies Explicit

One of the main problems when dealing with concurrency is shared state. Shared state is state, i.e. a chunk of memory, that is shared between different code parts which run in parallel. For example, two or more threads could have access to the same variable. If at least one of the participating threads has the right to modify the variable contents, this opens the window for data
Data races are situations where multiple threads access and manipulate the same data concurrently, and the result of the execution depends on the order in which the accesses took place. In order to prevent data races, accesses need to be synchronized with special means like locks or transactional memory. Manually managing the synchronization is notoriously complicated and leads to a considerable portion of bugs in concurrent applications. As soon as the programmer fails to protect shared state against concurrent modifications, the program contains a bug that is potentially very hard to find and can cause data corruption or program crashes.

But shared state also causes a lot of trouble for compilers that try to parallelize code. In a world without state this parallelization is relatively easy: pure functional programming languages do not allow functions to have side effects. This means that two functions cannot interfere with each other as it is impossible for them to access shared state. Hence, the compiler is free to execute functions in parallel to the extent permitted by data dependencies in the program.

However, a lot of situations can be modeled more easily with a stateful design, so the compilers for most programming languages need to deal with state. The main problem why parallelization in the presence of state is hard, is because there exist implicit dependencies between code and state. Functions can change arbitrary state, for example global variables, without specifying any of this in their signature. If two functions are called in a piece of code and the compiler wants to execute them in parallel, it has to be sure that they do not access shared state in an unsynchronized manner; otherwise the program semantics would not be preserved by the parallelization. But because of this lack of information about possible side effects of functions, the compiler has to stick to the execution order defined by the order of calls.

In Æminium, access permissions are used to make those implicit dependencies explicit. It forces a function to specify all side effects, i.e. all of the state it accesses, by requiring it to have an access permission available to each piece of state that is accessed. If a function tries to access state which has not been specified in its signature, a compile time error is reported. Hence, Æminium builds a permission-based effect system. An effect system is a formal system that allows the specification of the computational effects of computer programs in general and of functions in particular.

### 2.2.2 Unique and Immutable

Viewed from a concurrency perspective, unique and immutable exhibit very interesting properties. For unique, it is not necessary to protect the referenced object against concurrent modifying access because as exactly one reference exists, there cannot be competing accesses to the object. In the case of immutable references, a similar reasoning applies. As this permission type guarantees that no reference with modifying access exists, a data race is impossible; so again, objects do not need to be protected in any way against concurrent accesses. This is an extremely valuable piece of information if the compiler tries to
perform automatic parallelization. Consider the sample application shown in listing 2.8.

```
1 method unique Histogram computeRedHistogram(immutable RGBImage m);
2 method unique Histogram computeGreenHistogram(immutable RGBImage m);
3 method unique Histogram computeBlueHistogram(immutable RGBImage m);
4 method void equalize(unique RGBImage m, immutable Histogram r,
                      immutable Histogram g, immutable Histogram b);
5
6 method unique RGBImage histogramEqualization(unique RGBImage m) {
7     val r = computeRedHistogram(m);
8     val g = computeGreenHistogram(m);
9     val b = computeBlueHistogram(m);
10    equalize(m, r, g, b);
11 }
```

**Listing 2.8:** Plaid/Æminium code for performing histogram equalization on a color image.

In this piece of code, a histogram equalization is performed on a color image. An image histogram is a representation of the brightness distribution in a grayscale digital image and basically records the number of pixels with a given brightness. So for an eight-bit grayscale image, the histogram can be represented by a table with $2^8 = 256$ entries where entry $e_i$ contains the number of pixels with brightness $i$. For an RGB color image, the histograms of the three color channels are often computed independently. This works because each color channel can be viewed as a grayscale image. Histogram equalization is an operation that tries to increase the contrast of an image by spreading out the intensity values that occur most frequent. As its name suggests, histogram equalization works based on the histogram of the image; in the case of a color image it uses the histograms of all three color channels.

As the method signatures in listing 2.8 indicate, computing a histogram does not modify an image, which is why the matching methods all require an immutable permission to the image. The actual contrast modification performed in the equalize method does change the image contents and thus requires a unique permission. The body of histogramEqualization is straightforward: the histograms for all three color channels are computed and then passed to the equalize method. The matching permission flow graph is shown in figure 2.12. Note, however, that it has been idealized in two ways for presentation reasons. Firstly, the call to computeBlueHistogram would normally also be preceded by a split node which has been omitted to reduce the size of the graph. Secondly, for the same reason, the permission splitting for $r$, $g$ and $b$ has been omitted. As can be deduced from the method signatures, all three reference have the type unique Histogram but are passed as immutable Histogram to equalize. So to be exact, there would be four additional split nodes and four additional join nodes in the graph.

If the permission flow graph in figure 2.12 is interpreted from a concurrency perspective, it becomes clear that it lets the compiler identify independent operations. In this case, the independent operations are the three calls to the compute methods. This is also apparent in the graph, as there are no edges that directly connect the nodes representing the calls. The reason for this is that because they all take an immutable permission and do not specify any
other side effects, the order of their execution is not important. Therefore, it is legal for the compiler to generate code that executes all three method calls \textit{in parallel}. For example, it could generate code that assigns each execution of a method call to a different thread. In terms of the \textsc{æminium} runtime, each method invocation could be packed into a task and handed off to the runtime. As each task is associated with a set of dependencies on other tasks, it becomes possible for the runtime to execute tasks concurrently while preserving the correct semantics of the program.

\subsection{Shared Permissions}

While working with \texttt{unique} and \texttt{immutable} works very well in the shown examples, just two permission types are not flexible enough in most cases. Take for example a non-circular doubly-linked list. The inner list nodes which carry the values that are saved in the list each contain two references, one for the previous and one for the next list node. This also means that each inner list node is referenced by exactly two other list nodes. Hence, those references cannot be \texttt{unique} references because they point to the same object. But if they are \texttt{immutable}, all references to the list node have to be read-only and it becomes impossible to update the value stored inside that specific node. \textsc{æminium} uses \texttt{shared} permissions to allow the modeling of \textit{shared state}. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{histogramEqualization.png}
\caption{Permission flow inside the \texttt{histogramEqualization} method.}
\end{figure}
As discussed in section 2.1.2, shared permissions allow the existence of multiple references with read/write permission at the same time. Therefore, objects referenced through a shared reference must be protected against data races. For this purpose, Æminium introduces an atomic block statement.

```java
method void modify(shared Object >> shared Object o) {
    atomic {
        // Do something with o
    }
}
method void main() {
    val unique Object o = new Object;
    modify(o);
    modify(o);
}
```

Listing 2.9: Example use of the atomic block statement.

Listing 2.9 shows an example usage of atomic. Here, a new object is created and then used as an argument for calling the modify method twice. In contrast to prior examples, the method now requires a shared permission to its argument. As a unique permission can be split into two shared permissions, it is possible to feed one shared permission to each method call and thus execute them in parallel. The parallelism is safe in this example because access to the shared object is protected by atomic.

Note that the way the atomic statement is actually implemented under the covers is important for the semantics of the program. Æminium suggests using transactional memory semantics for atomic. However, in an actual implementation of the Æminium system, the designer could, for simplicity reasons, choose to first implement it using regular locks and then later switch to a transactional memory-based implementation. While both alternatives provide adequate protection against data races, they differ in various ways. Transactional memory semantics makes it difficult to allow the full spectrum of operations inside the atomic block. For example, I/O operations must either be forbidden inside atomic blocks or be handled as a special case which introduces additional complexity into the system. On the other hand, lock-based implementations are far more likely to lead to a deadlock.

### 2.2.4 Data Groups

Often, multiple objects are tightly connected and together form a more complex object, for example a data structure. In this case, managing the access to individual objects using atomic blocks is not necessarily safe. Take the linked list example again. Suppose the linked list contains ten elements of the form shown in listing 2.10.

```java
state ListElement {
    var shared ListElement next;
    var shared ListElement prev;
    var unique Value value;
}
```

Listing 2.10: State representing an element of a doubly-linked list.
Further suppose that the elements in the list shall be sorted. The sort operation relies on the values that are saved in the ListElement objects. Thus, those objects must not be modified while the sorting process is still in progress. As the list elements are all referenced through shared references, this means that the programmer has to synchronize separately on each list element object. This is tedious and possibly unsafe because it cannot be guaranteed that the whole sort() operation is atomic even if every access to a shared object is protected by an atomic block.

Therefore, Æminium introduces the notion of data groups [Lei98]. A data group represents a set of objects. In the original work, data groups are used to partition the state of one object. For example, for an object that represents a circle that is drawn on the screen, the state could be partitioned into a data group position containing the coordinates $x$ and $y$ and a second data group properties containing the color of the circle.

In Æminium, the concept of data groups is generalized. A data group now provides an abstract grouping for multiple objects that are somehow related but these objects do not necessarily need to form the state of one object. Each object that is referenced through a shared reference must be part of exactly one data group which is called the owner or owning data group of that particular object. To make this relationship also apparent in the syntax, shared<G> is written to express that the referenced object is part of the data group $G$. This also means that all shared references to an object must be associated with the same data group. The owner group therefore functions as a container for all shared references to an object. Applying the idea of data groups to the linked list example is straightforward, as the programmer can now put all objects that the list’s state consists of, i.e. all its list elements, in one data group and then synchronize access to the whole group conveniently by referring to the data group.

Data groups also provide a natural way of partitioning the heap. Because objects cannot be contained in more than one data group, two distinct data groups always contain disjoint sets of elements. This property becomes very important when two operations are executed on two data groups and it needs to be determined if it is safe to execute both operations in parallel. Because of the disjointness of distinct data groups, concurrent execution is allowed unless there exist other dependencies, for example induced by accessing a unique reference, that prevent parallel execution.

In order to address the problem of controlling access to objects contained in a data group as demonstrated by the sorting example, the concept of access permissions is applied to data groups. Quite similar to access permissions which provide aliasing information and access control for single objects, data group permissions provide the same for data groups. The three data group permission types exclusive, shared and protected will be explained in more detail in the following paragraph.

**Exclusive:** An exclusive data group permission resembles a unique access permission in that there exists at most one exclusive permission to a data group at the same time. If an exclusive data group permission exists, it
is therefore the only way to access the data inside the data group. Seen from a concurrency standpoint this means that unsynchronized access to all objects in the data group is safe.

**Shared:** A *shared* group permission is defined analogously to a *shared* access permission and allows an arbitrary number of other *shared* group permissions to that data group to exist at the same time. Because of this weaker guarantee, it is not safe to access any object inside the data group without proper synchronization. Therefore having a *shared* permission does not let the user access any object inside the group.

**Protected:** A *protected* group permission can be created by protecting access to a shared data group using the atomic block statement. The runtime system ensures that only one *protected* permission to a data group exists at a time in the whole system.

In contrast to access permissions, group permissions are not automatically split and joined. To split an *exclusive* permission into an arbitrary number of *shared* group permissions, the *share* block construct can be used. *Share* supplies each statement in the block with its own *shared* permission to the group. The statements in the block can also depend on other regular access permissions and the usual splitting and joining rules apply. However, if multiple statements require a *unique* permission to the same object, this is regarded as a static error. Permissions that are available but are not mentioned in the share block are left untouched. At the end of the *share* block, the *shared* group permissions are recombined into the group permission that was present when the block was entered. As *shared* group permissions behave exactly like *shared* access permissions in terms of splitting, *share* can also be used to further split a *shared* group permission.

A *shared* group permission, however, does not grant access to the objects contained in the associated data group because unsynchronized access creates the possibility of data races. First, the *shared* permission must be transformed into a *protected* permission by using an *atomic* block statement. The *atomic* statement is extended to allow the programmer to refer to the specific data group they want to protect. Just like a *shared* permission can be treated like a *unique* permission inside an atomic block, a *protected* group permission is treated as an *exclusive* permission, thus providing the illusion of exclusive access to the data group. Upon reaching the end of the *atomic* block, the group permission is reverted to the state it was in when the block was entered.

```c
1 // exclusive permission to G
2 share (G) {
3 // shared permission to G
4 atomic (G) {
5 // protected permission to G
6 }
7 // shared permission to G
8 }
9 // exclusive permission to G
```

**Listing 2.11:** Different group permission states.

Manual splitting and joining enables the programmer to directly influence the order in which operations are executed. Thereby they are able to express
higher-level dependencies that are not represented by data dependencies in the program. Listing 2.12 illustrates such a situation when the non-existence of data groups is assumed.

```
1 method void register(shared Subject s, unique Observer >> none Observer o);
2 method void update(shared Subject s);
3
4 method void main() {
5   val unique Subject s = new Subject;
6   val unique Observer o1 = new Observer;
7   val unique Observer o2 = new Observer;
8   register(s, o1);
9   register(s, o2);
10  update(s);
11  update(s);
12 }
```

Listing 2.12: Concurrent observer without data groups.

In this example, a subject and two observers are created. Following the standard observer design pattern, the subject maintains a list of observer objects that are interested in state changes of the subject. In this case it keeps a list of unique references to observer objects. As soon as the subject changes, it notifies all registered observers. To allow the maximum amount of concurrency in this example, both register and update take shared references to the subject. This permits the addition of observers in parallel to sending of notifications to already registered observer objects.

However, all function calls just depend on the initialization of s; there are no data dependencies amongst the function calls. Therefore, because of nondeterminism, it is now possible that updates are sent before any observers are registered and thus the messages are lost. Hence, the programmer needs a way to express the high-level dependency that the calls to update should only happen after the calls to register have been completed. Assuming the subject s is part of the data group G, the share construct makes it possible to implement this, as shown in listing 2.13. Because an exclusive group permission to G is recovered between the two share blocks, this makes the second block dependent on the first one, thereby enforcing the desired execution order.

```
1 share (G) {
2   register(s, o1);
3   register(s, o2);
4 }
5 share (G) {
6   update(s);
7   update(s);
8 }
```

Listing 2.13: Concurrent observer with data groups.

Figure 2.13 sums up the different types of access and group permissions that exist in Æminium. Solid arrows represent access or data group permissions. The numbers on the solid arrows specify the multiplicity of the relationships, i.e. there can either be one unique permission to an object or an arbitrary number of shared permissions, as expressed by n, or an arbitrary number of immutable permissions. Dotted arrows represent the possible transitions between the access permissions or data group permissions. As mentioned before,
an access permissions can be converted to another type of access permission by splitting or joining. Group permissions are converted via the `share` and `atomic` constructs. The dotted arrows are annotated with the necessary action that induces the permission conversion. The `/share` syntax expresses the end of a `share` block.

In section 3.9, the further integration of Æminium into Plaid is described.

2.3 Æminium Runtime

The runtime system [SMA11] is the second stage of the Æminium pipeline. Its purpose is to decouple the compiler from the actual hardware resources. After the compiler has analyzed the permission flow inside the program and identified work packages, those packages get packed into tasks and are assigned dependencies to other tasks. This data flow graph serves as the input for the runtime system. The main function of the runtime is to map runnable tasks to available hardware resources. To identify runnable tasks, it needs to permanently observe the dependencies of waiting states and as soon as all dependencies of a task are fulfilled, that task needs to get scheduled for execution.

The design of the runtime is directly influenced by Cilk’s [BJK+95] work-stealing algorithm. In this model, each processor maintains a ready pool which contains tasks that are ready to run. As soon as a processor notices that its ready pool is empty, it becomes a thief and chooses another processor using some kind of strategy. The strategy used by the Cilk runtime is choosing the victim processor uniformly at random. Once a victim processor has been chosen, the thief tries to steal a task from the victim processor’s ready pool. If the victim’s ready pool is empty as well, a new victim is chosen.

The work pool of each processor can be viewed as a stack-like data structure where new tasks are inserted at the bottom and tasks are stolen from the top. Figure 2.14 shows a typical situation. Processors two and three are busy
working on a task while processor three’s ready pool is empty. Processor three chooses another processor at random, in this case it is processor one, and steals the topmost task from the stack. If no other steal operation happens in the meantime, processor one will work next on task two.

