Concurrency by Default

Using Permissions to Express Dataflow in Stateful Programs

Sven Stork[†*] Paulo Marques[*] Jonathan Aldrich[†]

[†]Institute for Software Research
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
{svens, jonathan.aldrich}@cs.cmu.edu

[*]CISUC, Dep. Eng. Informática, Polo II
University of Coimbra
3030-290 Coimbra, Portugal
{stork, pmarques}@dei.uc.pt

Abstract

The rise of the multicore era is catapulting concurrency into mainstream programming. Current programming paradigms build in sequentiality, and as a result, concurrency support in those languages forces programmers into low-level reasoning about execution order.

In this paper, we introduce a new programming paradigm in which concurrency is the default. Our ÆMINIUM language uses access permissions to express logical dependencies in the code at a higher level of abstraction than sequential order. Therefore compiler/runtime-system can leverage that dependency information to allow concurrent execution.

Because in ÆMINIUM programmers specify dependencies rather than control flow, there is no need to engage in difficult, error-prone, and low-level reasoning about execution order or thread interleavings. Developers can instead focus on the design of the program, and benefit as the runtime automatically extracts the concurrency inherent in their design.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]; D.1.3 [*Concurrent Programming*]

General Terms Design, Languages

Keywords concurrency, programming language, access permissions, dataflow

1. Introduction

"The free lunch is over" [Sutter 2005] characterizes like no other statement one of the most fundamental technology shifts in the last few decades. Because it is no longer feasible to improve single CPU performance, hardware vendors started to integrate multiple cores into single chip. This means that programmers need to develop concurrent applications if they want to achieve performance improvements on new hardware. Writing concurrent applications is notoriously complicated and error prone, because concurrent tasks must be coordinated to avoid problems like race conditions or deadlocks.

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

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

To overcome this situation, we propose to transform *implicit* dependencies into *explicit* dependencies and then in-

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

fer the ordering constraints automatically. In our proposed system, by default, everything is concurrent, unless explicit dependencies imply a specific ordering.

We propose to use *access permissions* [Bierhoff and Aldrich 2007] to specify explicit dependencies between stateful operations. Access permissions are abstract capabilities that grant or prohibit certain kinds of accesses to specific state. In our approach each method need to specify permissions to all of the state it potentially accesses. Looked at from a slightly different perspective, our system ensures that every method only accesses state for which it has explicit permissions. The way we use access permissions to specify state dependencies resembles the way Haskell [Jones 2003] uses its I/O monad[1] to specify access to global state. But unlike the I/O monad, which provides just one permission to all the state in the system, access permissions allow greater flexibility by supporting fine-grained specifications, describing the exact state and permitted operations on it.

Following our new programming paradigm, *concurrency-by-default*, we are currently working on the semantics and implementation of ÆMINIUM. In this paper we use a Java-like syntax[2] to explain the concepts behind ÆMINIUM and its features.

The rest of this paper is organized as follows. Section 2 discusses the core language features of ÆMINIUM. Section 3 presents some currently open challenges. In section 4 we discuss related work. Section 5 concludes the paper.

2. Concurrency by Default

In ÆMINIUM every method must explicitly mention all of its possible side effects. This allows the system to compute the data dependencies within the code, and within those constraints, execute the program with the maximum possible concurrency. By following this approach our system resembles a *dataflow architecture* [Rumbaugh 1975]. But instead of producing and consuming data, our system supports shared objects and in-place updates.

To achieve scalability for upcoming massive concurrent systems, we need to use a fine-grained approach for specifying side effects. To avoid overly conservative dependencies, which would limit concurrency, we need a way to deal with object *aliasing*. In *access permissions* [Bierhoff and Aldrich 2007] we found a uniform solution for both problems, the specification of data accesses and the specification of aliasing. The next sections describe the approach in more detail.

