1 The Plaid Programming Language

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.

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

[1]Nullable fields are often used for this purpose.

method must be called to transition the object to that specific state. A file class, for example, only allows calls to `open()` if the object is in state `Closed`, and an iterator class only allows calls to `next()` if there are non-visited elements left in the collection. Another concrete example is Java's `BufferedReader` class, which allows the user to mark the current position in the stream, using `mark()`, and then later reset the stream to the marked position using `reset()`. Figure 1.1 shows a state machine for this portion of BufferedReader's behavior.

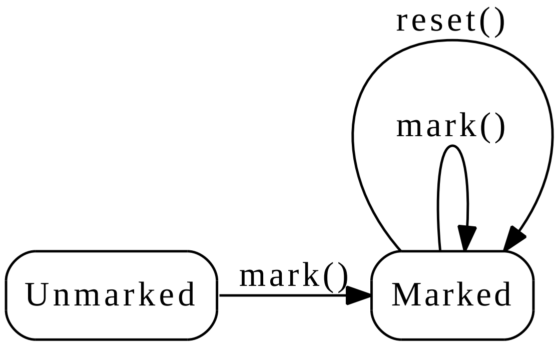

Figure 1.1: Abstract state machine for `BufferedReader`.

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 1.1: Simple protocol violation.

In this example, the intent of the `reset()` call was to reset the stream to the most recent mark—however, `mark()` is not called before which means that the reset call violates the usage protocol. Upon execution of the program, an `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 1.1. 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:

```
state MarkedReader {
 var Integer markPosition;
 method void reset() [MarkedReader >> MarkedReader];
 method void mark() [MarkedReader >> MarkedReader];
}

state UnmarkedReader {
 method void mark() [UnmarkedReader >> MarkedReader];
}
```

Listing 1.2: Usage protocol modeling using typestate.

In this Plaid snippet, two states called `MarkedReader` and `UnmarkedReader` are defined which represent the two states in figure 1.1. 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` $\neq -1$ 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.

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.

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

Listing 1.3: Aliasing problem.

Consider listing 1.3 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 1.2 illustrates the situation. In the case of unique, there is only one reference which grants full read/write access.

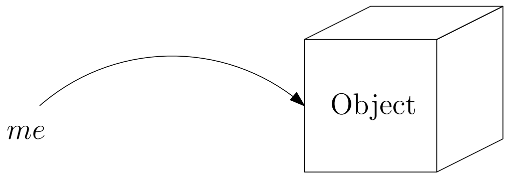

Figure 1.2: Unique access permission.

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 1.3, immutable does not restrict the number of immutable references to an object that can exist at the same time.

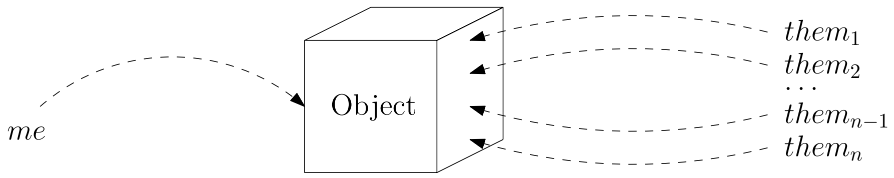

Figure 1.3: Immutable access permission.

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 1.4 shows, shared allows an arbitrary number of other references with or without write access to the object.

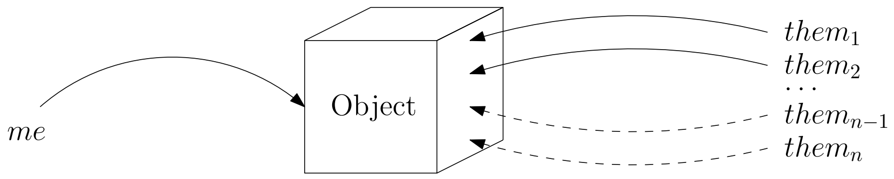

Figure 1.4: Shared access permission.

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 1.5 illustrates, full does not restrict the number of other references but requires all of them to be read-only.