It is important to carefully select the task that is stolen from the victim. Ideally, to reduce communication overhead, a big task should be removed from the victim’s pool. A big task keeps the processor busy for a longer period of time thus avoiding the need to waste time for transferring tasks between processors. Always stealing the topmost task and inserting new tasks at the bottom has proven to be a strategy that works especially well for tree-based computations. Such computations, which arise for example for divide and conquer algorithms, have the tendency that tasks generated early in the process are more likely to contain a large amount of work. As those early tasks are on the top of the stack and are, as such, the first ones to be stolen, this strategy works well for minimizing the communication overhead while avoiding spending too much time on choosing which task to steal.

The runtime also serves as an abstraction layer which hides the information about where individual tasks actually get executed. For example, there is an ongoing effort to investigate using the computational resources present on heterogenous hardware platforms like typical CPU/GPU combinations.

### 2.4 Related Work

Deterministic Parallel Java [BAD+09] extends the Java programming language and adds parallel language constructs with deterministic-by-default semantics. In the DPJ project, regions are used to partition the heap. Methods in the program must specify effect summaries that state which regions are read and written by the method. DPJ leverages this effect information to statically ensure that there are no interfering memory accesses between concurrent tasks in the program. Thereby DPJ prevents data races and guarantees deterministic semantics, meaning that for a given input, every execution of the program results in identical externally visible output. Parallel sections in the code must
be explicitly marked by the programmer with \texttt{cobegin} or \texttt{foreach} statements. Recently [BHH+11], DPJ added support for controlled nondeterminism in the form of \texttt{cobegin\_nd} and \texttt{foreach\_nd} statements.

Regions in DPJ are similar to Æminium’s data groups. However, data groups do not partition the whole heap but only the \texttt{shared} data. While access permissions are a natural way of describing effects in an object-oriented setting, DPJ also provides sophisticated means of describing effects on entities like arrays. Describing effects on arrays, which are ubiquitous in imperative high performance computing settings, is an issue that is not addressed very well in Æminium yet. With the inclusion of controlled nondeterminism, DPJ has moved towards Æminium which supports controlled nondeterminism through its \texttt{shared} permissions and atomic blocks. The most important design difference, however, is that Æminium declares concurrency to be the default while parallelism has to be introduced explicitly in DPJ. Therefore, it is Æminium’s ambition to free the programmer from having to decide which parts of the program to parallelize and from tuning it to perform well on a particular platform. The programmer just describes the dependencies in the program using access permissions and data groups and the Æminium system takes care of granularity management, taking into account the current platform properties.

Fortress [ACH+07] declares some language constructs to be parallel by default. For example, operands to an operator may be evaluated in parallel. However, Fortress does not support an effect system to ensure that those operations can indeed be safely run concurrently; it is up to the programmer to make sure no additional synchronization is necessary. As demonstrated, Æminium does not restrict the parallel-by-default idea to particular constructs but applies it to the whole language while using an effect system to make sure the parallelization does not change program semantics.

The Cilk [BJK+95] language provides fork-join parallelism just like Æminium does. To introduce parallelism in Cilk, the programmer has to annotate method calls with a special \texttt{spawn} keyword which expresses that this method call can be safely executed concurrently with other executing code. The system also provides a form of barrier synchronization through its \texttt{sync} keyword which forces the current method to wait for all method calls it has spawned to finish. Furthermore, the Æminium runtime is modeled closely after the runtime system that is part of the Cilk project. However, unlike in Æminium, Cilk does neither statically ensure that the parallelization is safe regarding data races nor is it able to infer possible positions of \texttt{spawn} and \texttt{sync} in the program without user interaction.

In general, as Æminium’s goal is to make the beneficial properties of functional languages regarding automatic parallelization available in imperative settings, all functional languages can be regarded as related work. Also, an Æminium data flow graph resembles typical data flow architectures [Rum75]. However, Æminium does extend this approach by supporting shared state and in-place updates.
3 Integration

The goal of this chapter is to explore the design space of the integration of Æminium into the Plaid language. Plaid is a natural fit for Æminium because Plaid’s type system has been built from the ground up with Æminium’s central prerequisite, access permissions, in mind. Because Æminium uses the permission flow to automatically parallelize programs, the type checking pass of the Plaid compiler provides enough information to analyze the dependencies between parts of the program. This means that the dependency analysis has to be invoked after the type checking pass. After the analysis, the program needs to be split into tasks that are suitable for execution on the Æminium runtime. To preserve the program semantics, the dependencies determined in the prior analysis pass must be translated correctly into inter-task dependencies. Figure 3.1 illustrates the whole process.

Before the dependency computation is analyzed on the basis of a concrete Plaid type system, the general problem of determining dependencies between program parts in a permission-based system is discussed in the next section.

3.1 Dependency Analysis

The purpose of the dependency analysis is to compute the dependencies between parts of the program. Ideally, non-interfering parts are detected. Two parts $p_1$ and $p_2$ are called non-interfering if they do not require exclusive access to the same resource. In that case, it is safe to execute those parts in parallel.

To identify dependencies between parts, the permission flow graph can be used, as was demonstrated in section 2.2.2. However, following the permission flow inside the program has an issue that has not yet been discussed. When looking at access permissions, it is unclear when the type checker actually needs to recover permissions to objects. For splitting, this question is not very hard to answer, as splitting is performed “on demand”. As soon as the current permission that is associated with a reference does not match the permission that is required, the appropriate split rule is used to split the permission. If no adequate split rule exists, the program cannot be typed. However, in the
case of joining permissions, this question is not that easy to answer. Consider the example given in listing 3.1.

```java
1 method void read(immutable Object >> immutable Object x);
2 method void main() {
3   val unique Object o = new Object;
4   read(o);
5   read(o);
6 }
```

**Listing 3.1:** Illustrating lazy and eager joining for access permissions.

In this piece of code, the `unique` permission to `o` needs to be split before the first call to `read`. After the call returns, however, it is unclear if the `unique` permission to `o` needs to be recovered at this point. Because of the convention that `read` returns exactly the `immutable` permission that has been provided as an argument, it would be sound to join back to a `unique` permission. This is called *eager* permission joining because the original permission is recovered as soon as possible. However, when looking ahead, a second call to `read` is the next thing that follows. As this call needs an `immutable` permission, too, it would also be possible to keep the existing `immutable` permission around and use it as an input permission for the second call. This is called *lazy* permission joining because the original permission is only recovered when it is absolutely necessary. Figure 3.2 shows the permission flow graphs in both cases.

![Permission flow graphs](image)

**Figure 3.2:** Permission flow graphs for eager and lazy joining.
Note that the existence of a unique permission to o between the two function calls, as depicted in the left flow graph, should not be regarded as a “bottleneck” by the dependency analysis. While it would not be incorrect to make the join1 node depend on the first read node and make the second read node depend on split2, it would still be missing out on possible concurrency. Even if a unique permission to o exists in between, the two calls to read are independent and should be executed in parallel.

Thus, it is reasonable to try to design the dependency analysis in a way that does not rely on a specific type checker behavior. One approach to do that is to let the actual uses of the objects in the program drive the dependency analysis instead of trying to follow the permission flow dictated by the type checker. This idea will be discussed in more detail in the following paragraphs.

For now, it will be assumed that the program operates on a set of abstract objects. Note that the term “object” does not necessarily refer to objects in the sense of object-oriented programming but only to some abstract entity that represents state. Furthermore, it will be assumed that each operation either changes a certain object, in which case it will be called a write to the object, or the operation does not change the object, in which case it will be called a read.

The goal of the dependency analysis is now to identify conflicting or interfering operations. Obviously, if two operations access different state, i.e. in our model different abstract objects, they do not conflict. If two operations do access the same object, they conflict with each other if at least one of them is a write operation. If both operations just read the object, it is not considered a conflict. Table 3.1 illustrates this.

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>write</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

**Table 3.1: Access operation compatibility.**

In this model, a program P can be represented as an n-tuple of either reads R[o_i] or writes W[o_i] to a certain abstract object o_i. The dependency analysis has to determine the set of dependencies \( D = \{ p_i \rightarrow p_j \} \) where a dependency \( p_i \rightarrow p_j \) means that operation \( p_j \) has to be executed before operation \( p_i \). For example, if the program contains the sequence \( W[a]; R[a] \) of operations, the read operation obviously has to depend on the previous write operation, otherwise it would not read the updated value for the object a.

Algorithm 1 shows the algorithm that is used to determine the dependencies in further detail. Its input is a program that is represented, like sketched above, as a tuple \( (p_i) \) of operations. The algorithm’s output is the set of computed dependencies in the form \( p_i \rightarrow p_j \) between the operations. The program is traversed in evaluation order, i.e. starting at \( p_1 \) and ending at \( p_n \). While doing that, the analysis keeps track of which operation accesses which object and update the dependencies accordingly. This is done using the Writer and Readers data structures. Writer maps an object o to the last operation p in
Algorithm 1: Dependency analysis.

**input**: The program $P = \overline{p}$

**output**: The set of dependencies $D = \{ p_i \to p_j \}$

$D \leftarrow \emptyset$

foreach operation $p$ in $P$ do

if $p = R[o_i]$ then

$w \leftarrow \text{Writer}(o_i)$

$D \leftarrow D \cup \{ p \to w \}$

$\text{Readers}(o_i) \leftarrow \text{Readers}(o_i) \cup \{p\}$

else // $p = W[o_i]$

$rs \leftarrow \text{Readers}(o_i)$

if $rs \neq \emptyset$ then

foreach reader $r$ in $rs$ do

$D \leftarrow D \cup \{ p \to r \}$

else

$w \leftarrow \text{Writer}(o_i)$

$D \leftarrow D \cup \{ p \to w \}$

$\text{Readers}(o_i) \leftarrow \emptyset$

$\text{Writer}(o_i) \leftarrow p$

the program that performed a write on that object. $\text{Readers}$ maps an object $o$ to the set of operations $p$ in the program that performed a read on that object since the last write occurred.

The algorithm itself is now straightforward. If the current operation $p$ is reading object $o$, a dependency from $p$ to the operation $q$ that has written $o$ last is added to $D$. The operation $q$ can be found easily by looking it up in the $\text{Writer}$ map. Also, $p$ is added to the set of readers of $o$. If the current operation $p$ is writing to object $o$, the current set of readers for $o$ is examined. Unless this set is empty, a dependency from $p$ to each operation $q$ in the set of readers for $o$ is added to $D$. If the set of readers is empty, a dependency from $p$ to the last writer of $o$ is added. This is done by looking up the last writer in the $\text{Writer}$ map. Regardless of whether $\text{Readers}(o)$ was empty or not, $p$ is set as the current writer of $o$ and the reader set for $o$ is cleared.

Table 3.2 shows an example run of the algorithm with the matching readers and writer sets after the execution of each statement in the program. Note that while the table shows the new dependencies that are being added in the current step, the readers and writer columns contain the full mapping including everything that has been added up to this point in the program.

Figure 3.3 shows the matching dependency graph. The nodes in the graph represent the read and write operations and the edges represent the dependencies. An edge $(p, q)$ means that $p$ depends on $q$, so the operation $q$ has to be performed before operation $p$ is performed. In this case, operation $o_1$ and $o_4$ are independent and could therefore be run in parallel.
Table 3.2: Example 1.

Moreover, the dependency graph is not connected. The nodes $o_0, o_2, o_3, o_6$ are not connected to any of the nodes $o_1, o_4, o_5$ or $o_7$. This is because at the moment operations are not allowed to access multiple objects. Therefore the analysis always ends up with one dependency graph for each object because a node that connects both subgraphs is forbidden. However, in reality operations often access multiple objects. Function calls, for example, can take an arbitrary number of arguments that in the current system would correspond to an arbitrary number of abstract objects.

Hence, it makes sense to allow operations in the program to consist of multiple reads and writes to different objects. Thus, the definition of an operation $o$ in the program is changed to represent a set of reads and writes to objects. In other words, it is possible to assign a write set and a read set to each operation in the program. The write set contains all objects that the operation writes to and the read set contains all objects that the operation reads. The approach presented above can still be used in the case that operations are allowed to access multiple objects.

Table 3.3 shows an example where the operation $o_2$ consists of a write to object $b$ and a read to object $a$. The matching dependency graph can be found in figure 3.4. Note that the dependency graph includes unnecessary transitive dependencies that are not needed to derive a partial order for the operations in the program. In the example at hand, it is the dependency $3 \rightarrow 0$ that is
Table 3.3: Example 2.

already expressed through the dependencies 3 → 2 and 2 → 0. These transitive dependencies exist because the objects are treated separately by the analysis. To remove the transitive edges in the graph, transitive reduction [AGU72] can be used.

Figure 3.4: Dependency graph for example from table 3.3.

3.2 The Plaid Type System

To investigate how this approach to dependency analysis works for a real Plaid program, Plaid’s type system needs to be discussed first. The main reason for this is that the analysis is heavily dependent on type information and, more specifically, access permission information. Plaid’s type system can be seen as an example of how type checking works in general in a typestate-based type system that incorporates permissions on a type level. Additionally, a simplification regarding Æminium is made. The analysis will only deal with unique and immutable for now, ignoring shared permissions and thus data groups altogether. Later, it will be discussed how the approach presented in the next sections can be extended to support all of Æminium’s language features.

The restriction to unique and immutable is a valid way of simplifying the system because it preserves the key properties that are interesting from a concurrency standpoint. The system still exposes operations that interfere with each other because they demand exclusive access to a certain resource as well as operations that are safe to run in parallel. Since data group permissions are conceptually very similar to access permissions, this is also an indicator that the same reasoning can be applied if the full Æminium system is used. Moreover, the
3.2. The Plaid Type System

subset of Æminium that only uses unique and immutable can be examined without making modifications to Plaid’s type system. In contrast, shared and data groups require profound changes of the type system.

### 3.2.1 The Core Language Syntax

Before typechecking, the Plaid AST is converted to a simpler representation which includes fewer elements than the normal AST. This so-called core language serves as the input for the type checker. The syntax of the internal language includes a core object calculus with first-class functions as well as methods, state changes, and mutable fields. Its conventions are inspired by the ones used in Featherweight Java [IPW01]:

\[
x, y, \text{this} \in \text{IDENTIFIER\text{\textsc{\textsc{n}}}\text{\textsc{\textsc{\textsc{n}}}}} \\
m \in \text{METHOD\text{\textsc{\textsc{n}}}\text{\textsc{\textsc{\textsc{n}}}}} \\
f \in \text{FIELD\text{\textsc{\textsc{n}}}\text{\textsc{\textsc{\textsc{n}}}}} \\
N \in \text{STATE\text{\textsc{\textsc{n}}}\text{\textsc{\textsc{\textsc{n}}}}}
\]

The this keyword is a special identifier that is bound to the receiver object of a method call. Like in Featherweight Java, smallcaps (IDENTIFIER\text{\textsc{\textsc{n}}}\text{\textsc{\textsc{\textsc{n}}}}) indicates syntactic categories, italics (m, f) indicates metavariables and sans serif (this) indicates a particular element of a category.

It is differentiated between declarations and expressions.

\[
\text{Declarations } D ::= \text{ var } T \ x = e \mid \text{ val } T \ x = e \mid \text{ method } T_r \ m(T_{pi} \gg T_{po} \ x)\{T_{li} \gg T_{lo}, T_{ei} \gg T_{eo} \ y\} \{ \ e \}
\]

Declarations are allowed in the global scope and inside state declarations. The keywords var and val introduce mutable and immutable variable bindings. In the description of a method declaration, a lot of special notation is used which will be discussed briefly in the following text. First, as mentioned before in section 2.1.2, each method parameter is annotated with two type annotations \(T_{pi}\), the input type, and \(T_{po}\), the output type. The input type \(T_{pi}\) indicates the type of the argument at the beginning of the method call and the output type \(T_{po}\) represents the argument’s type at the end of the method call. This accounts for the fact that in a type system that supports typestate, it is possible that the types of identifiers change during the execution of a program. Changes can occur due to two reasons. First, the permission associated with a reference may change because of a permission split or a permission join. Second, the state of an object may change because a state change operation has been applied. In addition to the argument, the receiver this itself can also change its type. The input and output types of the receiver are represented by \(T_{li}\) and \(T_{lo}\).

Furthermore, the type of objects in the environment of a method can also change, as indicated by \(T_{ei}\) and \(T_{eo}\). The environment of a method lists all
permissions to objects that are needed in addition to permissions to method arguments and to the receiver object. For example, objects that are referenced through global variables can be accessed by the method. As every object access requires a suitable permission to the target object in the permission-based system, this permission must be specified somewhere in the method signature. Otherwise the necessary permission will not be available upon object access which results in a type error. Therefore, the environment is a list of identifier-type bindings that describes the needed permissions to objects that are accessible through variables defined in surrounding scopes.