2.1 Access Permissions for Concurrency

2.1.1 Unique and Immutable Permissions

Consider the application in Figure 1 which computes over a collection of data. Starting with line 20 the main function creates a collection containing some randomly generated data. At line 22 we print the collection on the screen, then pass the collection into method calls to compute statistics and dependencies over the passed collection, and return corresponding objects describing the statistics and dependencies. We will assume these objects are used later on in the method body, in line 27 and beyond. After that, we remove existing duplicates from the collection and then print the updated collection to the screen (line 26).

```
1  class Collection { ... }
2  class Dependencies { ... }
3  class Statistics { ... }
4
5  Collection createRandomData()
6   : unit ⤇ unique(result)
7
8  void removeDuplicates(Collection c)
9   : unique(c) ⤇ unique(c)
10
11 void printCollection(Collection c)
12  : immutable(c) ⤇ immutable(c)
13
14 Dependencies compDeps(Connection c)
15  : immutable(c) ⤇ immutable(c),unique(result)
16
17 Statistics compStats(Connection c)
18  : immutable(c) ⤇ immutable(c),unique(result)
19
20 void main() {
21   Collection c = createRandomData()
22   printCollection(c)
23   Statistics s = compStats(c)
24   Dependencies d = compDeps(c)
25   removeDuplicates(c)
26   printCollection(c)
27   ...
28 }
```

Figure 1. Example: Unique and Immutable Permissions

Obviously, for concurrency purposes functions like `removeDuplicates` require a permission to modify the collection. On the other hand, functions like `printCollection`, which only examines the collection, only require a read-only permission. Access permissions allow us to specify exactly these requirements.

Access permissions are abstract capabilities that grant or prohibit certain kinds of accesses to specific state. Access permissions are associated with object references and specify in which way the owner of the permission is allowed to access/modify the referenced object. In our system we use the following kinds of access permissions:

Unique A unique permission to a reference guarantees that this reference is the only reference to the object at this moment in time. Therefore the owner has exclusive access to the object.

Immutable An immutable permission to a reference provides non-modifying access to the referenced object. Additionally a immutable permission guarantees that all other existing references to the referenced object are also immutable permissions.

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

[2] As everything is concurrent by default, we omit the sequentializing semicolon to emphasize this fact.

Shared A shared permission to a reference provides modifying access to the corresponding object. Additionally a shared permission indicates that there are potentially other shared permissions (aliases) to the referenced object through which the referenced object can be modified.

For brevity we write 'unique reference' when we mean 'a unique permission to a reference', as well as for immutable and shared permissions. When specifying permissions in code we write '**unique**(X)' when we mean that we have a unique permission to reference X. We use the pseudo-reference '`result`' to specify a permission to the return value.

We use *linear logic* [Girard 1987] to manage the access permissions in our system. Linear logic is a sub-structural logic for reasoning about resources. Once resources have been consumed they are no longer available. We use the symbol $\Rrightarrow$ to separate the pre-conditions (the permissions a method requires and consumes) from the post conditions (the permissions a method returns). Consider the following method signature : '**unique**(`this`) $\Rrightarrow$ **unique**(`this`)'. In this case the method requires that the caller must have a unique permission to the receiver object to call this method. Because we use linear logic, the input permission is consumed, and therefore the method has to produce a new unique permission to the receiver object upon its return. If the method did not return a permission to the receiver object, the caller would not be able to access the object any more.

Because access permissions play such an important role in our system, we promote them to first-class citizens and integrate them into our type system. Consider the following function that converts an Integer into its String representation, indicating the type (in this case I for Integer) and the value:

```
String repr(Integer a){ return "I"+a; }
```

In a standard ML-style type signature, this function would have the type '`Integer` $\rightarrow$ `String`', stating that the method takes an Integer as input and returns an String. In our system, the same function would have the following access permission signature:

immutable(a) $\Rrightarrow$ **immutable**(a), **unique**(result)

The access permission signature provides much more information regarding the behavior of the function. First, the immutable permission indicates that the function is not going to change the object we passed in. Secondly, indicated by the unique permission, we know that the reference to the returned String object is not aliased, because it is the only one in the whole system.

With this information we are now able to specify the exact permissions of each presented method. As shown in line 5, the `createRandomData` method requires no permissions (we indicated the empty set of permissions with `unit`) and produces a unique permission to the returned collection. Because `printCollection`[3] (line 11), `compDeps` (line 14) and `compStats` (line 14) do not modify the collection, they all just require an immutable permission to the collection, which is returned again after their completion. Additionally, `compStats` and `compDeps` return a unique permission to their returned objects, which are later needed, but are not important in the code shown. The `removeDuplicates` method requires, and returns after completion, a unique permission to the collection, as it is going to modify the collection.