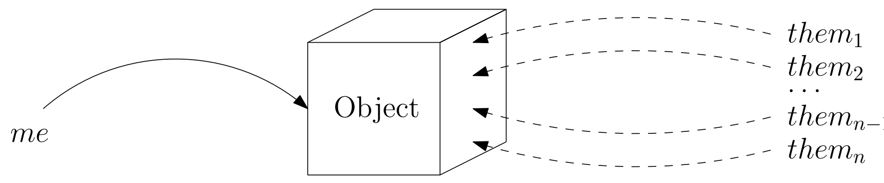

Figure 1.5: Full access permission.

Pure

A *pure* reference [Bie06] to an object provides read-only access to the object and, as shown in figure 1.6, does not restrict the number and quality of other references. It is thus an access permission with very weak guarantees.

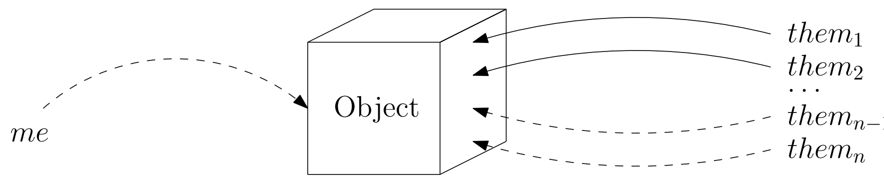

Figure 1.6: Pure access permission.

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 1.1 sums up the properties of the permission types again and additionally gives an overview of the compatibility amongst access permissions.

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 $A \multimap B$ 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 $A \wedge B$ from A and $A \Rightarrow B$ whereas linear logic only allows to conclude B from A and $A \multimap B$ because A is consumed in the process.

Permission kind	This reference	Other references	Compatible permission types
unique	read/write	none	none
full	read/write	read-only	pure, none
shared	read/write	read/write	shared, pure, none
immutable	read-only	read-only	immutable, pure, none
pure	read-only	read/write	full, shared, immutable, pure, none
none	none	read/write	unique, full, shared, immutable, pure, none

Table 1.1: Access permission taxonomy.

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 `unique` permission immediately violates the uniqueness guarantee. Thus, a `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.

```
method void modify(unique Object >> unique Object x);
method void main() {
 val unique Object o = new Object;
 modify(o);
 modify(o);
}
```

Listing 1.4: Method signature in Plaid showing the resource-like nature of access permissions.

In listing 1.4, a method `modify` is defined which takes a `unique` reference to its argument and gives it back after the method body has been executed. This is expressed by the pre-condition `>>` post-condition syntax which specifies the pre-conditions and the post-conditions of the method. In the body of the `main` method, a local variable `o` of type `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 `new` expression clearly is the only reference to the new object. After the initialization, the previously defined `modify` method is called. Following the resource interpretation of access permissions, the call *consumes* the `unique` permission to `o` and hands it to the called method. After `modify` returns, a `unique` permission to `o` is *produced* as specified by `modify`'s method signature. Because the permission is recovered, it is possible to call `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 `>>` sign. Hence, the `modify` method could also have been written `method void modify(unique Object x)`.

Consider a different example, shown in listing 1.5.

```
method void read(immutable Object >> immutable Object x);
method void pushOnStack(immutable Object >> none Object x);
method void main() {
 val unique Object o = new Object;
 pushOnStack(o);
 read(o);
}
```

Listing 1.5: Splitting example.

Here, two methods are defined. `read`, that does not modify its argument, takes an immutable permission and returns an immutable permission. And `pushOnStack` which saves the reference to its argument in a stack data structure.

Passing a unique permission to `o` to `pushOnStack`, as is done in `main`, is intuitively no problem because a unique permission is stronger than an immutable permission. A unique reference guarantees that no other references exist whatsoever, thereby automatically also satisfying 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, `pushOnStack` just returns a none permission as specified by the post condition in the method signature. Hence, the following call to `read` is illegal because `read` requires an immutable permission where only a none permission is available. Intuitively, this program should work because the call to `pushOnStack` does not require a unique permission to `o` at all. If it were somehow possible to convert the unique permission into two 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 *permission splitting*. As demonstrated before, some permissions are intuitively stronger than others; for example unique is stronger than immutable. Instead of passing a unique permission as an immutable in listing 1.5, the unique permission is split into two 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 unique into two unique permissions or a full permission into a full and an immutable.

