Ph.D. Research Proposal

Doctoral Program in Information Sciences and Technologies
Software Engineering

Concurrent Programming via Access Permissions

Sven Stork
stork@dei.uc.pt
svens@cs.cmu.edu

Advisor(s):
Paulo Marques
Jonathan Aldrich

25. September 2009
Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

Abstract

The aim of this doctoral thesis is to study implications of having a parallel-by-default programming language. This includes language design, runtime system, performance and software engineering considerations. We hope that the work helps to advance concurrent programming in modern programming environments.

Keywords

programming language, access permission, concurrent programming

Table of Contents

CHAPTER
ONE

Introduction

One of the most fundamental technology shifts in the last few decades is best characterized by “The free lunch is over” [1]. 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 and programs can execute concurrently to the extent permitted by data dependencies. Although functional programming can solve every problem, having explicit state, as provided by imperative languages, allows the developer to express certain problems in a more intuitive and efficient way. In an ideal world we would like to have the concurrent execution benefits of functional programming to with the expressiveness of an imperative object-oriented language.

Sharing state between concurrent tasks immediately raises questions like: *‘In which order should those accesses occur?’* and *‘How can one coordinate those accesses to maintain a program invariants?’* The reason why those questions are hard to answer is because there are *implicit* dependencies between code and state. Methods can arbitrarily change any accessible state without revealing this information to the caller. This means that two methods can be dependent on the same state, without the caller knowing about it. Because of this lack of information, current programming languages use the order in which code is written as proxy to express those implicit dependencies. Therefore the compiler has to follow mostly 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 infer the ordering constraints automatically. In our proposed system, by default, everything is concurrent, unless explicit dependencies imply a specific ordering. By using a concurrent by default approach, we eliminate explicit, and notoriously complicated and error prone, reasoning about sequential and parallel ordering. Instead of specifying when and where which operations/tasks should be executed, the programmer in our approach specifies which stateful effects [1] each operations performs. The system

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

we will use those dependency information to perform the operations in an non-interfering manner. The system will not only use the dependency information to perform concurrent execution, but also validate that the dependency information is consistent. We strongly believe that this approach will provide a significant improvement over current approaches and will lead to fewer concurrent bugs and more scalable software.

We propose to use *access permissions* [2] to specify explicit dependencies between stateful operations. Access permissions are abstract capabilities that grant or prohibit certain kinds of accesses to specific state. Our approach requires each method to specify permission 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 [3] uses its I/O monad[2] to specify access to global state. But unlike the I/O monad, which provides just one permission to all the state in the system, access permissions allow greater flexibility by supporting fine-grained specifications, describing the exact state and permitted operations on it.

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

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

CHAPTER
TWO

State of the Art

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

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

implicit An implicit concurrent system distinguishes itself from an explicit concurrent system, in the fact that it does not require the user to actively write code for concurrent execution. In an implicit system the semantics of the language or library interfaces imply that certain operations could be performed concurrently.

2.1 Explicit Concurrency

Explicit concurrency is all about the manual management of different threads of execution. The most simple, and most coarse grain, form of explicit concurrency is separate, sequential processes which exchange data via a communication channel. A minimalistic formal description of those *communicating sequential processes* (CSP) is given by [4] and is commonly know as the Π-calculus. It is the common case for CSP to communicate via message passing. In message passing all necessary synchronization is implicitly handled by the message passing abstraction and relieves the programmer from explicit managing

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

synchronization via low-level primitives. Because processes have strong isolation between each other, data needs to be copied from one process to another process. This leads to the fact that message passing systems are in general free of race-conditions, but also contributes to its inefficiency in the case of huge amounts of data. In general the support for spawning new processes is not directly included in mainstream programming languages and is rather provided via libraries. Those libraries range from simple wrappers, that call straight into the operating system (e.g. fork), to highly-sophisticated libraries that support complex communication operations and extend infra-structure management support (e.g. MPI).

The *Message Passing Interface* (MPI) [5, 6] is one example for such an sophisticated library. MPI is the established de facto standard for developing high performance distributed memory application. MPI implementations provide besides library itself, several tools for starting and managing multiple processes job. The latest MPI standard [6] also added the support for dynamically creating processes. The MPI standard defines a rich set of communication abstractions, ranging from synchronous and asynchronous point-to-point operations to complex collective operations (e.g. a collective reduce operation with a user-defined data types and operator functions).