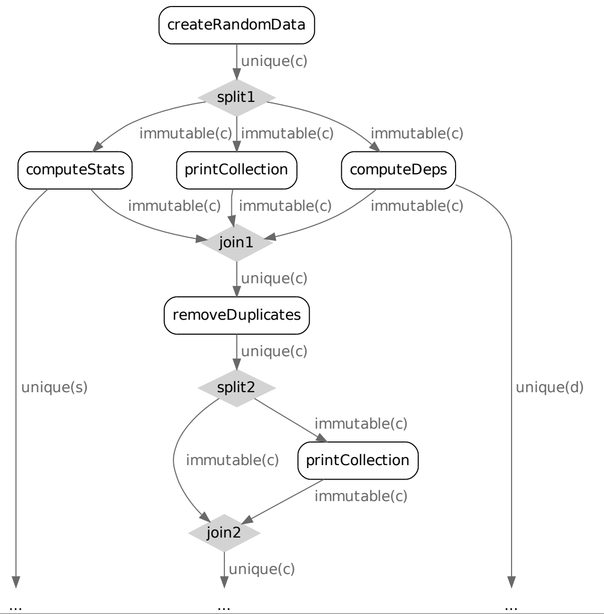

Figure 2. Example: Unique and Immutable Permissions Flow

Given the permission signatures and using textual order, our system is able to compute the permission flow through the program. Figure 2 shows the permission flow graph for the program which captures the existing data dependencies.

As specified in Figure 1, the `createRandomData` method generates a unique permission to the returned collection. The `printCollection`, `compStats` and `compDeps` functions require only an immutable permission to the collection. Therefore our system has to 'convert' the unique permission into three immutable permissions, one for each function. Like in Bierhoff's system, our system performs those 'conversions' by automatically splitting and joining permissions utilizing *fractions* [Boyland 2003]. This means after starting out with a unique permission, the system is able to split the unique permission into either multiple shared permissions or multiple immutable permissions. Remember that because of linearity, the unique permission is consumed and no longer available. The reverse works in a similar way. Once all shared or immutable fractional permissions have been collected, the systems is able to form a unique permission again.

[3] For accessing the output device our system also requires a permission. To keep the example simple and because data groups (explained in section 2.2) offer a better abstraction for dealing with this kind of problems, we omit the I/O-related permissions in this example.

The splitting of the unique permission into three immutable permissions is shown in Figure 2 as 'split1'. Once their input requirements are fulfilled via an immutable permission to the collection, those three methods are eligible for execution. The system can decide to execute them concurrently or sequentially, depending on available resources and relative execution costs.

The `removeDuplicates` method requires a unique permission to the collection, and therefore it depends on the completion of the `printCollection`, `compDeps` and `compStats` methods. Only when those methods complete will they return the immutable permissions to the collection, which they consumed when starting their execution. The system needs to collect all immutable permissions to the collection before it can join them back to a unique permission to the collection (see Figure 2, 'join1'). Remember that immutable guarantees that, at this point in time, there are only immutable permissions referencing the object. After the unique permission has been recovered, the input requirements for the `removeDuplicates` method is fulfilled and it can be executed. The second `printCollection` method (line 26) requires an immutable permission. Therefore, this method depends on the completion of the `removeDuplicates` method, before the system can split the returned unique permission to the collection into immutable permissions to the collection (see Figure 2, 'split2'). After the completion of the second `printCollection` method the system will automatically recover the unique permission to the collection[4] (see Figure 2, 'join2').

The advantage of this approach over explicit concurrency management is founded in the automation of dependency inference and the guarantee that those dependencies are met. If the programmer manages concurrency manually he might overlook dependencies and create race conditions or might overlook the absence of dependencies and miss available concurrency. In particular when it comes to the concurrent sharing of data, reasoning about dependencies becomes significantly more complicated.