Figure 1.7 lists some legal permission split rules. The variable Π 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 1.8. The nodes of the flow graph are the operations in the program that consume and produce permissions; for example

$$
\begin{array}{rcl}
\mathsf{unique}(x) & \Rrightarrow & \mathsf{full}(x)\ /\ \mathsf{pure}(x) \\
\mathsf{unique}(x) & \Rrightarrow & \mathsf{immutable}(x)\ /\ \mathsf{immutable}(x) \\
\mathsf{full}(x) & \Rrightarrow & \mathsf{shared}(x)\ /\ \mathsf{shared}(x) \\
\mathsf{full}(x) & \Rrightarrow & \mathsf{shared}(x)\ /\ \mathsf{pure}(x) \\
\mathsf{full}(x) & \Rrightarrow & \mathsf{immutable}(x)\ /\ \mathsf{immutable}(x) \\
\mathsf{immutable}(x) & \Rrightarrow & \mathsf{immutable}(x)\ /\ \mathsf{pure}(x) \\
\mathsf{immutable}(x) & \Rrightarrow & \mathsf{immutable}(x)\ /\ \mathsf{immutable}(x) \\
\mathsf{shared}(x) & \Rrightarrow & \mathsf{shared}(x)\ /\ \mathsf{pure}(x) \\
\mathsf{shared}(x) & \Rrightarrow & \mathsf{shared}(x)\ /\ \mathsf{shared}(x) \\
\mathsf{pure}(x) & \Rrightarrow & \mathsf{pure}(x)\ /\ \mathsf{pure}(x) \\
\Pi(x) & \Rrightarrow & \Pi(x)\ /\ \mathsf{none}(x)
\end{array}
$$

Figure 1.7: Legal permission split rules.

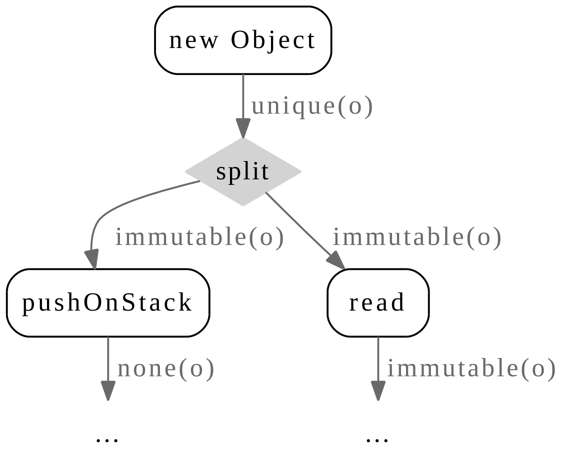

Figure 1.8: Permission flow graph for example from listing 1.5.

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 1.6 gives an example where permission restoration is necessary for the program to compile.

```
method void read(immutable Object >> immutable Object x);
method void modify(unique Object >> unique Object x);

method void main() {
 val unique Object o = new Object;
 read(o);
 modify(o);
}
```

Listing 1.6: Motivation for permission joining.

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 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 1.7. 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.

Figure 1.9 shows legal rules that combine splitting and joining, as indicated by the double arrows. The variable β 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 1.6. The permission flow graph illustrating the splitting and joining of permissions is shown in figure 1.10.