Figure 3.5 sums up the notation conventions that are used throughout the type system discussion. The inclusion of subtyping creates the necessity for distinction between specified and observed types. For example, for a method definition to be well-typed, the observed return type has to be a subtype of the specified return type.

$$
\begin{align*}
T_p & \quad \text{explicit parameter types} \\
T_e & \quad \text{environment variable types} \\
T_r & \quad \text{receiver types} \\
T_i & \quad \text{specified input types} \\
T_o & \quad \text{specified output types} \\
T_s & \quad \text{observed starting types} \\
T_f & \quad \text{observed ending types} \\
T_r & \quad \text{specified return types} \\
T_{ra} & \quad \text{observed return type}
\end{align*}
$$

Figure 3.5: Type system notation conventions.

The body of each method and the top-level program itself is an expression $$e$$ defined as follows:

$$
\text{Expressions } \quad e \ ::= \ \text{literal} \mid x \mid x.f \mid x.f := y \\
\quad \mid \ \text{let } x : T = e \ \text{in } e \\
\quad \mid \ \text{new } N \mid x \leftarrow N \\
\quad \mid \ \text{fn} \ (T_{pi} \gg T_{po} x)[T_{ei} \gg T_{eo} y] \Rightarrow e \\
\quad \mid \ x.m(y) \mid x \ y \\
\quad \mid \ \text{match}(x) \ \{ N \Rightarrow e \}
$$

Literals are either string literals or integer literals or the unit literal. Identifiers $$x$$ are straightforward. The expressions $$x.f$$ and $$x.f := y$$ represent read and write operations on the field $$f$$. The field read $$x.f$$ evaluates to the current value of the $$f$$ field of $$x$$. The field write $$x.f := y$$ assigns $$y$$ to the $$f$$ field of $$x$$.

The let expression $$\text{let } x = e_1 \text{ in } e_2$$ binds the value of $$e_1$$ to the variable $$x$$. Note that of the basic expressions, only let bindings allow arbitrary expressions as subexpressions. All other expressions, for example the field read $$x.f$$, already expect their subexpressions to be bound to a variable and thus only allow identifiers as subexpressions. Hence, let bindings are used to sequence all operations. Programs that adhere to this rule are said to be in administrative normal form [FSDF93], or A-normal form for short. This restriction simplifies
the type system because the tracking of typestate relies on sequencing. By pushing the problem of rewriting the program in A-normal form to a pass prior to type checking, the type system itself does not have to deal with it.

The new expression creates a new heap object of the supplied nominal type \( N \). The state change operation \( x \leftarrow N \) is a special typestate-related operation that is not present in regular programming languages. It replaces the value of \( x \) with a newly created object of state \( N \) which may or may not be the same state that \( x \) currently has.

As Plaid encourages a functional programming style, it integrates first-class functions. The expression \( \text{fn} \ (T_{pi} \gg T_{po} \ x)[T_{ei} \gg T_{eo} \ y] \Rightarrow e \) evaluates to a function with \( e \) as its body. Just like with methods, such an anonymous function, also called “lambda” in the following, can access objects through references defined in enclosing scopes. The same environment mechanism that is used for methods is also used for lambdas.

The method call expression \( x.m(y) \) binds \( x \) to this and then executes the body of \( m \) using \( y \) as the method argument. The application expression \( x \ y \) is defined like in the lambda calculus. It applies \( x \), which must be a function, to its argument \( y \) and evaluates to the result of the function.

The match construct \( \text{match}(x) \{N \Rightarrow e\} \) supplies the internal language with a powerful conditional expression. It matches its argument \( x \) against all supplied state names \( N \) in the specified order and evaluates to the associated \( e \) as soon as \( x \) matches against an \( N \).

### 3.2.2 Plaid Types

Plaid types are either permtypes or function types or void. A permtype \( T = p \ N \) pairs an access permission \( p \) with the current state information about an object as represented by a nominal type \( N \). In the following, permtypes will just be called “types” and “nominal type” will be used when explicitly speaking of the second part of a permtype. In this version of the type system, only unique, immutable and none permissions are supported. Function types are written using the usual arrow notation \( T \rightarrow U \) meaning a function that takes a parameter of type \( T \) and returns a value of type \( U \).

\[
\text{Plaid Types} \quad T ::= \ p \ N \ | \ (T_{pi} \gg T_{po})[T_{ei} \gg T_{eo} \ y] \rightarrow T_r \ | \ \text{void}
\]

\[
\text{Permissions} \quad p ::= \ \text{unique} \ | \ \text{immutable} \ | \ \text{none}
\]

\[
\text{Nominal Types} \quad N ::= \ < \text{NAME} >
\]

\[
\text{Declaration Types} \quad DT ::= \ \text{var} \ T \ f \ | \ \text{val} \ T \ f \\
| \ \text{method} \ T_r \ m(T_{pi} \gg T_{po})[T_{ti} \gg T_{to} \ T_{ei} \gg T_{eo} \ y]
\]

As usual, Plaid’s type system relies on typing contexts. A typing context is a set of typing assumptions and is usually expressed as a list of identifier-type bindings.

\[
\text{Linear Context} \quad \Delta ::= \cdot \ | \ x : T, \Delta
\]
However, in contrast to regular typing contexts, the typing contexts in Plaid are not lexical but linear. As discussed before, both the permission part and the state part of a type can change in the course of execution of the method. To distinguish the linear context from a merely lexical context, $\Delta$ is used instead of the more common $\Gamma$.

The linearity of the context becomes apparent in the typing judgments. Regular judgments only involve one typing context that provides the necessary input. However, as mentioned before, static type information can change over the course of typing a program which needs to be reflected in the form of the typing judgments. For this reason, Plaid’s typing judgments include an incoming context and an outgoing context.

$$\Delta_1 \vdash e : T \vdash \Delta_2$$

In the example shown above, $\Delta_1$ is the incoming context and $\Delta_2$ is the outgoing context. This judgment means that given the typing assumptions $\Delta_1$, the type $T$ can be assigned to the expression $e$ while the typing assumptions $\Delta_2$ are produced. Thus, the type system threads the context through the program, updating the context on the way as more expressions are typed.

### 3.2.3 Permissions

In the restricted system that only supports the three permission types unique, immutable and none, the permission splitting rules are straightforward. Permission splitting $p_1 \Rightarrow p_2/p_3$ describes how given a permission $p_1$, the permission $p_2$ can be acquired while leaving $p_3$ as the maximum residual permission.

\[
\text{(unique)} \quad \text{unique} \Rightarrow \text{unique}/\text{none} \\
\text{(immutable)} \quad p \in \{ \text{unique, immutable} \} \\
\text{p} \Rightarrow \text{immutable}/\text{mutable} \\
\text{(none)} \quad p \Rightarrow \text{none}/p
\]

The (unique) rule allows to always split a none permission off a unique permission and keep the unique permission. Similarly, like shown by (none), it is always possible to give up the current permission $p$, keep a none permission and leave the original permission $p$ as the residual. Furthermore, if a unique or an immutable permission is available, it is possible to split it into two immutable permission, as described by (immutable).

Intuitively, a unique permission is stronger than an immutable permission. This subpermission relation can be described by using the split judgments.

\[
\text{(split)} \quad p_1 \Rightarrow p_2 \quad \frac{p_1 \Rightarrow p_2}{p_1 \prec p_2} \\
\text{(trans)} \quad p_1 \prec p_2 \quad p_2 \prec p_3 \quad \frac{p_1 \prec p_2}{p_1 \prec p_3}
\]
3.2. The Plaid Type System

$p \Rightarrow p'$ is written as a shorthand for $p \Rightarrow p'/p''$ if the residual permission $p''$ is not important. A permission $p_1$ is defined to be a subpermission of $p_2$, if $p_2$ can be acquired out of $p_1$ by permission splitting. Note that if $p_1 \prec p_2$, $p_1$ is the stronger permission and $p_2$ is the weaker one. The (trans) judgment expresses the transitivity of the subpermission relation.

3.2.4 Subtyping

As mentioned before, a Plaid type is either a permtype, a function type or void, so three separate subtyping rules or needed. The subtyping rule for void is the simplest one.

\[
(\prec:\text{void}) \quad \text{void} \prec:\text{ void}
\]

As usual, \text{void} is defined as the only subtype of \text{void} by rule (\prec:\text{void}).

\[
(\prec:\text{permtype}) \quad N_1 \prec:\text{ } N_2 \quad p \prec:\text{ } p \quad N_2
\]

Rule (\prec:\text{permtype}) defines subtyping for permtypes. Note that the subpermission rule is not used at all. The subtyping relationship is solely based on the subtyping for nominal types. If the available access permission is not equal to the required access permission, the appropriate split rule is used to acquire the needed permission. If no such split rule exists, the program cannot be typed.

\[
(\prec:\text{lambda}) \quad y \subseteq y' \quad T'_{pi} \prec:\text{ } T_{pi} \quad T'_{ei} \prec:\text{ } T_{ei} \quad T'_{co} \prec:\text{ } T_{co} \quad T'_{r} \prec:\text{ } T_{r}
\]

Finally, subtyping for function types must be defined. As usual, input types are handled contravariantly while output types are handled covariantly. Note that special care has to be taken of the lambda environments. Besides ensuring the correct subtyping relationship between the types in the lambda environments, also the lists of identifiers have to be checked. Assume $L$ and $L'$ are two lambdas with types $T_L$ and $T'_L$ and identifier lists $\overline{y}$ and $\overline{y'}$ respectively. For $T_L$ to be a subtype of $T'_L$, every identifier in $\overline{y}$ must also be present in $\overline{y'}$, i.e. $\overline{y} \subseteq \overline{y'}$ must hold. If $\overline{y}$ contains identifiers not present in $\overline{y'}$, it would be unsound to allow $T_L \prec:\text{ } T'_L$ because in this case, $L$ could access objects for which it has no permission.

However, $\overline{y} \subseteq \overline{y'}$ is only a necessary and not a sufficient condition. Consider the following situation. Let $T_1 = (T_{pi} \gg T_{po}): | \rightarrow U$ and $T_2 = (T_{pi} \gg$
40 3. Integration

\[ T_{po} \trianglerighteq T_{ei} \trianglerighteq T_{eo} \rightarrow U. \] Obviously, the necessary condition \( \overline{y} \subseteq \overline{y}' \) is fulfilled. But it is not correct to consider \( T_1 \) a subtype of \( T_2 \) if \( T_{ei} \) is not equal to \( T_{eo} \). When typing a function application \( x \ y \) where \( x \) is of type \( T_2 \), the linear context is updated according to \( T_2 \). Specifically, the typing assumption for \( z \) is changed from \( T_{ei} \) to \( T_{eo} \). However, if \( T_1 <: T_2 \) holds, \( x \) could also be of type \( T_1 \) in which case \( z \) does not change at all. Hence, \( T_{ei} = T_{eo} \) must hold for every identifier-type binding \( T_{ei} \trianglerighteq T_{eo} \) in \( \overline{y}' \setminus \overline{y} \). This condition is described by \( \overline{y} \subseteq \overline{y}' \).

3.3 Analyzing the Type-Annotated AST

In this subsection, the typing judgment for each type of expression in the internal language will be discussed from the perspective of determining the correct read and write sets for the dependency computation.

\[
\frac{p_1 \Rightarrow p_2/p_3}{\Delta, x : p_1 \vdash x : p_2 \vdash \Delta, x : p_3 \vdash} \quad (\text{var})
\]

When typechecking a variable reference, the needed permission is split off using the appropriate permission split rule and the residual is left in the context. Regarding read and write sets, this expression is considered to be a write if \( x \) is typed with a \texttt{unique} permission and a read otherwise.

\[
\Delta_i = \Delta, y : T_{es}, z : T_{ps} \quad \Delta_o = \Delta, y : T_{eo}, z : T_{po} \\
T_{es} <: T_{ei} \quad T_{ps} <: T_{pi}
\]

\[
\frac{\Delta_i, x : T_f \vdash x \ z : T_r \vdash \Delta_o, x : T_f}{\Delta_i, x : T_f \vdash x \ z : T_r \vdash} \quad (\text{application})
\]

To type check an application \( x \ z \), it needs to be ensured that the left hand side \( x \) has a function type \( T_f \). Next, it needs to be checked that both the parameter and all bound variables have types that are compatible with the types that are mentioned in the function type. This is expressed by the subtype relationships. The function type \( T_f \) of \( x \) is left untouched. The outgoing context \( \Delta_o \) reflects all the changes possibly made to the types of the argument and the types of the bound variables. Finally, the application itself is typed with the type specified as the return type in the type of \( x \).

In terms of read and write sets, the variable \( x \) holding the function is just read, so it is part of the read set. The classification of the function argument \( z \) depends on the types \( T_{pi} \) and \( T_{po} \). In the restricted setting where the only permissions are \texttt{unique} and \texttt{immutable}, the application is considered a write to \( z \) iff \( z \) is typed with a \texttt{unique} permission and a read otherwise. The same principle applies to the bound variables. If the function application requires a \texttt{unique} permission to a bound variable, it is added to the write set, otherwise it is added to the read set.
3.3. Analyzing the Type-Annotated AST

\[\Delta_i = \Delta, z : T_{ps}, x : T_{ls}, y : T_{es}\]
\[\text{(method call)}\]
\[\{\text{method } T_r \; m(T_{pi} \gg T_{po})[T_{ti} \gg T_{to}, T_{ei} \gg T_{eo} \; y]\} \in T_{ls}\]
\[T_{ps} <: T_{pi} \quad T_{ls} <: T_{ti} \quad T_{es} <: T_{ei}\]
\[\Delta_o = \Delta, z : T_{po}, x : T_{to}, y : T_{to}\]
\[\Delta_i \vdash \; x.m(z) : T_r \vdash \Delta_o\]

Type checking a method call \(x.m(z)\) works quite similarly to type checking an application. The method argument type and the bound variables in the environment are treated exactly the same. The method call additionally mentions a receiver \(x\) whose type needs to be considered as well. If \(x\) is typed with a unique permission, the method potentially modifies the object that \(x\) refers to. Therefore, the method call needs to be treated as a write to \(x\) in this case. If \(x\) is typed with an immutable permission, the method call is treated like a read to \(x\).

\[\Delta_i = \Delta, z : T_{ps}, y : T_{es} \quad x : T_s \in \Delta_i\]
\[\text{(function call)}\]
\[\text{var } T_f \; f \} \in T_{ls} \quad T_f = (T_{pi} \gg T_{po})[T_{ei} \gg T_{eo} \; y] \rightarrow T_r\]
\[T_{ps} <: T_{pi} \quad T_{es} <: T_{ei}\]
\[\Delta_o = \Delta, z : T_{po}, y : T_{to}\]
\[\Delta_i \vdash \; x.f(z) : T_r \vdash \Delta_o\]

Because the expression \(x.f(z)\) is ambiguous in Plaid and can mean both a method invocation and an invocation of a lambda stored in a field, a separate typing rule needs to be introduced. If it turns out during typechecking that \(f\) is actually a field defined in the type of \(x\) and not a method, this judgment is used to type the expression. In terms of read and write sets, this is treated exactly as one would expect after having seen the judgments for applications and method calls before. As \(x\) is just read, it is part of the read set. The classification of \(z\) once again depends on the type of the parameter that the lambda specifies. If it specified unique, \(z\) is part of the write set and otherwise it is added to the read set.

\[y : T_{ei}, x : T_{pi} \vdash \; e : T_{ra} \quad \neg y : T_{ef}, x : T_{pf}\]
\[T_{ef} <: T_{ra} \quad T_{pf} <: T_{ra}\]
\[\Delta \vdash \; \text{fn} \; (T_{pi} \gg T_{po} \; x)[T_{ei} \gg T_{es} \; y] \Rightarrow e : (T_{pi} \gg T_{po})[T_{ei} \gg T_{es} \; y] \rightarrow T_r \vdash \Delta\]

To type functions, it first needs to be checked that all variables that are mentioned in the lambda environment are actually present in the context. Subsequently, it is verified that the body expression \(e\) has a type \(T_{ra}\) that is a subtype of the specified return type \(T_r\). Similarly, after evaluating \(e\), the types of the variables in the environment types and the types of the parameters need to be compatible in a subtyping sense with the types that are specified in the type of the function.
Note that while the definition of a lambda captures variables, it does not capture any permissions. Therefore a lambda definition does not modify the linear context in any way. Permissions to variables are only passed in upon calling the lambda. Because of this, the dependency analysis always assigns an empty read set and an empty write set to a function expression. However, a new instance of the dependency analysis is run on the body of the lambda.

\[
\Delta, x : \text{unique} N \vdash x \leftarrow N' : \text{void} \vdash \Delta, x : \text{unique} N'
\]

The state change operation \( x \leftarrow S \) requires a writable permission to \( x \) which in the restricted setting automatically means that a \text{unique} permission to \( x \) is needed. Hence, a state change operation is always considered a write to \( x \).

\[
\{ \text{var } T_f \} \in T
\]

(field read, function)
\[
T_f = (T_p \gg T_p) \gg T_e \gg z \rightarrow T_r
\]
\[
\Delta, x : T \vdash x.f : T_f \vdash \Delta, x : T
\]

When typechecking a field read and a value of function type is stored inside the field, \( x.f \) is typed with the appropriate function type. This is considered a read to \( x \).

\[
\{ \text{var } p_f \} \in T
\]

(field read, weak)
\[
p = \text{readPermission}(p_x, p_f)
\]
\[
\Delta, x : p_x N_x \vdash x.f : p \vdash \Delta, x : p_x N_x
\]

When typechecking a field read and the value that is stored inside does not have a function type, the weak field read rule is used. Here, the maximum permission that can be provided without a change to the target object is split off. If the field \( f \) is declared as having a \text{unique} permission, it is only possible to split off a \text{none} permission, otherwise \text{unique}'s guarantees would not hold any more. If \( f \) is defined as \text{immutable}, it is safe to split off an \text{immutable} permission for \( x.f \). This behavior is wrapped in \text{readPermission} here. A weak field read is considered to be a read operation on \( x \).