2.1.2 Shared Permissions

In the previous section we saw how we can use unique and immutable permissions to extract concurrency. However having only unique and immutable is of limited use. Because there exists only one unique permission to an object at a time, there can be only one entity modifying the object at a time. Shared memory and objects are in general used as communication channels between several concurrent entities, which may modify the shared state. Therefore, we need a mechanism to allow concurrent execution and modifying access to a shared resource. A *shared permission* provides exactly these semantics.

[4] Assuming that the statement that depends next on the collection requires unique permission.

```
1  class Queue {
2    void enqueue (Object o)
3      : unique(this), shared(o) ⤇ unique(this)
4
5    Object dequeue()
6      : unique(this) ⤇ unique(this), shared(result)
7  }
8
9  Queue createQueue() : unit ⤇ unique(result)
10
11 void disposeQueue(Queue q) : unique(q) ⤇ unit
12
13 void producer(Queue q) : shared(q) ⤇ shared(q)
14 { atomic { q.enqueue(...) ... } }
15
16 void consumer(Queue q) : shared(q) ⤇ shared(q)
17 { atomic { Object o = q.deqeueu() ... } }
18
19 void main() {
20   Queue q = createQueue()
21   producer(q)
22   consumer(q)
23   disposeQueue(q)
24 }
```

Figure 3. Example: Producer/Consumer with Shared Permissions

As explained before, a shared permission allows modifying access to the referenced object and indicates that the there are potentially other shared references out there, through which the referenced object could be changed. In our system, similar to immutable permissions, statements that depend on the same shared object can be executed concurrently. Obviously, allowing concurrent access to the same object opens the window for race conditions. Therefore we require that every access through a shared reference must occur inside an atomic context. We introduce the *atomic-block* statement into our language, `atomic { ... }`, with the common *transactional memory* [Larus and Rajwar 2007] semantics. In particular this means that a block of statements is completely executed, and all modifications become visible to the rest of the system atomically. It is important to note that all code inside an atomic context is sequentially executed in the given lexical order. If several different atomic blocks cause conflicting accesses, the runtime system will detect those and resolve them (in general by aborting, rolling back and retrying some of the atomic blocks). Therefore, an atomic block provides the illusion of having exclusive access to the all accessed resources. Although the placement of atomic blocks could be inferred automatically, for granularity reasons, we require the user to explicit specify atomic regions. This approach allows the user to have fine-grain control over the size of critical sections, while our system can adapt the approach described in [Beckman et al. 2008] to verify and enforce the correct usage of atomic blocks.

Figure 3 shows a simplified producer/consumer example, where the producer and consumer communicate via a queue. Beginning in line 19 main calls `createQueue` to obtain a new queue object. This queue is then passed to the `pro-`

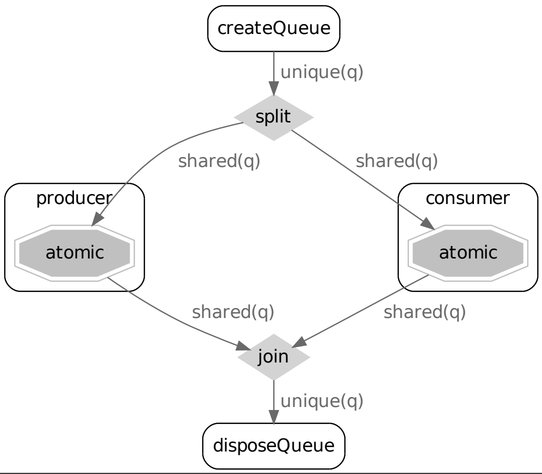

Figure 4. Example: Producer/Consumer with Shared Permission Flow

ducer and consumer methods (lines 21 + 22). Finally the program calls the disposeQueue method to free the queue.

This program's permission flow is shown in Figure 4. Both the consumer and producer methods require a shared permission to the queue. Therefore, the unique permission returned by createQueue (line 9) is automatically split by the system into shared permissions (Figure 4, 'split'). This means that both the producer and consumer methods have their required input permissions and can be executed in parallel. Because the queue is shared, both methods need to be in an atomic context when accessing the queue (lines 14 + 17). As shown in line 2 and 5, both the enqueue and dequeue methods require a unique permission to the queue. Because the atomic block provides an illusion of exclusive access, we can treat the shared permission to the queue as a unique permission, and permit the access to the queue. Because disposeQueue requires a unique permission to the queue, it depends on the eventual completion of producer and consumer to return the shared permissions to the queue and join them back to form a unique permission (Figure 4, 'join').