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`.

$$
\begin{aligned}
\textsf{unique}(x,1) &\iff \textsf{full}(x,1/2)\ /\ \textsf{pure}(x,1/2)\\
\textsf{unique}(x,1) &\iff \textsf{immutable}(x,1/2)\ /\ \textsf{immutable}(x,1/2)\\
\textsf{full}(x,\beta) &\iff \textsf{immutable}(x,\beta/2)\ /\ \textsf{immutable}(x,\beta/2)\\
\textsf{immutable}(x,\beta) &\iff \textsf{immutable}(x,\beta/2)\ /\ \textsf{pure}(x,\beta/2)\\
\textsf{immutable}(x,\beta) &\iff \textsf{immutable}(x,\beta/2)\ /\ \textsf{immutable}(x,\beta/2)\\
\textsf{shared}(x,\beta) &\iff \textsf{shared}(x,\beta/2)\ /\ \textsf{pure}(x,\beta/2)\\
\textsf{shared}(x,\beta) &\iff \textsf{shared}(x,\beta/2)\ /\ \textsf{shared}(x,\beta/2)\\
\textsf{pure}(x,\beta) &\iff \textsf{pure}(x,\beta/2)\ /\ \textsf{pure}(x,\beta/2)\\
\Pi(x,\beta) &\iff \Pi(x,\beta)\ /\ \textsf{none}(x,0)
\end{aligned}
$$

Figure 1.9: Legal rules for fractional permissions combining splitting and joining.

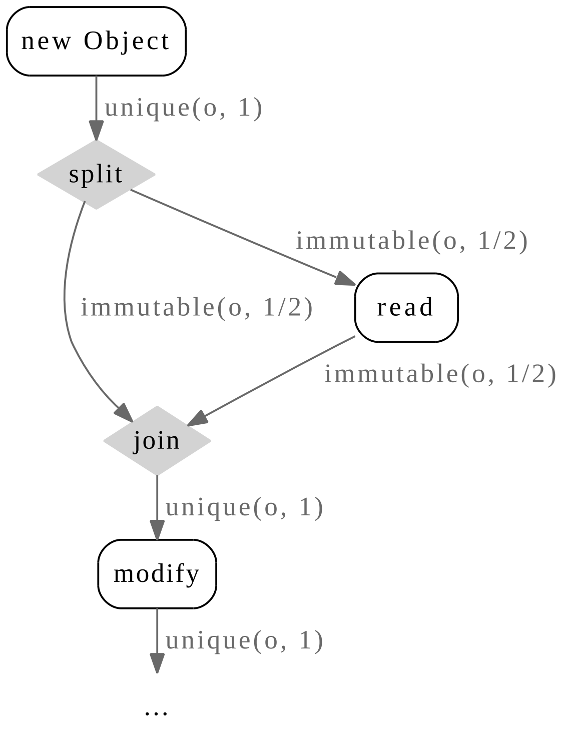

Figure 1.10: Permission flow graph demonstrating fractional permissions.

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 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.

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.

```
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 1.7: Gradual typing in Plaid.

Listing 1.7 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.

1.4 Æminium

The idea of the Æminium 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 typestate of objects but to enable the programmer to express dependencies between operations.

Although Æminium's ideas represent the foundations of one of the key features of Plaid, it is an independent project from Plaid. The Æminium approach can

[2]Actually, even `dyn` can be omitted here. The type is automatically assumed to be `dyn` in this case.

be applied to any programming language that builds access permissions into the language. As shown in figure 1.11, Æminium's general design includes two ma-

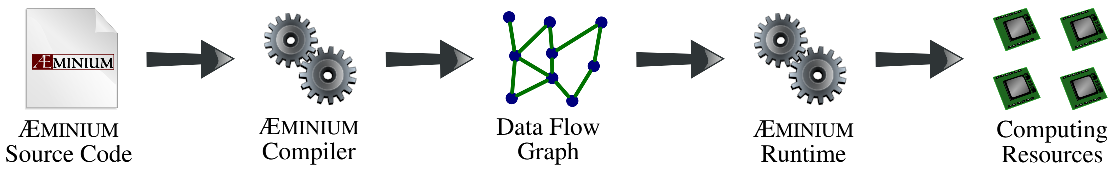

Figure 1.11: Æminium pipeline.

jor 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. Æminium's runtime system is discussed in further detail in section **??**.

1.4.1 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 races*. 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.

1.4.2 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 1.8.