The current system does not include a strong field read. A strong field read is a field read that requests a permission that is stronger than the maximum permission that can be provided without violating the permission invariants defined in the type of the target object. For example, requesting a \text{unique} permission to a field is not possible without giving up the permission to that field in the object. In a nominal type system, this can be realized by unpacking. Unpacking exposes the structural type and allows temporarily breaking the nominal type invariants. To be able to use the referenced object with its nominal type again, the object type must be packed back up. In the restricted setting, a strong field read \( x.f \) would always require a \text{unique} permission to \( x \) and is therefore considered a write to \( x \).

\[
T_x = p_x N_x \quad \{ \text{var } T_f \} \in T_x \quad T_f = p_f N_f
\]

(field assignment)
\[
p_x = \text{unique} \quad T_y < : T_f
\]
\[
\Delta, y : T_y, x : T_x \vdash x.f := y : \text{void} \vdash \Delta, x : T_x
\]
The field assignment is always typed as \texttt{void}. It is ensured that a unique permission is available to the target \texttt{x}; otherwise it is not allowed to change the referenced object. Furthermore, it is checked that the types of \texttt{f} and \texttt{y} are compatible. Note that the assignment consumes the permission to \texttt{y} because it is now stored in the field \texttt{f}. This expression is always considered to be a write to \texttt{x}. Depending on whether \texttt{y} has an immutable permission or a unique permission, it is considered to be a read or a write to \texttt{y} respectively.

Let allows the definition of a new variable binding in the scope of \texttt{e}. It is assumed that variables bound by let expressions can be renamed as needed. To type a let binding, the input context \(\Delta\) is used to type the bound expression \(e_1\) which yields the context \(\Delta_t\). After \(x\) has been added to the context \(\Delta_t\), the resulting context is used to type \(e_2\), yielding the context \(\Delta'\). The output context for the entire let expression is \(\Delta'\) minus the type assumption for \(x\). This preserves lexical scoping. The part \(x = e_1\) of the let binding is considered to be one entity and \(e_1\)'s read and write set is assigned to it. Additionally, \(x = e_1\) is always considered to be a write to \(x\).

\[
\begin{align*}
T_1 <: T & \quad \Delta \vdash e_1 : T_1 \vdash \Delta_t \quad \Delta_{t,x} : T \vdash e_2 : T_2 \vdash \Delta' \quad \Delta'' = \Delta' \setminus x \quad \Delta \vdash \text{let } x : T = e_1 \text{ in } e_2 : T_2 \vdash \Delta''
\end{align*}
\]

For match, each nominal type \(N\) that is matched against a variable of nominal type \(N_x\) must be declared a case of \(N_x\). In a nominal setting, this can be ensured by requiring \(N\) to be subtype of \(N_x\); i.e. \(N <: N_x\) must hold. This is valid because explicitly declaring \(N\) to be a case of \(N_x\) by using the “case of” keywords is the only way of establishing a subtype relationship in a nominal system. Note that in a structural type system, this is not correct.

As \texttt{match} is a conditional expression and it is not statically known which branch will be taken, the type checker needs to conservatively approximate the variable types as well as the type of the whole \texttt{match} expression in the outgoing context. Before typing an expression inside the \texttt{match}, the type information about \(x\) needs to be strengthened. Inside a particular \texttt{case} of the \texttt{match}, a smaller type for \(x\) than before can be assumed. Hence, before typing a case expression, \(x\) is inserted with type \(p \ N\) instead of \(p \ N_x\) into our incoming typing context \(\Delta\). After that, \(e\) can be typed which yields an outgoing context \(\Delta'\). This is done for each case that is declared in the \texttt{match}.

The resulting outgoing contexts \(\Delta''\), one for each \texttt{case}, must now be combined into one outgoing context \(\Delta'\) that is used as an incoming context to type the expressions following the \texttt{match}. Analogously, the types \(T_x\) for the case expressions need to be combined into one type \(T'\) that is assigned to the match expression as a whole. This type combining operation \(\Rightarrow\) is defined as an operator on two Plaid types which returns the resulting combined Plaid type. However, it can be generalized to operate on two linear contexts \(\Delta_1\) and \(\Delta_2\).
by applying the type combining operator to the types of each identifier that is present both in $\Delta_1$ and in $\Delta_2$.

Let $T_1 = p_1 N_1$ and $T_2 = p_2 N_2$ be two Plaid types. To determine their combined type $T = p N$, the permission part and the nominal type part need to be handled separately. To combine two permissions, the weaker permission of the two is kept.

\[
\frac{p_1 \preceq p_2}{p_1, p_2 \succ p_2} \quad \frac{p_2 \preceq p_1}{p_1, p_2 \succ p_1}
\]

The result of the combination of two nominal types is defined to be their least common supertype. The nominal type $N$ is called the least common supertype of $N_1$ and $N_2$ iff

- $N_1 \ll N$ and $N_2 \ll N$ and
- if there exists an $N'$ such that $N_1 \ll N'$ and $N_2 \ll N'$, then $N \ll N'$.

Let $lcs(N_1, N_2)$ denote the nominal type $N$ that satisfies these requirements.

\[
\frac{N = lcs(N_1, N_2)}{N_1, N_2 \gg N}
\]

Using the definitions of type combining for permissions and nominal types, it is possible to define type combining for full Plaid types.

\[
\frac{T_1 = p_1 N_1}{p_1, p_2 \gg p'_{1,2}} \quad \frac{T_2 = p_2 N_2}{T_1, T_2 \gg T'}
\]

The dependency analysis needs to take an approach that is similar to the one used by the type checker. It analyzes each case expression separately and constructs the read and write sets accordingly. In the next step, it needs to conservatively approximate the read and write sets for the whole `match` expression.

```
1 method void modify(unique Object x);
2 method void foo(unique Object x, unique Object y) {
3    match (x) {
4        case A {
5            modify(x);
6        }
7        case B {
8            modify(y);
9        }
10    };
11    modify(x);
12}
```

Listing 3.2: Approximating read/write sets for `match`.

Listing 3.2 illustrates why conservatism is important. In this example, each case expression accesses a different reference to potentially modify the referenced object. After the `match` expression, `modify` is called again on the object
3.4 Dependency Analysis Examples

In the following paragraphs, a few example Plaid programs will be shown and it will be discussed how they are typed and how the dependency analysis figures out the correct dependencies between the expressions. Consider the first example, shown in listing 3.3.

```plaintext
1 method void read(inmutable Object x);  
2 method void modify(unique Object x);  
3  
4 method void foo(unique Object x) {  
5    modify(x);  
6    read(x);  
7    read(x);  
8    modify(x);  
9 }
```

Listing 3.3: Dependency example 1.

Here, the permission flow is straightforward. After the first call to modify, the permission to x is split into two immutable permissions and each of the read calls can run independently. Subsequently, the immutable permissions are joined and the unique permission is recovered. Obviously, both read calls have to depend on the first modify call and the second call to modify has to depend on both read calls. To observe the approach of the dependency analysis, the AST as it is output by the type checker needs to be examined. Listing 3.4 shows the program in the internal syntax.

```plaintext
1 method void foo(unique Object x) {  
2   let t0 = apply(modify, x) in // R = { modify } W = { t0, x }  
3   let t1 = apply(read, x) in // R = { read, x } W = { t1 }  
4   let t2 = apply(read, x) in // R = { read, x } W = { t2 }  
5   apply(modify, x) // R = { modify } W = { x }  
6 }
```

Listing 3.4: Dependency example 1 in internal syntax.

As mentioned in the previous section, let bindings are used to sequence the individual expressions. The more explicit syntax apply(x, y) is used here...
for the function application $x y$. Note that global methods are modeled as lambdas because they do not have a receiver object. The individual lines have already been annotated with the respective read and write sets. As described before, calls to functions that take a unique permission are considered writes to the referenced object which is why $x$ is in the write set for both modify calls.

```plaintext
method void modify(unique Object x);
method void foo(unique Object x) {
  val f = fn () => {
    modify(x);
  };
  f();
  modify(x);
}
```

**Listing 3.5:** Dependency example 2.

The second example, shown in listing 3.5, illustrates the importance of the lambda environment. Here, the local variable $f$ holds a lambda that uses a variable $x$ that is defined in the surrounding scope. As mentioned before, the definition of the closure does not capture an access permission yet but the permission is only passed in when the function is actually called. Because $f$ requires a unique permission to $x$, the calls to $f$ and modify in lines 8 and 9 must be executed sequentially. In terms of dependencies, the call to modify has to depend on the call to $f$. The representation of the program in the internal syntax is shown in listing 3.6.

```plaintext
method void foo(unique Object x) {
  let f = fn () => {
    modify(x);
  } in
  // $R = \{ modify \}$
  // $W = \{ f \}$
  let t0 = apply(f, unit) in
  // $R = \{ f, modify \}$
  // $W = \{ t0, x \}$
  apply(modify, x)
  // $R = \{ modify \}$
  // $W = \{ x \}$
}
```

**Listing 3.6:** Dependency example 2 in internal syntax.

The key observation here is that the call to $f$ adds modify to the respective read set and $x$ to the matching write set. This information is gained from $f$’s type which is $\text{Unit}^\text{immutable} (\text{unique Object})[\cdot] \to \text{Unit} \text{modify, unique Object} \to \text{Unit}$. The important part is the associated lambda environment which lists all permissions that need to be passed in when $f$ is called. Note that the function type of modify is also associated with an access permission, immutable in this case, to unify the reasoning about object types and function types. Furthermore, the lambda environment specifies the need for a unique permission to $x$ which is needed to call the modify function inside the lambda body.

```plaintext
method immutable Integer fib(immutable Integer n) {
  match (n <= 2) {
    case True {
      1;
    }
    case False {
      fib(n - 1) + fib(n - 2);
    }
  }
}
```

**Listing 3.7:** Dependency example 3.
Finally, listing 3.7 shows an example involving the `match` construct. It is a recursive implementation of the Fibonacci function and thus represents a very simple instance of the broader class of divide and conquer algorithms. This example illustrates how the dependency analysis deals with different permission requirements in the branches of the `match`. The internal language representation is shown in listing 3.8.

```
method immutable Integer fib(immutable Integer n) {
    let t0 = methodCall(n, <, 2) in // R = {n} W = {t0}
    match(t0) {
        case True =>
            1 // R = Ø W = Ø
        case False =>
            let u0 = methodCall(n, −, 1) in // R = {n} W = {u0}
            let u1 = apply(fib, u0) in // R = {fib, u0} W = {u1}
            let u2 = methodCall(n, −, 2) in // R = {n} W = {u2}
            let u3 = apply(fib, u2) in // R = {fib, u2} W = {u3}
            methodCall(u1, +, u3) // R = {u1, u3} W = Ø
    }
}
```

Listing 3.8: Dependency example 3 in internal syntax.

Note that every binary operation, like basic arithmetics or comparisons, on an integer value is actually a method call in Plaid. Similar to function application, the syntax for method calls has also been made more explicit and `methodCall(x, m, y)` is written instead of `x.m(y)`. The declaration type `method immutable Integer + (immutable Integer ⇒ immutable Integer)[immutable Integer ⇒ immutable Integer]` is assumed for the add operation on integers; the declaration type for substraction looks analogously. Similarly, the declaration types for comparison operations like “<=” only differ in their return types which is `immutable Boolean`. Note that in all cases the environment is empty except for the needed permission to the receiver object `this`.

To determine the read and write set for the whole `match` expression, the expressions inside the `cases` need to be analyzed first. The first case expression is very simple because it just consists of the integer literal 1. It would be possible to view every integer as a global object of type `immutable Integer` and, following this idea, view each use of an integer literal as a `read` of this particular object. However, as it is impossible to acquire a permission that grants modifying access to such an integer object, there cannot exist interfering accesses to those objects. Therefore, it is valid to assign empty sets to the read and write set of the integer literal expression.

The second case expression is more interesting. Note that because of the administrative normal form, subexpressions like `n - 1` must be bound to an extra variable before they can be used in other expressions. Both method calls are considered to be reads to `n` because the substraction method only requires an `immutable` permission to the receiver object and hence the type checker has typed `n` as `immutable Integer`.

The read and write sets for the whole match expression are now determined using the conservative approach described in the previous section. As the variables `ui` are local to the case block, they have been omitted in the sets for
3. Integration

the enclosing match expression. However, it would not create any problems to also add them to the overall sets as long as the identifiers are unique in the whole method.

### 3.5 Address Permissions

Like demonstrated before, access permissions make it possible to mediate access to objects. Simultaneously, they allow the programmer to specify the side effects of functions. However, access permissions alone are not able to capture every form of effect that exists in Plaid. Consider the Plaid code in listing 3.9.

```plaintext
1 var unique Object g;
2 method void foo () {
3     g = new Object;
4 }
```

**Listing 3.9:** Changing of a mutable variable.

Here, the method `foo` changes the (global) mutable variable `g` by assigning a newly created object. Yet it would be incorrect to specify the need for a `unique` permission to `g` in the environment of `foo`. This is because `foo` does not actually change the *object* that `g` points to at all. It is the variable—or the pointer—`g` itself that is being modified. This situation can also arise in a context that does not involve global state as shown in listing 3.10.

```plaintext
1 method void foo () {
2     var immutable Integer x = 0;
3
4     val f = fn () => { x = 1; };
5     val g = fn () => { x = 2; };
6
7     f();
8     g();
9 }
```

**Listing 3.10:** Changing of a mutable local variable.

In this example, two local lambdas are defined that both change a mutable variable in their surrounding scope. As seen before, passing a `unique` permission to `x` does not capture the semantics and furthermore would be impossible any way because there is only an `immutable` permission available. However, as Æminium relies on type information to parallelize code, every effect needs to be specified in the type.

One way to solve this problem are *address permissions*. Address permissions essentially view variables as “objects”, too, and introduce means of managing and controlling access to mutable variables in a program. Thus, address permissions share the same idea as access permission but control access to the variables themselves and not to the objects the variables refer to. A simple system could for example differentiate between reads and writes to variables. Let `&x` denote the address of a mutable variable, alluding to the address operator `&` from C. A specification like `unique &x` then means that the method or function needs a `unique` permission to the the variable `x`. Viewed from the
standpoint of side effects, it also means that the function potentially changes x.