2.2 Data Groups for Higher-Level Dependencies

In some situations, application level dependencies exist that cannot directly inferred via data dependencies. As an Example of high-level dependencies, consider the common observer pattern. It is unclear whether the observers of a subject need to be attached to the subject before the subject can be updated. In some situations it is important for observers not miss the first update (e.g., to initialize the observer correctly), while in other situations it does not matter if the first update is missed (e.g., a news feed). We propose to use *data groups* [Leino 1998] to allow the specification of such high-level dependencies.

Consider the simple observer example shown in Figure 5. The program creates a new subject which is then passed to newly created observers and to several update method

```
1  class Subject {
2    void add(Observer o)
3      : shared(this), shared(o) ⤇ shared(this)
4
5    void update() : shared(this) ⤇ shared(this)
6  }
7
8  class Observer {
9    Observer(Subject s)
10     : shared(s) ⤇ shared(s), shared(result)
11   { s.add(this) }
12
13   void notify(Subject s)
14     : shared(this), shared(s) ⤇ shared(this), shared(s)
15 }
16
17 void update(Subject s) : shared(s) ⤇ shared(s)
18 { s.update() }
19
20 void main() {
21   Subject s = new Subject()
22   Observer obs1 = new Observer(s)
23   Observer obs2 = new Observer(s)
24   update(s)
25   update(s)
26   ...
27 }
```

Figure 5. Example: Concurrent Observer

calls. The observer constructor simply adds the current object as subscriber to the provided subject (line 11). The update call triggers the notification of the subject (line 18). Furthermore assume we want to extract the maximum parallelism possible by allowing the concurrent creating/addition of observers and concurrent updates. A first attempt would be to use shared permissions to the subject in the Observer constructor call (line 9) and the update call (line 18). Using this approach leads to the dependencies shown in Figure 6. The problem is that, as shown, the construction of the Observer objects and the update function only have dependencies with the Subject but not amongst each other. Therefore they can be executed concurrently in any order. This could lead to the update method being called before any Observer is attached to the subject. While this behavior might, in some scenarios, be acceptable (e.g., a small gadget that display the latest news), it can also be completely unacceptable in other situations (e.g., when the observer depends on the initial values of the subject). One way to ensure that the observers have been attached before the update calls get executed is to change the Observer constructor to require a unique permission to the subject. But this also creates a problem since it would limit parallelism, as all Observer object constructions would be serialized.

To allow the user to specify such additional dependencies without sacrificing concurrency, we add data groups to our system. Data groups are abstract collections of objects. In particular an object can be associated with exactly one data group at a time. Data groups provide a higher-level abstraction and provide *information hiding* with respect to what state is touched by a method.

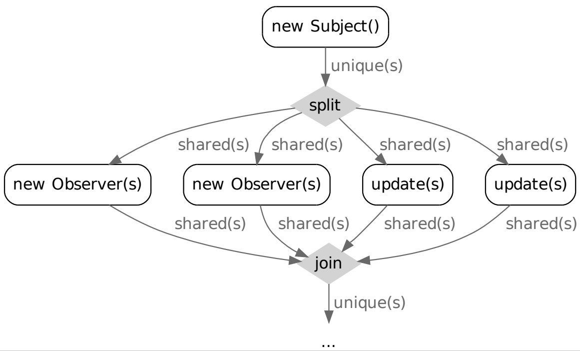

Figure 6. Example: Concurrent Observer Flow

In our system a data group can be seen as a container which contains all shared permissions to an object. Since unique permissions already provide exclusive access to the referenced object and immutable permissions can safely be shared, we do not associate unique and immutable permission with data groups. Therefore unique can be used to transfer an object between data groups. We extend the definition of access permissions to optionally refer to the associated data group. We write ‘**shared**(*REF*|*DG*)’, where *REF* is the object reference and *DG* specifies the data group. Similar to access permissions for objects, we introduce access permissions to data groups:

atomic An atomic permission provides exclusive access to a data group. Working on an atomic data group automatically leads to the sequentializing the corresponding code. This is similar to a unique permission for objects. Requiring an atomic permission must be explicitly specified.

concurrent A concurrent permission to a data group means that multiple other concurrent permissions to the data group exist. Code working on a concurrent data groups is executed with concurrency by default. This is similar to a shared permission for objects. Concurrent permission is the default, so using the concurrent keyword is optional.