```
method unique Histogram computeRedHistogram(immutable RGBImage m);
method unique Histogram computeGreenHistogram(immutable RGBImage m);
method unique Histogram computeBlueHistogram(immutable RGBImage m);
method void equalize(unique RGBImage m, immutable Histogram r,
                     immutable Histogram g, immutable Histogram b);

method unique RGBImage histogramEqualization(unique RGBImage m) {
 val r = computeRedHistogram(m);
 val g = computeGreenHistogram(m);
 val b = computeBlueHistogram(m);

 equalize(m, r, g, b);
}
```

Listing 1.8: 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 1.8 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 1.12. 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 1.12 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 *in parallel*. For example, it could generate code that assigns each execution of a method call to a different thread. In terms of the 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.

1.4.3 Shared Permissions

While working with unique and 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

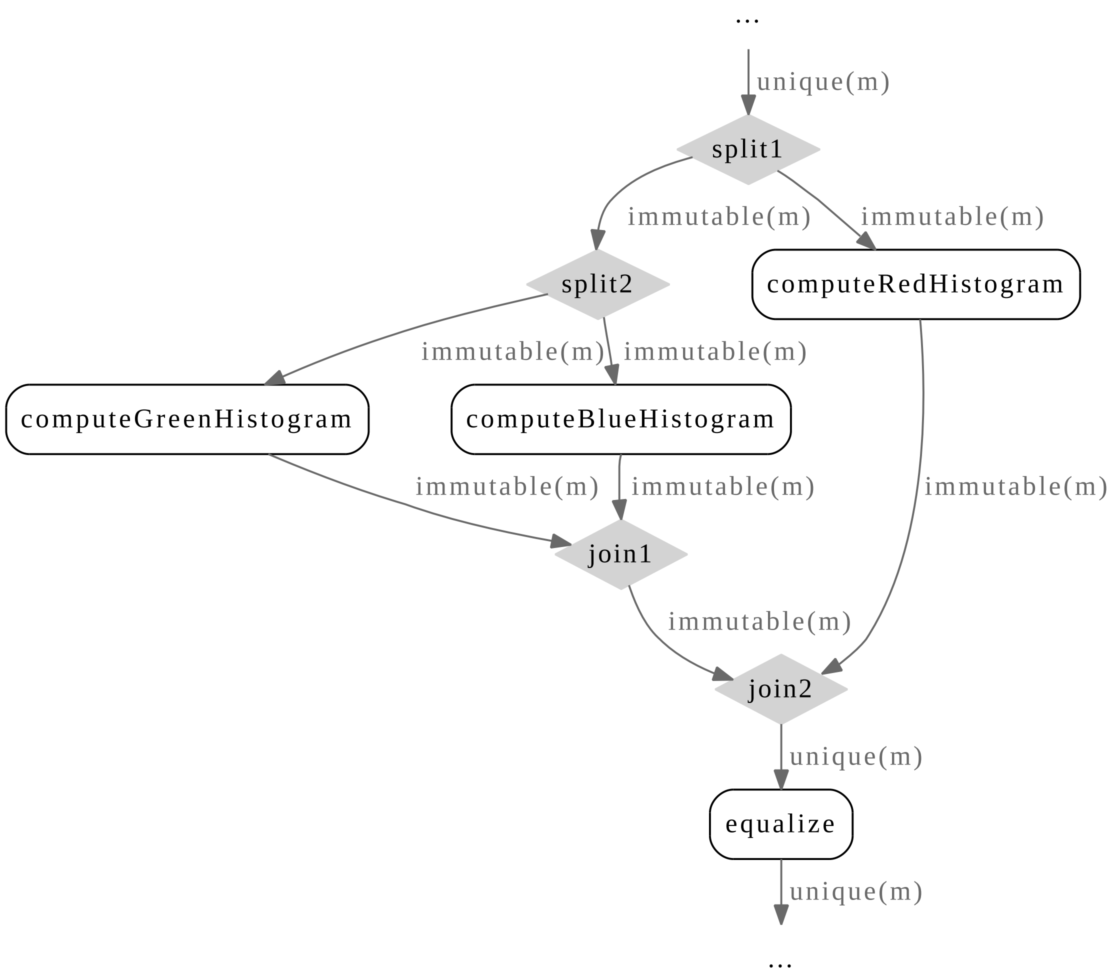

Figure 1.12: Permission flow inside the `histogramEqualization` method.

exactly two other list nodes. Hence, those references cannot be unique references because they point to the same object. But if they are 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. Æminium uses shared permissions to allow the modeling of *shared state*.

As discussed in section 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.