This approach can be used to specify the side effects of the lambdas in listing 3.10. The type of both lambdas becomes Unit [unique &x] → Unit and reflects the fact that they change x. Thus it becomes possible for the dependency analysis to deduce that it is not safe to execute f and g in parallel. If both lambdas only read x, they just need an immutable address permission to x. Address permissions are managed by the type checker and can be split and joined just like regular access permissions.

However, handling address permissions is actually easier than handling access permissions because address permissions cannot be consumed inside methods. If a method takes an immutable permission to an object for example there is the possibility that it splits the permission in its body and does return exactly the permission that was available at the beginning of the method. This issue created the need for fractions and borrowing in the first place. With address permissions this is not a problem because it simple not possible to “store away” part of an address permission. Additionally, it is relatively easy to infer the needed access permissions of a function because reads and writes to variables can be identified quite easily.

Another possible way of addressing the problems is to map address permissions to access permissions. This could be achieved through a rewrite rule like

```
var p T x;  val y = new { var p T x; };.
```

This rewrite rule places the mutable variable inside an object. Note that the object has a structural type here. If a nominal type system is used instead of a structural type system, a matching nominal type needs to be defined accordingly. Because the variable is located inside an object now, changing the variable is equivalent to modifying the surrounding object. Thus, regular access permissions are able to mediate accesses to the object and thereby to the variable. Of course, a second rewrite rule has to convert each usage of x by a dereference y.x. Note that y needs to be defined as an immutable binding (val) to prevent recreating the very problem that needed to be solved initially.

### 3.6 Typing of Lambdas

Because control structures in Plaid are not built into the language but are defined in terms of more fundamental language concepts, the typing of lambdas plays a central role. Listing 3.11 shows the definition of ifElse.
The general idea of `ifElse` is that it takes three parameters: a boolean condition, a function containing the code that shall be executed if the condition evaluates to true and a function that contains the code that should be executed in any other case. It is provided in a curried form here, meaning that it is defined as a chain of functions that each take exactly one argument.

Typing `ifElse` poses various difficulties. A first idea for the type of `ifElse` could be

\[
\text{immutable Boolean} \to \forall T. (\text{Unit} \to T) \to (\text{Unit} \to T) \to T.
\]

Obviously, `ifElse`’s type needs to be polymorphic in the return types of its lambda arguments and in its own return type\(^1\). Therefore, it is reasonable to abstract on the type \(T\).

However, this typing completely ignores the environments of the if and else blocks. Note that neither of the lambdas is allowed to take parameters but both are allowed to access objects in scope. The access permissions that are needed for these accesses are thus described solely via the respective lambda environments. Because it is not known in advance how these environments will look like, the type of `ifElse` needs to also be polymorphic in the lambda environments. In the following paragraphs, it will be assumed that lambda environments are types as well\(^2\). Hence, it becomes possible to abstract on them using universal types.

So a new idea for the type of `ifElse` is

\[
\text{immutable Boolean}[\cdot] \to \forall T. \forall A. (\text{Unit}[A] \to T)[\cdot] \to \forall B. (\text{Unit}[B] \to T)[\cdot][A \oplus B] \to T.
\]

Note that the lambda environment of the last part is declared as \(A \oplus B\). Because it is not statically known which lambda is called—the one for the “if” part or the one for the “else” part—, it is necessary to collect the possible side effects and make a conservative approximation. The “merge” operation \(\oplus\) describes this approximation.

As seen in section 3.2, a lambda environment consists of a list of identifier-type bindings. Concerning the permission part of the types, merging two lambda

\(^1\)Actually, the return types do not have to match up exactly. It is sufficient if the type of the value that the whole expression evaluates to is a supertype of the return types of the two lambda arguments.

\(^2\)Note that a lambda environment is not a syntactic entity as objects or functions are. For example, it does not make sense to declare a variable of a lambda environment type. Therefore, lambda environment types must be of a different kind than regular types.
environments $\alpha$ and $\beta$ works as follows. For each identifier in $\alpha$, it is checked whether an entry for the same identifier exists in $\beta$. If both $\alpha$ and $\beta$ contain an entry for an identifier $x$, the entry with the stronger permission is added to $\alpha \oplus \beta$. For example, if $\alpha$ contains `immutable T $\gg$ immutable T x` and $\beta$ contains `unique T $\gg$ unique T x`, $\alpha \oplus \beta$ contains `unique T $\gg$ unique T x`. If the permissions change, i.e. if pre-permission and post-permission differ, it is safe to assume the stronger pre-permission and the weaker post-permission for the entry in $\alpha \oplus \beta$. For example, if $\alpha$ contains `unique T $\gg$ immutable T x` and $\beta$ contains `immutable T $\gg$ unique T x`, $\alpha \oplus \beta$ contains `unique T $\gg$ immutable T x`. If an identifier-type binding is only present in one of the environments, it is added to $\alpha \oplus \beta$ without changes.

Note, however, that also the state part of the types in the environment can change. For this reason, the merge operation shares some of the issues of the subtyping relationship for function types that has been discussed in section 3.2. Assume, without loss of generality, that $\alpha$ mentions an identifier $x$ that is not contained in $\beta$. It is not allowed for $\alpha$ to specify that $x$ changes its state from $T$ to $U$ with $T \neq U$. If it were allowed, it could happen that the linear typing context would be wrongly updated. Since a conservative approximation is made, the entry `unique T $\gg$ unique U x` would also be contained in $\alpha \oplus \beta$. This means that the linear context would contain the type assumption `unique U` for $x$ after typing the whole `ifElse` expression. However, if the lambda whose side effects are described by $\beta$ is called at runtime, $x$ does not change its state. Therefore, all entries that are only contained in $\alpha$ and not in $\beta$ must be of the form `p T $\gg$ p T x`.

The example shown in listing 3.12 illustrates the difficulties that `ifElse` poses for the dependency analysis.

```java
1  method immutable Boolean predicate(immutable Object x);
2  method void doSomething(unique Object x);
3  method void doSomethingElse(immutable Object x);
4  
5  method void foo(unique Object x) {
6    ifElse (predicate(x)) { 
7      doSomething(x);
8      } { 
9      doSomethingElse(x);
10     };
11  }
12  doSomethingElse(x);
```

**Listing 3.12:** Plaid code that uses `ifElse`.

In this example, the if part needs a `unique` permission to $x$ while the else part only needs an `immutable` permission. Because this program has to be analyzed statically, it is necessary to make the second call to `doSomethingElse` depend on the whole `ifElse` expression. Otherwise, depending on the branch that is taken, there would be the possibility of a data race on $x$. As the dependency analysis inspects the types in the AST, it is important to examine how this example looks in the internal syntax. Listing 3.13 shows this representation.
The main issue here is that when \texttt{t3} is called, its type does not reflect the full lambda environment yet. An explicit intermediate step could help to solve this problem as demonstrated in listing 3.14.

After introducing the intermediate temporary variable \texttt{t5}, its type

\[
\texttt{t5} : \text{(Unit[unique Object $\gg$ unique Object x] $\rightarrow$ Unit)} \\
\quad [\text{unique Object $\gg$ unique Object x}] $\rightarrow$ Unit
\]

reflects the full lambda environment. The dependency analysis can now correctly indentify the dependency between the second call to \texttt{doSomethingElse} and the \texttt{ifElse} expression.

### 3.7 Borrowing for Local Variables

Listing 3.15 shows an example where a local variable acts as an alias.

```
1 method void foo(unique Object x) { 
2   val unique Object a = x; 
3 }
```

Listing 3.15: Local alias.
Here, a local alias \( a \) is defined that is declared with the same permission type as the reference it is initialized with, \texttt{unique} in this case. There are different ways of dealing with this aliasing situation. One possibility is to declare this a type error because the existence of two \texttt{unique} references to the same object is forbidden. An argument supporting this decision is that introducing an alias does not really increase expressiveness because if \( a \) is a better name, the programmer could have also named the parameter \( a \) instead of \( x \).

Another possible way is to allow local aliases but remove them using a local alias analysis during, for example, type checking. This gives the programmer more flexibility but creates the somewhat strange situation that, in the case of \( a \) and \( x \), two \texttt{unique} references to an object exist at the same time which violates the permission guarantees. It also makes code like \texttt{modify(x); modify(a);} hard to understand because it suggests that two different objects are passed, as both references are \texttt{unique}. However in reality, \( a \) is an alias to \( x \) and, when the dependency analysis becomes active, has already been removed and has been replaced by \( x \).

The third possibility is to split the permission of \( x \). Similar to a field read, \( a \) is granted the \texttt{unique} permission it requests and \( x \) is left with the residual permission, \texttt{none} in this case, using the split rule \( \texttt{unique}(x) \Rightarrow \texttt{unique}(x) / \texttt{none}(x) \) in the process. Hence, this is the only alternative that can be called borrowing. Following this approach, each use of \( x \) that requires more than a \texttt{none} permission and that lies between the assignment and the point where \( a \) goes out of scope will be considered a type error. As soon as \( a \) goes out of scope, it becomes possible to transfer the permission back to \( x \) again. This is also necessary for the method to be well-typed because, as \texttt{unique Object} \( x \) is a shorthand for \texttt{unique Object} \( \gg \texttt{unique Object} x \), the method signature of \texttt{foo} specifies that a \texttt{unique} permission to the argument is available after the call returns.

Only the last approach affects the dependency analysis, as borrowing for local variables is forbidden in the first approach and the second approach removes aliases before the dependency analysis even becomes active. The assignment in line 2 of listing 3.15 has an empty read set and the write set \{a, x\} because \( a \) is initialized and the assignment requires a \texttt{unique} permission to \( x \). After the assignment, \( x \) is only left with a \texttt{none} permission. If \( a \) goes out of scope before the end of the method body is reached, it becomes possible to transfer the \texttt{unique} permission back to \( x \) and use it accordingly, i.e. use it to modify the referenced object. However, the dependency analysis has to take care that expressions that use \( x \) and expressions that use \( a \) are not considered to be independent. Consider listing 3.16.

```java
1  method void modify(unique Object x);
2  method void foo(unique Object x) {
3     {  
4        val unique Object a = x;
5        modify(a);
6     }
7  }
8  modify(x);
9 }
```

Listing 3.16: Example illustrating the need for join nodes.
This example is based on the assumption that blocks can be opened at arbitrary positions in a method body and that the usual scoping rules of Java or C apply. Thus, the local variable \(a\) has block scope and goes out of scope in line 6. Note that such block expressions are currently not part of the Plaid language specification [PG10a] but a similar situation can arise if local variables are defined in `case` blocks that are part of a `match` expression. It is clear that the second call to `modify` has to depend on the first one. However, it is not apparent to the dependency analysis that the `unique` permission to the object is transferred from \(a\) back to \(x\) when the end of the block is reached.

Thus, as the type checker keeps track of the permission flow, it has to insert additional information into the AST to inform the dependency analysis of the permission flow. One possibility are `join` nodes that provide the needed connection between \(a\) and \(x\). If a join node of the form `Join[unique Object a / none Object x \(\Rightarrow\) unique Object x]` is inserted into the AST before the final call to `modify` in line 8, the dependency analysis has enough information to set up the correct dependencies. Listing 3.17 shows the above example represented in the internal syntax. In this case, the join node is considered to be both a write to \(x\) and a write to \(a\). Therefore, the join node functions as a “bottleneck” which prevents uses of \(a\) prior to the join node and uses of \(x\) after the join node from being viewed as independent.

```
method void foo(unique Object x) {
  let a = x in
  let t0 = modify(a) in
  join[unique Object a / none Object x \(\Rightarrow\) unique Object x] in
  modify(x)
}
```

Listing 3.17: Example from listing 3.16 in internal syntax.

To conclude, allowing local aliases enables the programmer to directly influence the splitting behavior of the type checker. Moreover, local aliases pose additional obstacles for the dependency analysis. It is therefore a legitimate question whether local aliases should be allowed, especially when taking into account that they do not increase expressiveness. Everything that can be done using \(a\) and \(x\) in this example could have also been achieved by just using \(x\) and letting the type checker automatically split permissions as necessary.

### 3.8 Java Interoperability

One of Plaid’s design goals is to provide good interoperability with Java. Making it easy to call Java code from Plaid code gives Plaid programmers instant access to Java’s extensive standard library. However, it also raises a lot of questions regarding the integration of Java types into Plaid’s type system. Furthermore Java methods do not expose their side effects in the way that is needed by Æminium.

Regarding the latter issue, there are two possible solutions. One possibility is to wrap calls to Java methods in Plaid methods that provide the necessary type information, as demonstrated in listing 3.18.
In this case, the method requires a unique permission to the stream it prints to and an immutable permission to its argument \( n \). Note that the method is annotated with @DoNotParallelize. This is important because the whole purpose of the method is to add an abstraction layer between the Java methods without effect descriptions and the Æminium compiler. Hence, the annotation means that this method should be compiled using the non-parallelizing version of the compiler that does not rely on type information.

An alternative way is to provide a method signature for Java methods that fits the Æminium system. The developer can then inform the Æminium system using an annotation like the one shown in listing 3.19 about all the effects of a method.

```
@AEminiumType(PrintStream.print,
   (immutable Integer -> Unit)[unique PrintStream this])
```

Listing 3.19: Providing an annotation.

### 3.9 Syntax Changes

As seen in section 2.2.4, the Æminium extension to Plaid is fairly syntax-heavy. Therefore, the necessary syntax extensions need to be carefully designed not to interfere with other present or planned language features of Plaid.

It is planned for Plaid to include abstract type members in a similar fashion as Scala [Oa04] does. Type members are members of a state that do not carry a value but a type. In contrast to Scala, Plaid does not make a distinction between type members and type parameters. In fact, type parameters are treated as syntactic sugar and mapped to type members internally.

```
state ListElement {
  type T;
  val T value;
}
val x = new ListElement with { type T = immutable Integer; };
```

Listing 3.20: Usage of abstract type members.

Listing 3.20 shows how a state for a generic list element looks when using type members. To instantiate the state with a concrete type, the with keyword is used. In the structural type that is supplied on the right hand side of the composition operator, the type member can be bound to a specific type, in this case the nominal type Integer. To simplify the definition of generic nominal types, additional syntactic sugar that resembles Java’s syntax for generic classes can be defined.

\(^3\) See sections 5.1 and 5.2.
Consider the examples in listing 3.21. In line 1, the type members are bound explicitly by name. In line 2, the type members are bound implicitly by using the order in which they are defined in the state. This of course forces the type system to keep track of the definition order.

Establishing an order on the type members of a state is also important when looking at state composition in the presence of type members.

In listing 3.22, two states Foo and Bar are defined which both define type members A and B. However, they define the type members in a different order which means that the composite state Comp needs to adopt either A’s or B’s ordering. A simple rule to define an ordering for the type members in the resulting state $Z = X \text{ with } Y$ is to take the order from X and add all additional type members at the end in the order that they appear in Y. Type members that are present in both states are added using X’s order. In the example, ordered type members of Comp are T, A, B, D, C, F, E. Note that the implication of this is that state composition is not symmetric on a syntactic level any more because Bar with Foo would result in a different ordering of the type members. However, on the type level Foo with Bar and Bar with Foo still represent the same type because they contain the same type members.

Because Æminium’s data groups behave a lot like type parameters, essentially the same reasoning can be applied to group members.

Listing 3.21: Syntactic sugar for defining generic nominal types.

Listing 3.22: Importance of type member ordering when composing states.

Listing 3.23: Definition of group members.

Listing 3.23 shows the definition of a state S with two group members, owner and G. Analogously to type members, different syntactic styles can be used to
bind the data groups. As mentioned before, the third variant that is used in
the definition of u depends on having a defined order for the group members.

The programmer must also be able to declare state-internal data groups (inner
data groups). This is possible using an explicit assignment syntax, as demon-
strated in listing 3.24.

```
1 state LinkedList {
2     group owner;
3     group data;
4     group internal = new group;
5 }
```

**Listing 3.24:** Definition of an inner data group.

The use of the `new` keyword here does not imply a runtime operation but a static
initialization. Just like type members, group members cannot be reassigned.

Note that `owner` is a keyword. As discussed in section 2.2.4, every shared
object belongs to exactly one data group. This data group is called the owner group
of the object. As the owner group plays an important role for certain data group
specific operations, e.g. `unpackInnerGroups` which will be discussed shortly, it
makes sense to enforce a standardized name for the owner group. One could
also use the convention that the first data group defined in a state becomes
the owner group but having a special name for this data group makes it easier
to understand code. If arbitrary names were allowed for the owner group, the
developer would always have to check the order of group definitions in the
particular state.