Erlang [7] is one of the few programming languages that has built-in support for process creation and communication channel between processes. Erlang processes do not map directly to operating system processes at language level. But Erlang provides a strong isolation guaranty between processes and a high-level communication abstraction, which allows processes to run either on the same virtual machine or on different virtual machines on different nodes. Therefore we consider Erlang a member of the CSP family and inherits all the corresponding features and shortcomings.

Threads are concurrent entities inside a process that shared the address space with their host process. Therefore, threads allow fast and easy shared memory communication. Instead of sending data between processes, and eventually duplicating shared data, data can be uniformly accessed by all threads in the system. One side-effect is that all accesses to shared data need to be coordinated to avoid *race-conditions*. Similar to process management, many older programming languages (like for instance C, C++[2], ...) support threads via external libraries, providing simple wrappers around operating system functionality. As shown in [8], if threads are not part of the programming language the, compiler can generate wrong code while optimizing the program. Therefore many modern programming languages support threads at a language level and provide explicit descriptions of the used memory model [9].

Mutexes and *semaphores* are the most commonly used and supported synchronization primitives when it comes to protecting access to shared resources. The usage of those low-level primitives is notoriously complicated and error-prone. Several different static verification mechanism have been proposed to verify the correct usage of those locking primitives. We present two example systems which relate closest to our research. Terauchi [10] describes a type-system for generating a linear-system based on its input program. The linear-system is constructed in a way that if and only if the linear system is solvable then the corresponding program is guaranteed to be free of race-conditions. In [11] a concurrent extension of typestate [12] is described to detect race-conditions. While protected

[2]For the upcoming C++0X standard, a thread and memory model is currently under development : http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/

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

Transaction Memory (TM) [13], which avoids the explicit reasoning about which lock protects which shared state, became a very active research area during the past few years. In TM the programmer indicates, by using an atomic-block, which code should run as if it would be one atomic-instruction. Like an atomic instruction, either the whole execution successfully completes and the rest of the system can see the changes or no changes are performed at all. The underlying runtime-system will automatically take care of protecting access to shared resources, detection of possible conflicts and automatic do conflict resolution. We consider TM more an implicit than an explicit concurrency control mechanism because, the programmer does not specify which lock protects which data, he rather declares which piece of code should requires to be run under atomic conditions.

Cilk [14] is a programming language which includes higher-level, but still explicit, concurrency abstractions. Cilk extends the C programming language with three new keywords: `cilk` (to mark spawn-able functions), `spawn` (to spawn function calls asynchronously) and `sync` (to wait for completion of previously started asynchronous functions). Figure 2-1 shows a simple Cilk program for concurrently computing a Fibonacci number. Also Cilk simplifies the management of concurrent tasks, it still relies on the programmer to explicitly specify where and how to extract concurrency and to correctly synchronize access to shared resources. Cheng [15] describes a mechanism to check for race-freedom when locks are used for protecting access to shared resources. The proposed approach is not a general verification, but rather debugging tool for checking the absence of race-conditions for one specific input by sequentiallizing the execution of Cilk program. Besides language-based, higher-level abstractions, one of the major contributions of Cilk is its very efficient runtime-system [16], which employs a work-stealing approach for load-balancing.

Kilim [17] is an actor-based programming language for shared memory systems. In Kilim actors run concurrently inside the same process and communicate via message passing. Similar to Microsoft Singularity operating system [18], Kilim uses statically verified ownership transfer between actors to avoid expensive data copy operations. Therefore Kilim merges the implicit synchronization of message passing with the performance associated to shared memory communication. But the programmer is still in charge of specifying the concurrency and need to map the concrete problem to the actor model.

Axum[4] [19], similar to Kilim, is a actor based programming language. But unlike Kilim, which mainly support mail boxes as communication primitive, Axum provides an extensive set of operators to build dataflow-graphs. To achieve strong isolation between actors, all data that is send must be serialized before the send operation and de-serialized after its reception. Axum supports the specification of communication protocols for specific channels, which allows the detection of certain deadlock scenarios by detecting a protocol violation. To avoid the expensive data copy operations that are involved in the message passing, Axum supports shared state via 'domains'. Domains are groups of data/objects that are shared between several actors. Each actor has to specify if is just reading or

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

[4]Formerly known as Maestro.