```
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 1.9: Example use of the `atomic` block statement.

Listing 1.9 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.

1.4.4 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 1.10.

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

Listing 1.10: 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 unsychronized 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.

```
// exclusive permission to G
share (G) {
 // shared permission to G
 atomic (G) {
  // protected permission to G
 }
 // shared permission to G
}
// exclusive permission to G
```

Listing 1.11: 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 1.12 illustrates such a situation when the non-existence of data groups is assumed.

```
method void register(shared Subject s, unique Observer >> none Observer o);
method void update(shared Subject s);

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

 register(s, o1);
 register(s, o2);
 update(s);
 update(s);
}
```

Listing 1.12: 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 object 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 1.13. 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.

```
share (G) {
 register(s, o1);
 register(s, o2);
}
share (G) {
 update(s);
 update(s);
}
```

Listing 1.13: Concurrent observer with data groups.

Figure 1.13 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.

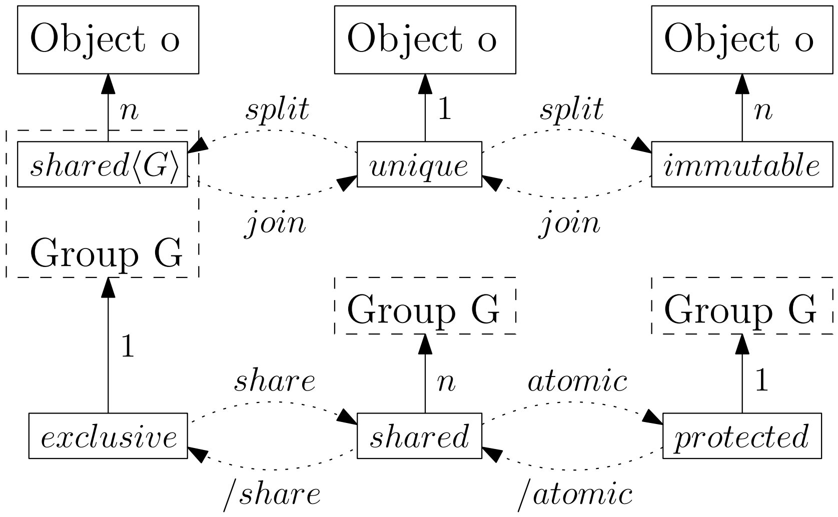

Figure 1.13: Different types of permissions in Æminium.

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

Bibliography

[ASSS09] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM.

[BA07] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *SIGPLAN Not.*, 42:301–320, October 2007.

[Bie06] Kevin Bierhoff. Iterator specification with typestates. In *Proceedings of the 2006 conference on Specification and verification of component-based systems*, SAVCBS '06, pages 79–82, New York, NY, USA, 2006. ACM.

[Boy03] John Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[DF04] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, ECOOP '04, pages 465–490, Berlin, Heidelberg, 2004. Springer-Verlag.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, January 1987.

[Lei98] K. Rustan M. Leino. Data groups: specifying the modification of extended state. *SIGPLAN Not.*, 33:144–153, October 1998.

[PG10a] The Plaid Group. The Plaid language: Dynamic core specification. `http://plaid-lang.googlecode.com/hg/docs/spec/current/current.pdf`, June 2010.

[PG10b] The Plaid Group. The Plaid programming language. `http://www.plaid-lang.org`, 2010.

[SAM10] Sven Stork, Jonathan Aldrich, and Paulo Marques. μAEminium language specification. Technical Report CMU-ISR-10-125R, Carnegie Mellon University, October 2010.

[Sie] Jeremy Siek. What is gradual typing? `http://ecee.colorado.edu/~siek/gradualtyping.html`.

[SMA09] Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 933–940, New York, NY, USA, 2009. ACM.

[SY86] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12:157–171, January 1986.