To access a shared object, the necessary group permission to its owning data
group needs to be present. Just like with access permissions, methods need
to specify their required group permissions in their signature. To keep data
group parameters separate from regular parameters, the syntax as illustrated
in 3.25 is used.

```
1 state S {
2     method void foo<shared group G>() {
3     }
4 }
5
6 method void foo<shared group G>() {
7 }
8
9 method void bar<shared group H>() {
10    foo<H>();
11 }
12
13 val unique S s = new S;
14    s.foo<H>();
15 }
```

**Listing 3.25:** Data group parameters for methods.

The use of angle brackets instead of regular brackets prevents ambiguities in
the case that arrays are added to Plaid with the usual array indexing syntax.
Consider an expression like `o.m[x]()`. If `o` has a type that defines a method
`m`, this would have to be considered a method call with `x` being a type or group
parameter. However, if `o.m` has an array type and stores values of function
type, this expression actually selects a specific function in the array and calls
it. Angle brackets avoid these problems.
As discussed before, data group permissions are not split and joined automatically but manually by the user. The share block shown in 3.26 allows splitting exclusive or shared group permissions into an arbitrary number of shared permissions.

```
1 share (G_1, ..., G_n) {
2     e_1;
3     || e_2;
4     ... 
5     || e_n;
6 }
```

**Listing 3.26:** Share block construct.

More specifically, each expression $e_i$ in the share block is supplied with its own shared permission to the group. The expressions are separated by the $|$ symbol to clarify that the expressions are run in parallel. If multiple expressions in the share block request a unique permission to the same object or an exclusive group permission to a group whose permission is not split by the block itself, this is considered to be a type error. Upon completion of the share block, all shared group permissions of the accessed data groups are joined back to the group permission that existed before the block was entered.

To protect against data races, Æminium forces users to enclose accesses to shared objects with an atomic block. The general form of the atomic block statement is shown in listing 3.27.

```
1 atomic (G_1, ..., G_n) {
2 }
```

**Listing 3.27:** Atomic block construct.

Atomic blocks transform a shared group permission to each data group that is mentioned to a protected group permission to the respective data group. As mentioned before, this provides the illusion of exclusive access and enables the user to use the data group as if she had an exclusive group permission. Explicitly referring to the required data groups is not strictly necessary for atomic blocks because the required groups could be inferred from the types of the objects that are accessed inside the block. However, this is likely to make code harder to read because it is not immediately clear which data groups are used. Inferring the used data groups can still be useful if the inferred set of data groups is used to perform a check against the set of data groups explicitly specified in the atomic block. Upon completion of the atomic block, the data group permissions of the accessed data groups are reverted to the state they were in before the atomic block was entered.
Listing 3.28 shows an example where all language elements are used to implement a concurrent counter. Note that the unique permission to `c` is automatically split inside the share block as both increment and decrement require a shared permission to their argument. As both are passed a shared permission to data group `S`, the owner group of `c` is also bound to `S`. Inside the increment and decrement methods, access to the shared data is protected with atomic blocks which prevents data races. Listing 3.29 illustrates how group permissions to inner data groups are managed in the system.
This example shows an implementation of a doubly linked list in Æminium. The data group `data` represents the data group to which all the objects that are stored in the list belong. All items that the list consists of belong to the inner data group `internal` which is thus the owner group of all `LinkedListItem`s.

It is initially unclear which group permission needs to be associated with inner data groups. Intuitively, the group permission to the inner data group depends on the group permission that is currently available for the owner group. If only a `shared` group permission is available for the owner group, it would be unsound to provide an `exclusive` group permission to the inner group. To solve this problem, Æminium introduces the `unpackInnerGroups` construct. This block construct transitively `trades` the current group permission to the owner group for permissions to the inner data groups that do not violate permission guarantees. If an `exclusive` group permission to the owner group is available, it is consumed by the `unpackInnerGroups` expression and `exclusive` group permissions to the inner data groups are produced. If only a `shared` or a `protected` group permission to the owner group is available, `shared` group permissions to the inner data groups are produced. Analogously to `atomic` and `share`, all group permissions are reverted to their original state when the end of the block is reached.

Note that in listing 3.29, two versions of a method that adds an element to the list exist. They only differ in the group permission they require to the owner
group. In the version that requires an exclusive permission to the owner, there is no need for atomic blocks as inside the *unpackInnerGroups* block exclusive permissions to all inner data groups are available and thus unsynchronized access is safe. The other method only requires a shared permission to the owner and therefore has to acquire a protected permission to the inner data group before accessing the contained objects. It is planned to allow the specification of generic group permissions. These would enable the programmer to write just one add method. The compiler could then generate multiple versions of the code for the method with varying degrees of synchronization depending on the available group permission.

The position of *unpackInnerGroup* blocks could be inferred automatically because every time a group permission to an inner data group is requested but currently not available, it has to be acquired via an *unpackInnerGroup* block. If the the required group permission cannot be provided because the available group permission to the owner group is too weak, this is regarded as a type error.

### 3.10 Tracking Data Groups

The dependency analysis that has been discussed before can be extended to also deal with data groups. As access to shared references is mediated through data group permissions, it is sufficient to look at the flow of data group permissions.

Like the previous section has demonstrated, functions and methods need to specify their required group permissions in their signature and respectively in their type. If read and write sets are extended to also contain data groups, this allows determining dependencies between expressions that use data groups. Consider listing 3.30.

```
1 method void f<exclusive group G>() {} 
2 method void g<exclusive group G>() {} 
3 method void h<exclusive group G>() {} 
4 
5 method void foo<exclusive group A, exclusive group B>() { 
6 f<A>(); 
7 g<A>(); 
8 h<B>(); 
9 }
```

Listing 3.30: Expressions using data groups.

In this example, a method *foo* that requires two exclusive permissions to data groups *A* and *B* is defined. All three methods *f*, *g* and *h* that are called inside also request exclusive permissions. As *f* and *g* require an exclusive permission to the same data group *A*, the call to *g* has to wait until the call to *f* has finished. Thus, if an expression needs an exclusive permission to a data group *G*, this is treated as a write to *G* and consequently, *G* is put into the expression’s write set. Because *h* accesses a different data group *B* and exclusive, just like unique, guarantees us that *A* and *B* cannot refer to the same data group, the call to *h* is independent from the other two calls.
Due to the fact that group permissions are not split and joined automatically, the group permission flow in other cases always adheres to the order that the constructs have been written down in the program. Consider the example in listing 3.31.

```
method void foo<exclusive group G, shared group H>() {
    share (G) {
        share (G) {
        }
        share (H) {
        }
        share (H) {
    }
    atomic (H) {
    }
    atomic (H) {
}
}
Listing 3.31: Dependencies between share blocks.
```

Both `share` blocks that depend on `G` are supposed to be executed one after the other. The same applies to both `share` blocks that split the group permission `H`. And just like that, also the second `atomic` block has to depend on the first one. Hence, `share` and `atomic` blocks are always considered to be a write their respective data groups, regardless of whether they require a `shared` or and `exclusive` group permission.

In listing 3.31, the blocks involving `G` can run in parallel to those involving `H` unless additional permission flow induces a dependency between the expressions. The reason for this is the disjointness property of data groups. Additional dependencies could, for example, be introduced if all blocks required a `unique` permission to the same object.

Finally, `unpackInnerGroups` blocks do not explicitly mention the data groups whose group permissions are potentially modified. An `unpackInnerGroups` block always implicitly operates on the owner group and all inner data groups. Thus, it must be considered a write to the owner group and all inner data groups.

```
state S {
    group owner;
    group inner = new group;
    method void foo<exclusive group owner>() {
        inner = new group;
    }
    method void bar<exclusive group owner>() {
        unpackInnerGroups {
            // Operate on inner group
        }
        foo<owner>();
    }
}
Listing 3.32: Example involving `unpackInnerGroups`.
```

In listing 3.32, both the unpack expression and the following call to `foo` are considered writes to the owner group. Hence, the method call depends on the previous unpack expression.
3.11 Code Generation & Granularity

After the dependency analysis, the dependencies between all the expressions in the program are known. This means that it is possible to put each expression into a task and then set up the inter-task dependencies accordingly. Because each expression corresponds to one task, the conversion from expression dependencies to task dependencies is trivial. However, because access permissions allow Æminium to extract very fine-grained parallelism, the issue of controlling granularity becomes important. Creating and scheduling tasks poses a significant overhead, so tasks should be reasonably sized or otherwise the costs of creation and scheduling quickly outweigh the benefits of concurrent execution.

There are cases where it is always better to assign two expressions to the same task instead of putting each in its own task. Figure 3.6 shows a chain of expressions $E_i$ where each expression depends on its predecessor. This can happen if each expression $E_i$ needs a unique permission to the same object $o$ which creates dependencies that correspond to the order that the expressions have been written down in in the program. In this case, putting every expression in its own task would not yield higher parallelism but would significantly increase the necessary overhead.

![Figure 3.6: Linear dependencies.](image)

However, it is not always possible to put multiple expressions in one task without reducing parallelism. Let $B \rightarrow A$ denote the relationship that $B$ depends on $A$ and suppose that $A$ and $B$ are two expressions with $B \rightarrow A$. If there exists an expression $C$ with $C \rightarrow A$, like shown in figure 3.7, then putting $A$ and $B$ into the same task reduces parallelism. This situation can arise if expression $A$ initializes a variable $x$ and $B$ and $C$ are independent but both take an immutable permission to $x$. Putting $A$ and $B$ into the same task would prevent $B$ and $C$ from running concurrently.

![Figure 3.7: Putting $A$, $B$ and $C$ into the same task reduces parallelism.](image)

Also, $A$ and $B$ should not be put into the same task if there exists a third expression $C$ that $B$ depends on, as illustrated in figure 3.8. This dependency configuration can occur if expressions $A$ and $C$ need an immutable permission to some variable $x$ and $B$ needs a unique permission to $x$. Assuming that $A$ and $C$ are otherwise independent, it is possible to run them in parallel if they are put into separate tasks. Thus, if $A$ and $B$ are put into one task, $A$ and $C$ are prevented from running concurrently.
To summarize, it is always advantageous to put expressions \( A \) and \( B \) with \( B \rightarrow A \) into the same task iff no third expression \( C \) exists with \( C \rightarrow A \) or \( B \rightarrow C \). In these cases, no possible parallelism is prevented while the overhead induced by task creation and scheduling is decreased.

However, it can also be beneficial to reduce parallelism.

Consider the code shown in listing 3.33. Let \( f \) represent a method whose execution time is very small. In this example \( f \)'s body is actually empty but it can be as well assumed that it contains a trivial operation such as incrementing an integer. The dependency analysis determines that the two calls in \( g \) are independent and, following the reasoning in the paragraphs above, the calls will not be put into the same task because the parallelism needs to be preserved. In this case, however, the costs for creating tasks and scheduling them in the \( \text{Æminium} \) runtime clearly outweigh the benefits of executing both calls in parallel. Therefore it would be more sensible to generate a single task for \( g \)'s body and lose parallelism.

However, in order to be able to make this decision, the compiler must have a notion of “cost”. Using such a cost model, it becomes possible for the compiler to assign an abstract cost value to expressions in the program. Based on these estimated costs, heuristics can decide whether to sacrifice possible parallelism or not.

Additionally, granularity problems occur for typical divide and conquer algorithms. A divide and conquer algorithm divides the initial problem into two or more sub-problems until the sub-problems are trivial and can be solved directly. The computation of the \( n \)-th Fibonacci number can be regarded as one of the simplest instances of this class of algorithms.

Consider the code shown in listing 3.34. Let \( f \) represent a method whose execution time is very small. In this example \( f \)'s body is actually empty but it can be as well assumed that it contains a trivial operation such as incrementing an integer. The dependency analysis determines that the two calls in \( g \) are independent and, following the reasoning in the paragraphs above, the calls will not be put into the same task because the parallelism needs to be preserved. In this case, however, the costs for creating tasks and scheduling them in the \( \text{Æminium} \) runtime clearly outweigh the benefits of executing both calls in parallel. Therefore it would be more sensible to generate a single task for \( g \)'s body and lose parallelism.

However, in order to be able to make this decision, the compiler must have a notion of “cost”. Using such a cost model, it becomes possible for the compiler to assign an abstract cost value to expressions in the program. Based on these estimated costs, heuristics can decide whether to sacrifice possible parallelism or not.

Additionally, granularity problems occur for typical divide and conquer algorithms. A divide and conquer algorithm divides the initial problem into two or more sub-problems until the sub-problems are trivial and can be solved directly. The computation of the \( n \)-th Fibonacci number can be regarded as one of the simplest instances of this class of algorithms.

Consider the code shown in listing 3.34. Let \( f \) represent a method whose execution time is very small. In this example \( f \)'s body is actually empty but it can be as well assumed that it contains a trivial operation such as incrementing an integer. The dependency analysis determines that the two calls in \( g \) are independent and, following the reasoning in the paragraphs above, the calls will not be put into the same task because the parallelism needs to be preserved. In this case, however, the costs for creating tasks and scheduling them in the \( \text{Æminium} \) runtime clearly outweigh the benefits of executing both calls in parallel. Therefore it would be more sensible to generate a single task for \( g \)'s body and lose parallelism.

However, in order to be able to make this decision, the compiler must have a notion of “cost”. Using such a cost model, it becomes possible for the compiler to assign an abstract cost value to expressions in the program. Based on these estimated costs, heuristics can decide whether to sacrifice possible parallelism or not.
As shown in listing 3.34, the base case for \( n \leq 2 \) just returns 1. In all other cases, the problem is split into two smaller sub-problems, computing the \((n-1)\)-th and the \((n-2)\)-th Fibonacci number, which are then used to compute a solution for the original problem. Because of this approach, the call graphs of functions that implement a divide and conquer algorithm show a typical tree-shaped structure. Figure 3.9 visualizes the calls that are made to compute \texttt{fib(4)}. Note that each leaf node in the graph represents a base case as defined in the recursive function. In the case of the Fibonacci function, each leaf node just evaluates to 1.

![Call graph for \texttt{fib(4)}.](image)

Divide and conquer algorithms often lend themselves very well to parallelization because the sub-problems can be solved independently. Speaking in terms of the Æminium runtime, it is thus necessary to put the independent calls into separate tasks. However, as discussed above, some of the calls are trivial and, for example, just return an integer constant. Hence, the costs for creating a task greatly outweigh the benefits of parallel evaluation on the lower levels of the tree.

To efficiently implement divide and conquer algorithms, \textit{cutoff thresholds} are used. The cutoff threshold is usually an integer value that is somehow derived from the input parameter. In the case of the Fibonacci function, the argument can be used directly; in the case of a sorting function that takes a list as its argument, the threshold could be expressed in terms of the length of the list. The threshold is used to decide when to stop the creation of new tasks and continue task-locally instead.

```plaintext
1 @DoNotParallelize
2 method unique Solution taskLocally(immutable Instance inst);
3
4 method unique Solution divideAndConquer(immutable Instance inst) {
5   ifElse (size(inst) <= cutoffThreshold) {
6     taskLocally(inst);
7   } {
8     val instPair = divide(inst);
9     val inst1 = instPair.first();
10    val inst2 = instPair.second();
11    combine(divideAndConquer(inst1), divideAndConquer(inst2));
12  }
13 }
```

\textbf{Listing 3.35:} General divide and conquer implementation using a cutoff threshold.
Listing 3.35 shows how a general divide and conquer implementation using a cutoff threshold could look like in Plaid. In this piece of code, it is checked whether the size of the current problem instance is below the cutoff threshold. Given that this is the case, a special `taskLocally` method is called. This method would need to be annotated with the special annotation @DoNotParallelize. The annotation would force the compiler to generate sequential code for the annotated method. This means that `taskLocally` would completely run in the context of the calling task, as long as `taskLocally` does not directly or transitively call other methods that spawn new tasks. If the instance size is greater than the threshold, the problem is divided into two sub-problems and it is recursed on each one. Because it can be deduced that both recursive calls are independent, they are put into different tasks and executed in parallel. Note that this assumes that the cutoff threshold is greater than the threshold for the base case, so the base case is actually handled inside `taskLocally`.