Unlike with access permissions to objects, the user must manually split and join permissions to data groups. To avoid tedious and error prone management of permissions for data groups, we propose a *split block* construct. A split block converts a unique permission to its data group into an arbitrarily number of concurrent permissions that may be used in its body block. Having concurrent permissions inside the body block of the data group allows the body to be executed concurrently. After the execution of its body block, the split block will join all concurrent permissions back to a unique permission :

split (DataGroup grp) { ... }

Additional we propose the enhancement of the atomic block construct to refer to the data group of the objects that are going to be modified :

atomic (DataGroup grp) { ... }

The explicit specification of data groups is optional as it can be automatically inferred from the code in the atomic block's body. Nevertheless, when present, it can be used to verify the body against the explicit specification. Having the explicit knowledge of which data groups are accessed inside and atomic block could allow optimizations of the transactional memory system or its complete replacement via a more lightweight approach [Boehm 2009].

```
1  class Subject<SG> {
2    void add(Observer<SG> o)
3      : shared(this|SG), shared(o|SG) ⇛ shared(this|SG)
4
5    void update()
6      : shared(this|SG) ⇛ shared(this|SG)
7  }
8
9  class Observer<SG> {
10   Observer(Subject<SG> s)
11    : shared(s|SG) ⇛ shared(s|SG), shared(result|SG)
12   { s.add(this) }
13
14   void notify(Subject<SG> s)
15    : shared(this|SG), shared(s|SG)
16        ⇛ shared(this|SG), shared(s|SG)
17 }
18
19 void update(Subject<SG> s)
20    : shared(s|SG) ⇛ shared(s|SG)
21 { s.update() }
22
23 void main() {
24   group <SubG>
25
26   split (SubG) {
27     Subject<SubG> s = new Subject<SubG>()
28     Observer<SubG> obs1 = new Observer<SubG>(s)
29     Observer<SubG> obs2 = new Observer<SubG>(s)
30   }
31   split (SubG) {
32     update<SubG>(s)
33     update<SubG>(s)
34   }
35   ...
36 }
```

Figure 7. Example: Concurrent Observer with Data Groups

Figure 7 shows the observers example using the data group approach. We use a syntax similar to type parameters to specify and pass data groups around. A group parameter can be used at the class level (line 1) or the function level (line 19). The ‘**group**<*Z*>’ command creates a new group with the name *Z*. The group command always returns an atomic permission to the new group.

In the enhanced example, in line 24, a new data group with the name ‘SubG’ is created. In line 26 the ‘split’ block is used to split the atomic permission of the ‘SubG’ data group into an arbitrary number of concurrent permissions. Having a concurrent permission reestablishes a concurrent-by-default environment. Thus, the statements in the body block may be executed concurrently up to explicit data dependencies. This is shown in Figure 8. The second ‘split’ block (line 31) requires an atomic permission to the ‘SubG’ data group and therefore depends on the completion of the

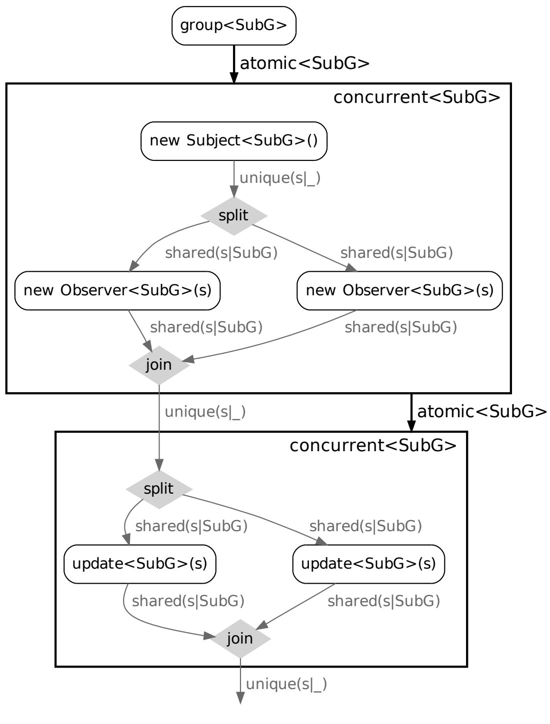

Figure 8. Example: Concurrent Observer with Data groups Flow

first split block. After completion of the first split block's body, all the concurrent permissions to the 'SubG' group can be gathered and joined back into an atomic permission.

The dependencies between data groups and data dependencies are visualized in Figure 8. The atomic group permission, generated by the group command, will be split by the first split block (first rectangle) into concurrent group permissions. The statements inside the corresponding block follow the normal data dependency mechanism. The system will automatically split the unique permission of the subject into shared permissions, to allow the concurrent execution of the `Observer` creation. After the completion of the block, the system will join the shared permissions back into a unique permission and the split block will join the concurrent group permissions back into an atomic permission. The second split block (second rectangle) will take the atomic group permission generated by the first split block and split it again into concurrent permissions for its body. Inside the body, the normal approach of automatically splitting and joining object permissions is then performed.

The advantage of using data groups over explicit concurrency management is again based on automatic dependency inference and the guarantee that those dependencies are met. Data groups allow the programmer to explicitly model her design intent in the source code. Not only does this allow the ÆMINIUM system to infer the dependencies and correct execution, it also improves the quality of the code itself by explicit documenting those dependencies.

3. Challenges

So far we have only presented a high-level overview of ÆMINIUM. Since it is at an early stage of development there are still several open issues to solve. In particular, the following questions deserve closer attention:

Overhead An open question is how much specification overhead does our approach cause for the developer. Permissions are modular and should be automatically inferable most of the time. But data groups model an *effect system*, and it may be a challenge to declare function effects without creating a blowup in specification size.

Granularity Since we target commodity hardware, we have to find a good trade-off between the very fine granularity of parallelism our system is able to extract and the execution/synchronization overhead. One possible way to tackle this problem could be by adapting a cost semantic model as developed by [Spoonhower 2009].

Runtime-System We need to find an efficient way to represent code along with its data dependencies in an intermediate format that allows efficient execution. Also, taking the granularity argument into account, we most likely need to develop a dynamic runtime system that automatically adapts the program to the hardware platform.

Legacy Code When designing a new language, one cannot ignore the vast amount of legacy code that exists. We propose to integrate legacy code, which has no permissions, by assuming the most restrictive permission type. This effectively sequentialises the execution of those code fragments but allows a semantically correct usage of legacy code in our system.

Deadlock Our system avoids race conditions, but does not protect against deadlocks. For instance, it is known that using an atomic block at the wrong granularity can lead to deadlock [Martin et al. 2006].

4. Related Work

As discussed in Section 1, ÆMINIUM was inspired by wanting to realize the concurrency benefits of functional programming in an imperative setting. Therefore, all functional programming languages can be seen as related work. In particular *Haskell* [Jones 2003], with is monad system, relates closely to our system.

Greenhouse [Greenhouse and Scherlis 2002] describes an annotation and policy system for specifying relationships between locks and state in systems with explicit concurrency. In Greenhouse's system state can be grouped into *regions* and locks can be associated with state or regions of state. While Greenhouse uses data groups to show the absence of race conditions, our approach uses data groups to infer possible correct orders of execution. Our use of data groups is also similar to *ownership* systems [Clarke et al. 1998].

Boyland [Boyland 2003] presented a system that uses 'read' and 'write' permissions to automatically infer dependencies between operations. Its goal was to verify the correctness of already explicitly parallelized programs. Our approach reverses this scenario: we use permissions for extracting concurrency based on the inferred dependencies. Additionally, our system supports shared permissions.