However, forcing the programmer to manually tune the cutoff threshold in order to achieve good performance on their current hardware platform contradicts Æminium's ambition to provide automatic parallelization. So far, all means of granularity control that have been presented were static mechanisms. Their goal was to optimize the structure of the task graph that is generated for the body of a method at compile time. One approach that could be worthwhile pursuing is dynamic granularity control.

Dynamic granularity control mechanisms are based on a feedback loop between the code that is generated by the compiler and the runtime. This means that the compiler has to generate two different code paths, one that introduces more parallelism and one that does not. To decide between the code paths, a call to the runtime is placed in the code. The runtime can then decide dynamically, i.e. under consideration of the current resource usage, if more parallelism is needed or not. The critical property of this approach is the heuristic that is used to select the code path. This heuristic must be easy to evaluate because otherwise the decision process slows down the program. At the same time it must be complex enough to avoid greatly overloading or underloading the system.
4 Implementation

The prototype of the Æminium compiler has been developed on the basis of the compiler [PG11] for the Plaid programming language whose compiler pipeline is shown in figure 4.1.

![Diagram of Plaid compiler pipeline.](image)

**Figure 4.1:** Plaid compiler pipeline.

The Plaid compiler takes a Plaid source file as input and produces a Java source file as its output. Its generated Java code contains calls to a special runtime library, the Plaid runtime. After being compiled by an unmodified Java compiler, the resulting output files are executed on a standard Java virtual machine. During execution, calls into the Plaid runtime help to keep track of the current program state. Note that because of the necessary runtime component, the Plaid compiler cannot be regarded as a pure source-to-source compiler.

![Diagram of Æminium compiler pipeline.](image)

**Figure 4.2:** Æminium compiler pipeline.

The Æminium compiler takes the same approach but modifies the Plaid compiler in certain places, as illustrated by figure 4.2. As discussed in the previous chapter, an additional dependency analysis needs to be inserted into the workflow. Moreover, the code generation phase needs to be adapted so that it generates code that makes use of the Æminium runtime system. Figure 4.3 shows the workflow inside the Æminium compiler. Italics indicates modifications to a pre-existing compiler pass; completely new passes are written in a bold font. Because the dependency analysis operates on type information, it
needs to run after the type checking pass. After that, the modified code generation phase wraps the Plaid code into Æminium tasks and can leverage the computed dependency information to set up the correct task dependencies.

Figure 4.3: Workflow inside the Æminium compiler.

4.1 The Plaid Runtime

The Plaid runtime manages the current state of program execution and keeps track of all variables and objects inside a Plaid program. It can also establish references to the original Plaid source code which increases the quality of error messages. Moreover, it serves as an interface to debuggers or profilers.

Figure 4.4 shows the part of the Plaid runtime class hierarchy that will be discussed in further detail in the following paragraphs. The PlaidRuntimeMap class represents the actual runtime and exposes all services the runtime provides through its interface. It is implemented following the singleton design pattern. The main task of PlaidClassLoaderMap is to manage the dynamic loading of Plaid states that are referenced inside the Plaid program. It is also implemented as a singleton. The PlaidObject interface represents the interface that all objects that represent actual values inside the Plaid program have to implement. For example, the definition of a lambda results in the creation of a PlaidFunctionMap object. Not all classes that implement the PlaidObject interface are shown in the graph. Finally, the PlaidScope hierarchy contains the classes that manage the resolution of names in the Plaid program and map them to the correct PlaidObjects.

Regarding the usage of the runtime in the generated code, one of the most important differences between the Plaid compiler and the Æminium-enabled compiler is that the latter generates code where multiple tasks may use the Plaid runtime at the same time. If the runtime is used in such a concurrent fashion, it needs to be thread-safe or otherwise it cannot correctly maintain the program state. However, as discussed in the previous chapters, Æminium statically guarantees freedom from data race. This raises the question of why the Plaid runtime needs to be thread-safe if data race freedom is enforced before the code is even executed. After all, if the dependency analysis works correctly, the task dependencies should be set up in a way that prevents data races. However, this is only true for the actual objects that the permission-annotated references in the program refer to. Only the types of variables in the Plaid program are tracked by the type system, other references that may exist in the generated Java source are not subject to any static analysis against data races. This means that every object that implements the PlaidObject interface does
not have to be thread-safe because it is already statically protected by Plaid’s type system and Æminium’s dependency analysis.

However, any other object which does not represent a value in the Plaid program and that is still potentially accessed by concurrent tasks needs to be designed in a thread-safe way. In the following paragraphs, it will be discussed how this has been achieved for the other classes shown in figure 4.4. Both the PlaidRuntimeMap and the PlaidClassLoaderMap classes are implemented as singletons which is problematic in a multithreaded environment. Listing 4.1 shows the typical way of implementing the singleton pattern in Java.

```java
1 class Singleton {
2     private static Singleton instance = null;
3
4     private Singleton() {
5         }
6
7     public static Singleton getInstance() {
8         if (instance == null) {
9             instance = new Singleton();
10         }
11
12     }
```

Listing 4.1: Singleton implemented in Java.
Because the lines 8 and 9 are not atomic, various problems can occur. For example, multiple threads could evaluate `instance` to `null` and thus multiple Singleton objects would be allocated. A simple way of making the shown Java code thread-safe is to declare the `getInstance()` method as synchronized. However, this means that synchronization is performed every time `getInstance()` is called. As the runtime singleton serves as the central mediator to all services provided by the runtime, this method gets called very often and should impose minimal overhead. By exploiting the guarantees of the Java memory model and declaring `instance` as volatile the synchronization overhead can be decreased significantly. This approach is described in further detail in [BBB+].

Figure 4.5 shows the PlaidScope class hierarchy in more detail. The PlaidScope interface offers four operations: looking up elements by name, inserting new elements, updating existing elements and removing elements. Inserting elements differs from updating existing elements in that the existence of an element with the specified name is considered an error. In contrast, `update` considers it an error if no such element exists. The scope objects are actually the main runtime data structures that are used in a concurrent fashion. More specifically, access to the members mutableScopeMap and immutableScopeMap that are defined in AbstractPlaidScopeMap needs to be synchronized.

![Diagram of PlaidScope class hierarchy](image-url)

**Figure 4.5:** The PlaidScope class hierarchy.

The two data structures can be protected against internal corruption by using a concurrent collection, for example a ConcurrentHashMap. However, this only secures thread-safety when accessing the internal fields but does not synchronize the methods that are exposed by the PlaidScope interface. This means that while individual operations like put and get on mutableScopeMap and immutableScopeMap are now thread-safe, there might still exist problems on a higher level. Consider the sample implementation of PlaidLocalScopeMap.lookup shown in listing 4.2.
4.2 The Æminium Runtime

Figure 4.6 shows the most important interfaces present in the Æminium runtime and their relationships. As discussed in section 2.3, the Æminium runtime is based on the idea of abstract tasks that each encapsulate a certain part of the functionality of a program. The Task interface represents the interface to such an abstract task and provides methods to set and query the result. Note that the getResult method is supposed to have blocking semantics. Hence, the call returns immediately if the task has already finished and otherwise blocks until this event occurs.

Tasks are scheduled for execution by calling the schedule operation defined in the Runtime interface. When scheduling a task, its parent task and its data dependencies on other tasks need to be specified. If a task $t_1$ specifies that it depends on another task $t_2$, this means that $t_1$ must not be executed before $t_2$.
has finished. Note that calls to `schedule` return immediately and trigger the execution of tasks asynchronously. Tasks can be created using the `createTask` operation. This operation needs to be passed an object implementing the `Body` interface and returns the newly constructed task object.

If a task spawns sub-tasks, those tasks are called children of the parent task. As can be seen in figure 4.6, the parent-child relationship can be established when scheduling a task by passing a reference to the parent task. If a parent task is specified, the parent implicitly depends on all its children. Therefore, the execution of an Æminium task is only considered to be completed—and thus its result only becomes available—when the execution of all its children is completed. In summary, the Æminium runtime differentiates between two types of dependencies. First, data dependencies on other tasks that have to be set up explicitly. And second implicit parent-child dependencies.

The `Body` interface only specifies the `execute` operation. An implementation of this method contains the actual code that is to be executed upon running the task. When `execute` is called it gets passed a reference `rt` to the `Runtime` object and a reference `current` to the task object it is part of as arguments. Inside its body, `execute` can use `rt` to create and schedule new tasks thereby giving it the possibility to introduce more parallelism by splitting its work into more possibly independent sub-tasks. It can also set the result of its enclosing `Task` object by calling `setResult` on `current`.

Listing 4.3 shows how the runtime is supposed to be used. Note that the bodies created in the example are implemented with an anonymous class. Upon execution of this program, task `t2` will wait five seconds until `t1` has finished and then print “Hello World”. The `createNonBlockingTask` method that is used in this example is not listed in figure 4.6 for presentation reasons. Different
methods are offered by the runtime to create different types of tasks. This task type information is used by the runtime to optimize scheduling behavior and helps to pass knowledge about certain tasks from the compiler down to the runtime.

```java
public class Example {
    public void main(String[] args) {
        Runtime runtime = Factory.getRuntime();
        runtime.init();
        Body b1 = new Body {
            public void execute(Runtime rt, Task current) {
                Thread.sleep(5000);
            }
        };
        Body b2 = new Body {
            public void execute(Runtime rt, Task current) {
                System.out.println("Hello World");
            }
        };
        Task t1 = runtime.createNonBlockingTask(b1, Runtime.NO_HINTS);
        Task t2 = runtime.createNonBlockingTask(b2, Runtime.NO_HINTS);
        Collection<Task> deps = new ArrayList<Task>();
        deps.add(t2);
        runtime.schedule(t1, Runtime.NO_PARENT, Runtime.NO_DEPS);
        runtime.schedule(t2, Runtime.NO_PARENT, deps);
        runtime.shutdown();
    }
}
```

Listing 4.3: Usage example for the Æminium runtime.

The `execute` method is where the Æminium compiler puts the code that is generated for the individual expressions in the program. Thus, the compiler creates a new body object wrapping the generated Plaid code, creates the enclosing task object for it, sets up the task dependencies and schedules it for execution.

### 4.3 Dependency Analysis

The dependency analysis pass is realized using the visitor pattern and closely follows the approach discussed in section 3.3. While traversing the type-annotated AST in evaluation order, the read and write sets are determined for each expression. The dependencies are then computed using the `Readers` and `Writer` functions described in section 3.1.

Note that special actions need to be performed when a lambda definition is encountered. In this case, the body of the anonymous function is not part of the program flow inside the current method. Therefore, the current dependency analysis needs to spawn a new instance of the dependency analysis visitor and run it on the subtree of the AST that represents the definition of the lambda.

The computed dependencies are saved directly to the respective AST nodes. Each dependency is annotated with the variable that induced the dependency. Consider the example in listing 4.4.
4. Implementation

```java
1  method void modify(unique T x);
2  method void read(immutable T x);
3
4  method void foo(unique T x) {
5    read(x);
6    read(x);
7    modify(x);
8  }
```

**Listing 4.4:** Dependency analysis example.

The permission flow in this program is straightforward. The `unique` permission is split and when `x` is used as an argument to the `read` calls, `x` has type `immutable T`. After the permission join, `x` is typed as `unique T` in the call to `modify`. Listing 4.5 shows the internal representation of this program with the matching read sets and write sets.

```java
1  method void modify(unique T x);
2  method void read(immutable T x);
3
4  method void foo(unique T x) {
5    let t0 = read(x) in // R = { read, x } W = { t0 } 
6    let t1 = read(x) in // R = { read, x } W = { t1 } 
7    modify(x);       // R = { modify } W = { x } 
8  }
```

**Listing 4.5:** Dependency analysis example in internal syntax.

There exist two dependencies here: the call to `modify` has to depend on both calls to `read`. Figure 4.7 shows the AST of `foo`’s body with the computed dependencies represented by dotted edges. Note that `modify` and `read` are modeled as global variables holding lambdas here, so the AST contains `Application` nodes instead of `MethodCall` nodes. The dependency annotated AST is the input for the following code generation pass.

![Figure 4.7: AST of foo’s body with dependencies.](image)

4.4 Code Generation

The current Plaid compiler pipeline does not include the transformation to an intermediate language yet. Therefore, code is directly generated from the AST.
and the code generation pass is, just like the dependency analysis, implemented as a visitor that traverses the AST. After the dependency analysis has finished, each AST node is annotated with the computed dependencies to other nodes like shown in figure 4.7. The code that is to be generated for the expressions needs to be distributed to multiple tasks by the code generation. Expressions that have been identified as independent by the prior analysis pass can be put into separate tasks that do not depend on each other.

Figure 4.8: Possible task partitioning.

Figure 4.8 shows a possible partitioning of expressions into tasks for the example from listing 4.4. Note that each of the Application nodes is associated with a different task. Before the tasks can be scheduled, the dependencies between the expressions need to be converted to inter-task dependencies. In the example at hand, task 3 depends on task 1 and task 2.

Following this approach, the structure of the task graph reflects the permission flow inside the program. Moreover, the compiler can exploit its knowledge about certain constructs in the program and wrap them in specialized task types provided by the Æminium runtime. For example, the runtime includes an AtomicTask class that is intended to be used for atomic blocks. An AtomicTask can refer to the set of data groups that the atomic block needs access to. In the generated code, each atomic block will correspond to the creation of an AtomicTask object. As seen in section 3.10, two consecutive atomic blocks that refer to the same data group are supposed to be executed one after the other. This can be expressed by a dependency from the second atomic task object to the first one.

Local variables that are also accessed by other tasks must be put in a location that is also accessible by other tasks. It is possible to identify these variables by looking at the incoming dependencies for a certain expression. If one of these dependencies is induced by a local variable, this variable needs to be stored on the heap. In the current implementation, this is handled by the PlaidScope mechanism.
The code in foo’s body can be viewed as a *data flow graph* consisting of three tasks. Such data flow graphs are generated for the body of each function in the Æminium program. However, function calls represent natural boundaries in this graph. When a function calls another function, the two data flow graphs need to be connected by an edge so that, ultimately, the whole Æminium program is represented by a connected data flow graph. In other words, the runtime system must be informed that the calling task is the parent of the tasks that the called function’s flow graph consists of.

As mentioned in section 4.2, the relationship between parent and child task needs to be setup when the child task is actually scheduled. Therefore, tasks that encapsulate the body of a called method need to be somehow connected to the task that encapsulates the call-site code. More specifically, the task containing the call-site code needs to have access to the task object that wraps the body of the called method so that the calling task can specify itself as the parent task when calling `Runtime.schedule()`. Therefore, every Æminium method *returns* the task that wraps its body.

Consider the example in listing 4.4 again. Following the presented approach, the code in the body of foo is put into its own task. The two read calls in foo are independent and are subsequently put into two distinct tasks that do not depend on each other. The task that wraps the call to modify has to wait until the two read tasks are finished and thus depends on those two tasks. Each of the calls to read or modify returns the task object wrapping the body of the respective function.

Figure 4.9 illustrates all task objects that are created as soon as the task representing foo is scheduled and executed. Each rectangle represents one task object. If a task is the child of another task, the rectangle representing the child task is contained within the rectangle that represents the parent task. The dashed rectangles hint at the tasks that the data flow graphs of read and modify consist of. Note that each of call induces the creation of two tasks. If it can be statically determined that the called method returns an Æminium task, the outer task can be eliminated as the returned task can be used instead.

Listing 4.6 shows a simplified version of the code that is generated for the example. For presentation reasons, the creation of Body objects has been omitted. The `new Task() { ... }` syntax is supposed to express the creation of a new Task object that encapsulates the code between the braces. Also, the parameter names to Runtime.schedule have been made explicit to increase readability. Note that when read and modify are called, the tasks that wrap the respective function calls are set as the parent tasks. Analogously, the foo task passes itself as the parent task when scheduling the tasks $t_1$, $t_2$ and $t_3$ that foo’s data flow graph consists of. Additionally, upon scheduling $t_3$ the necessary task dependencies to $t_1$ and $t_2$ are specified.
In order to maintain compatibility with nonÆminium functions, the interfaces present in the Plaid runtime must not change. Figure 4.10 shows the interface that all function objects in Plaid must implement. Multiple arguments are handled via tuples that are packed before a function call and are unpacked as part of the prelude of a function body. The interface that is implemented by Plaid methods is similar and also returns a value of type PlaidObject.
To make it possible for a Plaid function to return a task object, the Java interoperability features of Plaid can be used. As shown in figure 4.4, it is possible to wrap Java objects in `PlaidJavaObjectMap` objects and use them like `PlaidObject`s. Therefore, the task object can be packed into a `PlaidJavaObjectMap` object and can be returned by the function.