Among recently developed programming languages, *Fortress* [Allen et al. 2008] is the most comparable to our concurrency-by-default paradigm. Fortress changes the semantics of certain programming constructs, like tuple constructors or for-loops, to be concurrent by default. Like our system, Fortress takes advantage of the high-level atomic block primitive to synchronize. Unlike our system, Fortress does not infer data dependencies or enforce the correct usage of atomic blocks, and therefore it has no built-in protection against data races.

Another related concurrent programming language is *Cilk* [Blumofe et al. 1995]. Cilk extends C with three additional keywords: `cilk`, `spawn` and `sync`. Every method annotated with `cilk` can be asynchronously spawned-off with the `spawn` keyword. The `sync` keyword is used to wait for a previously started asynchronous task to complete. The Cilk runtime implements a highly effective work stealing mechanism to achieve high performance. Like Fortress, Cilk does not provide any build-in protection against race conditions or support for correct synchronization of shared resources like ÆMINIUM does.

Axum (formerly known as *Maestro*) [Mic 2009] is an actor-based programming language. Axum comes with several operators to allow the explicit construction of dataflow graphs, which can hierarchically composed. For efficiency reasons, Axum also provides *domains*, containers for state, which allows associated actors to access the enclosed state. Actors can either be readers or writers of shared state and scheduling will follow the one writer or multiple reader model. Many concepts in Axum and ÆMINIUM look similar, in particular the dataflow approach, and the use of data-groups/domains combined with the explicit specification of accesses. But ÆMINIUM focuses on object-oriented programming, automatically infers the dataflow graph and supports true shared state between concurrent entities.

5. Conclusion

We presented ÆMINIUM, a novel programming language for highly concurrent systems, that uses access permissions and data groups to make side effects explicit. In ÆMINIUM everything is concurrent by default and concurrent execution is solely limited by explicit and automatically-inferred dependencies. ÆMINIUM requires only local reasoning about side effects and handles dependency inference and concurrent execution automatically. Therefore we believe that ÆMINIUM, by following our concurrent-by-default paradigm, represents a major step towards programming highly concurrent systems.

Future work will focus on the semantics of the system, the implementation of an efficient runtime system and investigation of practical solutions to the granularity problem.

Acknowledgments

This work was partially supported by the Portuguese Research Agency – FCT, through a scholarship (SFRH / BD / 33522 / 2008), CISUC (R&D Unit 326/97), the CMU—Portugal program, DARPA grant #HR0011-0710019, NSF grants CCF-0546550 and CCF-0811592, and Army Research Office grant number DAAD19-02-1-0389 entitled “Perpetually Available and Secure Information Systems.”

References

E. Allen, D. Chase, J. Hallett, V. Luchangco, J.W. Maessen, S. Ryu, G.L. Steele Jr, and S. Tobin-Hochstadt. The Fortress language specification version 1.0. Technical report, Sun Microsystems, Inc, 2008.

N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. *Proc. ACM SIGPLAN conference on OOPSLA*, 43(10):227–244, 2008.

K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proc. ACM SIGPLAN conference on OOPSLA*, pages 301–320, 2007.

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *Proc. ACM SIGPLAN symposium on PPoPP*, 30(8): 207–216, 1995.

H.-J. Boehm. Transactional Memory Should Be an Implementation Technique, Not a Programming Interface. Technical Report HPL-2009-45, HP Laboratories, 2009.

J. Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72. Springer, 2003.

D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *Proc. ACM SIGPLAN conference on OOPSLA*, 33(10):48–64, 1998.

J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *Proc. ICSE*, pages 453–463, New York, NY, USA, 2002. ACM.

S.L.P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 1 edition, 2007.

K. Rustan M. Leino. Data groups: specifying the modification of extended state. In *Proc. ACM SIGPLAN conference on OOPSLA*, pages 144–153, New York, NY, USA, 1998.

M. Martin, C. Blundell, and E. Lewis. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

Axum Programmer's Guide. Microsoft Corporation, 2009. `http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx`.

JE Rumbaugh. *A parallel asynchronous computer architecture for data flow programs*. PhD thesis, Massachusetts Institute of Technology, 1975. MIT-LCS-TR-150.

D. J. Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, May 2009.

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