Each time a function is called, a runtime check is used to check whether the returned object is a regular Plaid object or a task. If it is a regular object, it is directly used as the return value of the function call. Otherwise, the task is scheduled for execution, whereupon the calling task passes itself as the parent, and the task’s return value is subsequently queried using `Task.getResult()`.

### 4.5 Granularity Management

Before generating code, additional granularity management can be applied as described in section 3.11 to increase the amount of work per task. Thereby the amount of overhead introduced by creating and scheduling individual tasks is decreased. In the current implementation, only optimizations that do never reduce the possible parallelism are implemented. Therefore, code that is generated for two expressions $A$ and $B$ is put into the same task iff $B$ only depends on $A$ and $B$ is the only expression that depends on $A$. Consider the example in listing 4.7.

```java
1 method void modify(unique T x);
2
3 method void bar(unique T x) {
4    modify(x);
5    modify(x);
6 }
```

**Listing 4.7:** Linear dependency chain.

In this example, putting each call to `modify` in a separate task is never advantageous as the calls must be executed sequentially. Thus, the data flow graph for `bar` just consists of one task that contains both calls. Figure 4.11 illustrates which tasks are created when `bar` is called.

Future versions of the code generation will focus on minimizing the number of tasks that are generated. This can be achieved by implementing the strategies described in section 3.11.
Figure 4.11: Generated tasks upon execution of `bar`.


5 Evaluation

In this chapter, the code that is generated for the Fibonacci example will be analyzed in further detail. This analysis will encompass a more detailed look at the structure of the generated task graph as well as a quantitative analysis. Listing 5.1 shows a Plaid program that computes the fifth Fibonacci number.

```plaid
package testInputs.fibonacci;

val (immutable Integer >> immutable Integer) -> immutable Integer fib = 
  fn (immutable Integer >> immutable Integer n) => {
    match (n <= 2) {
      case True {
        1;
      }
      case False {
        fib(n - 1) + fib(n - 2);
      }
    }
  }

method void main() [void >> void] {
  fib(5);
  unit;
}
```

Listing 5.1: Fibonacci program in Plaid.

In this implementation, the recursive function that computes the Fibonacci numbers is realized as an anonymous function that is stored inside the global variable `fib`. Because of the typing issues with Plaid’s `ifElse` construct that have been discussed in section 3.6, the example directly uses `match` to branch. To allow both calls to `fib` in the `False` case to run concurrently, the `immutable` permission to `n` needs to be split. As a type system without fractional permissions is used in the current implementation of the type checker, the permission to the argument must not be borrowed. Otherwise the type checker would have to prohibit all permission splitting in the body of `fib` because it could not guarantee the full recovering of the original permission. Thus, in the current type checker implementation, a function of type $T \rightarrow U$ is considered as borrowing the permission to its argument whereas a function of type $(T \gg T) \rightarrow U$ is considered as not borrowing the permission. This convention might be subject to change in future versions; for example a new keyword could be introduced to explicitly mark permissions to arguments as borrowed.

Note that the `main` method returns `unit` because the type of `fib(5)`, `immutable Integer`, is currently not a subtype of `void`. In the current implementation, `void` is a shorthand for `none Unit`. Thus, in order to be well-typed, `main` returns `unit` which is a predefined literal of type `none Unit`.

Figure 5.1 shows the task graph that is handed to the Æminium runtime upon running the program from listing 5.1. Because the task also includes edges
that represent a parent-child relationship in the graph, it is not called a data flow graph. The task graph encompasses all computations that are necessary to compute the program’s final result. Each node in the graph represents a task object $t_i$. Solid edges $(t_i, t_j)$ represent data dependencies, i.e. task $t_i$ needs the result of task $t_j$. Dashed edges $(t'_i, t'_j)$ express child-parent relationships, i.e. task $t'_i$ has been spawned by task $t'_j$ whose completion thereby implicitly depends on the completion of $t'_j$.

As described in section 4.4, each task that encapsulates the body of a function spawns sub-tasks that might enable the runtime to exploit concurrency. In figure 5.1, those sub-tasks are named $F:N$ where $F$ stands for the function whose body the sub-tasks are part of and $N$ is a number that is assigned by the compiler. The sub-tasks are not consecutively numbered because the numbers are initially assigned to expressions and then later reused for tasks. Hence, if the code that is generated for multiple expressions is put into the same task, only one of the numbers that had been assigned to the expressions is also assigned to the sub-task.

If one considers the graph that consists of the same nodes but whose edge set only contains the dashed edges, this graph is actually a tree. This tree has the typical shape that one would expect a tree representing the computation of a Fibonacci number using the conventional recursive algorithm to have. It is asymmetrical because the problem of computing the $n$-th Fibonacci number is divided into two smaller problems of different size. Because the same function $\text{fib}$ is called over and over again, there are also a lot of recurring patterns in the graph. In the following, the three different subgraphs that the complete task graph consists of will be analyzed in more detail.

Figure 5.2 shows the task graph of the main method. It consists of the task main and two independent sub-tasks main:1 and main:4. The dependency analysis determines that the expressions $\text{fib}(5)$ and unit can be safely evaluated in parallel and therefore the generated code for these expressions is put into separate tasks that do not depend on each other. Thus, the sub-task main:4 does nothing more than set its result to unit. The other sub-task main:1 encapsulates the function call to $\text{fib}$. Via the mechanism described in section 4.4, the task representing the body of $\text{fib}$ is connected to the calling task main:1 as indicated by the dotted node in the graph.

Figure 5.3 shows the task graph of $\text{fib}(n)$ if its argument $n$ is less or equal to 2 and thus the base case of the recursion is reached. The body task $\text{fib}$ spawns a single sub-task $\text{fib}:1$ which itself spawns a sub-task $\text{fib}:5$. The sub-task $\text{fib}:1$ contains the Plaid code that compares $n$ to 2 and feeds the result into the match expression. As there is a direct data dependency from the match to the method call that realizes the comparison, the generated code for both expression is put into the same task. Inside the match, the first case block becomes active and hence a sub-task $\text{fib}:5$ representing its body is spawned. Because the base case of the recursion is reached, this sub-task only sets its return value to 1. The return value is then propagated upwards in the task hierarchy so that the task $\text{fib}$ also sets its return value to 1.
Figure 5.1: Task graph for the Fibonacci example from listing 5.1.
Figure 5.2: Task graph for main method.

Figure 5.3: Task graph for base case.

Figure 5.4 shows the task graph of $\text{fib}(n)$ if its argument $n$ is greater than 2. The first sub-task that is spawned is obviously again the task $\text{fib}:1$ that encapsulates the comparison operation and the \texttt{match}. Contrary to figure 5.3 however, the second \texttt{case} block becomes active. The expressions $\text{fib}(n-1)$ and $\text{fib}(n-2)$ are packed into the tasks $\text{fib}:6$ and $\text{fib}:11$ respectively as indicated by the dotted nodes. Note that one can also deduce the fact that $\text{fib}:6$ contains the call to $\text{fib}(n-1)$ from the graph structure because the subtree\textsuperscript{1} of $\text{fib}:6$ has a larger height as $n - 1$ represents the larger subproblem. Because the dependency analysis has determined that both calls are independent, the corresponding code is put into separate, independent tasks which can be executed in parallel. As the add operation needs the values of its operands, the sub-task $\text{fib}:16$ depends on both tasks that compute these operands, namely $\text{fib}:6$ and $\text{fib}:11$. The result of the summation is propagated upwards in the task hierarchy and eventually $\text{fib}$'s result is set to this value.

Regarding the number of tasks that are generated, the task graph for the recursing branch is optimal in the context of a data flow architecture. If the two recursive calls to $\text{fib}$ are to be executed concurrently, they have to be put into separate tasks. Furthermore, the following summation of the results has to be put into its own task, too. If it was put into a task that also contains a call to $\text{fib}$, this would induce a data dependency on the other task that calls $\text{fib}$. Hence, the two calls would not run concurrently any more.

The two sub-tasks that are generated for $\text{main}$’s body illustrate very well the finely-grained parallelism that the system is able to extract. However, in this case it would have been more sensible to create a single task for everything

\textsuperscript{1}As mentioned before, it is only possible to speak of a tree when the edges representing data dependencies are ignored.
in the body of \texttt{main}. In order to come to this decision, the compiler needs a notion of which operations are expensive and which are not. This is beyond the scope of the current prototype implementation but has to be addressed in future versions to generate efficient code.

\section*{5.1 Initial Benchmarks}

In order to evaluate the performance of performing automatic parallelization in the way that has been described in the previous chapters, the Fibonacci example from listing 5.1 was compiled once using the regular Plaid compiler and once using the Æminium compiler. To generate a sufficient number of tasks and thus provide an interesting problem size, the program was changed to compute the 23\textsuperscript{rd} Fibonacci number instead of the 5\textsuperscript{th}. The resulting programs were run on an eight-core system using two Intel Xeon X5460 quad-core processors. The system had 32 gigabytes of main memory and ran under 64-bit Fedora GNU/Linux, version 7. The versions of the installed 64-bit Sun Java virtual machine and compiler were 1.6.0_16. For each data point, the corresponding program run was repeated 25 times and the average execution time was determined. To restrict the parallel version of the Fibonacci program to a certain number of cores, the \texttt{global.processorCount} configuration variable provided by the Æminium runtime was used.

Figure 5.5 shows the speedup of the parallel program that was compiled by the Æminium compiler versus the sequential program that was compiled by the regular Plaid compiler for \( n = 23 \), \( n = 25 \) and \( n = 27 \). The execution time of the sequential version is independent of the number of tasks as it does not use the permission information to perform any form of parallelization. On just one core, the running time of the parallel version is 18.2\% higher than the execution time of the sequential version. This was to be expected, as the creation and the scheduling of tasks poses a significant overhead. As soon as more than one core is used, the parallel version is faster than the sequential version. The speedup for the parallel version increases until it reaches its maximum for six cores. The maximum speedup versus the sequential version depends on the
input size. For $n = 23$ it is 2.18, for $n = 25$ it is 2.79 and for $n = 27$ it is 3.08. For more than six cores, the speedup decreases.

It is surprising that the parallel version is faster than the sequential version, at least when the programs are run on more than one core, even without any form of granularity management. As described in the previous section, the Æminium compiler does not try to determine a cutoff threshold to limit the number of tasks that are generated. Therefore, the costs of task creation and scheduling quickly outweigh the benefits of parallelization in the lower levels of the tree from figure 5.1. However, one has to take into account that the Plaid compiler itself is in an early stage and the usage of the Plaid runtime poses a substantial overhead. This overhead hides the additional costs that come with creating and scheduling the task objects. Thus, this result must not be over-interpreted. As soon as the Plaid compiler starts generating more optimized code, the lack of granularity control will lead to parallel code that runs at least one order of magnitude slower than the sequential version.

The fact that the speedup is higher for bigger input parameter $n$ is peculiar. As more tasks are generated for larger values of $n$, this means more management overhead for the Æminium runtime, so a lower speedup is to be expected. However, in this case the opposite effect can be observed. Additionally, it is concerning that the speedup decreases if more than six cores are used which indicates scalability problems.

To investigate the reason for these behavior, it has been tried to eliminate the influence of the Plaid runtime. In order to do that, programs were written that use the Æminium runtime and create the same task graph structure but put plain Java code into the body of the task objects. However, as it turned out, there exists a fundamental problem with this approach. As can be seen in figure 5.5 when looking at the speedup for one core, the Plaid runtime hides the overhead induced by the Æminium runtime. This means that most of the time is spent executing the code inside the tasks. In the case plain Java code is
used, the main portion of the execution time is spent on creating and managing
the task objects. Because of this different behavior, it is not valid to compare
the results obtained by running a Plaid/Æminium version of a program and
a Java/Æminium version. The precise analysis of the runtime behavior of
programs that interact with both the Plaid runtime and the Æminium runtime
will be left to future work.

To conclude, the Æminium prototype compiler demonstrates that it is possible
to automatically generate parallel code for small Plaid programs using access
permission information. The generated code targets the data flow architecture
of the Æminium runtime and at the same time uses the infrastructure provided
by the sequential Plaid compiler in the form of the Plaid runtime library.
While there is still a lot of room for improvement, the measured speedups are
acceptable for a prototype implementation. Note that although the parallel
code is even competitive with the sequential code in terms of performance on
one core, this does not render granularity control redundant. As mentioned
before, the overhead for task creation and scheduling is currently hidden by
the substantial overhead of the Plaid runtime itself. Depending on the future
implementation strategy, it might make sense to further optimize the Plaid
runtime for concurrent access. The current thread-safety enhancements to the
Plaid runtime have not been made with performance but with correctness in
mind. However, in the long run the Plaid compiler will use a code generation
strategy that does not rely on the Plaid runtime at all any more.
6 Conclusion

It has been shown how static information in the form of access permissions can be used to analyze the dependencies between expressions in a Plaid program. This dependency information has been used to generate parallel code that targets the Æminium runtime system. A code generation strategy has been developed and implemented on the basis of the Plaid compiler and the generated code has been evaluated.

As access permissions are integrated in the type system in Plaid, a preliminary type system for Plaid that is able to handle unique and immutable permissions has been discussed. On the basis of this type system, it has been shown how the dependencies between expressions can be set up by assigning read and write sets to them. These read and write sets are computed by tracking the access permission part of the type that is assigned to variables in the program by the type checker. It has been shown how this approach interacts with various language features present in Plaid and also with language extensions that will be made in the scope of the full integration of Æminium into Plaid. Moreover, the issue of coarsening the granularity of parallelism has been discussed.

A code generation strategy that leverages the dependency information to generate parallel code has been presented and evaluated. The prototype implementation has shown that the task of controlling the granularity is the most important problem that needs to be solved. At the same time, the quality of the code that is generated by the Plaid compiler needs to be improved to acquire more meaningful test results.

6.1 Outlook

The task of creating a usable compiler for the Æminium approach on the basis of Plaid has many facets. First of all, besides extending the syntax as described in section 3.9, the remaining parts of the Æminium system still need to be integrated into Plaid. Once Plaid’s type system has stabilized, it becomes possible to add the necessary elements for Æminium. As discussed in [SAM10], these are mainly data group permissions and a type for data groups. Moreover, the interaction of Æminium with gradual types and other access permissions that are not part of Æminium like pure and full needs to be investigated.

After implementing a type checker for this extended type system, the dependency analysis and the code generation passes can be adapted. This constitutes the implementation of a first prototype compiler that supports the full Æminium system and makes it possible to conduct usability studies. Such an evaluation is important because so far, it is unclear if programming in Æminium
is desirable from a programmer’s point of view. The advantages of data-race freedom and automatic parallelization come at the price of high annotation burden and a more complicated type system.

Furthermore, the problem of generating efficient code needs to be solved in order to make Æminium really useful in practice. As demonstrated in the previous chapters, a permission flow analysis can expose very fine-grained parallelism. In order to prevent the benefits of concurrent computations from being outweighed by the costs for creating and scheduling tasks, the granularity of parallelism needs to be controlled. Giving the programmer direct influence over the granularity is a feasible way of dealing with this situation. However, as expressed by its mission statement “Concurrency by Default”, it is Æminium’s ambition to free the programmer from having to manually tune their program to achieve good performance.

Therefore, both the compiler and the runtime system must be involved in solving this issue. A combination of static and dynamic means of controlling granularity as presented in section 3.11 could be the key to solve this issue. On the compiler side, a cost model could be used to decide whether the creation of a separate task for a certain part of the program is worthwhile or not. On the runtime side, it would be possible to decide between different code paths with varying degrees of parallelism based on the currently available resources. Moreover, the possibility of profile-guided optimization exists.

The introduction of an intermediate format in the form of the data flow graph also creates the possibility of basing additional tools like profilers on this intermediate format. These tools could help the programmer to analyze program behavior and identify bottlenecks on a higher level of abstraction.
Bibliography