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

FIGURE 2-1: A Cilk example program to concurrently compute the Fibonacci number. The `main` and all spawn-able functions must be marked with the `cilk` specifier. In line 21 an asynchronous computation is started, indicated by the `spwan` keyword. With the `sync` keyword the program waits for the completion of this task. While asynchronous executing the `fib` function, it recursively spawns off new asynch. tasks (line 11 and 12), and waits in the following line via the `sync` keyword for their completion.

also writing to the mutable state of the corresponding domain. This allows the system to order accesses to mutable state and avoid race conditions. This approach can be seen as a very simple form of access permissions to domains (to be discussed later).

Dryad [20] is a runtime-system for execution of dataflow graphs. Dryad allows the execution of programs on platforms ranging from multi-core CPUs to big computer clusters. Dryad is mainly used as backend execution engine for high-level concurrent programming languages. For instance DryadLINQ [21], a LINQ[5] implementation, and SCOPE [22], a high-level data processing langauges, use Dryad as (one of) their backend execution engines.

X10 [23] is an new programming language that has been developed in a DARPA-funded supercomputing initiative. It aims at the creation of a next generation programming language for high-performance computing. One of the major design-goals of X10 was the support for distributed computing. X10 uses a global partitioned address space, where the

[5]Discussed in section 2.1.

distinct partitions are called places, to take the *Non-Uniform Memory Access* (NUMA) of current systems into account. X10 allows the programmer to start asynchronous activities in other places to modify or fetch data from places. Asynchronous activities can be coordinated via barrier-like objects, called clocks, which allow the execution of activities based on phases. X10 also provides an atomic-block for protecting the access to shared resources.

Access Permissions are a novel abstraction mechanism, originally introduced by Bierhoff [2] to solve the alias problem in typestate verification. Access permissions encode the information regarding how the referenced object can by used through the current reference as well as through possible other reference in the system. Beckman [24] shows how to use access permission to enforce the correct usage of atomic blocks, by requiring all accesses through shared and full references must occur inside an atomic context.

2.2 Implicit Concurrency

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

Pure functional programming [3] provides a good match for implicit concurrency. The lack of side-effects and the explicit dependencies inside the code allow the runtime-system/compiler to extract high-levels of concurrency. As previously mentioned, pure functional programming is not suitable for all cases (e.g. high productivity and easy of use). Therefore functional programming languages increase their features by allowing mutable state and side-effects. When it comes to mutual state and side-effect, *Haskell* [3] has one of the most interesting approaches in dealing with those. In Haskell all side-effects, namely the change of mutable state, must explicit be mentioned in the function signature. For instance, a function that needs to perform I/O must declare that it requires an I/O monad. The I/O monad is a permission to change the 'world' and everything in it. The flow of the I/O monad is used by Haskell to sequentialize the execution of all methods that change mutable state and therefore avoid race-conditions. Having just one permission for the whole system is rather limiting and leads to a major bottleneck in highly-concurrent systems.

HPF [25], Nesl [26, 27] and ZPL [28] are examples of data-parallel programming languages. In these languages the programmer works mainly with arrays and the application of functions to the all or just part of the array elements. Those programming languages naturally fit to scientific computing, which mainly involves computation on huge data-sets. General purpose programs, like for instance web a server , are hard to realize in such programming languages.

OpenMP [29, 30] is an industry standard for shared memory programming in C and Fortran. OpenMP specifies transparent annotations that allow the compiler to automatically parallelize program. OpenMP supports parallelization of non-regular code via parallel sections and tasks, but the main focus of OpenMP is the parallelization of regular problems that are expressed via loops. The goal of OpenMP annotations is not to tell the compiler how to parallelize the code, but rather to point the compiler to the code-fragments that make most sense and is legal to parallelize. Having said that, the programmer still

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

Figure 2-2: A OpenMP example program that performs a parallel matrix multiplication. After allocating memory and initializing the matrices (line 28, code omitted for brevity) the program calls the `matrix_mult` function to perform the matrix multiplication. The `matrix_mult` implements a standard matrix multiplication algorithm, consisting of three nested loops. The pragma in line 7 tells an OpenMP capable compiler, that the following (the most outer) for-loop should be automatically parallelized.

has a fair amount of liberty on controlling how it's done. In Figure 2-2 a simple OpenMP program for the concurrent computation of a matrix-multiplication is shown. Similar to data-parallel programming languages, OpenMP is a natural fit for scientific computing but has several short-comings when it comes general purpose programming (e.g. limited support for expressing parallelism in irregular structures). Intel developed an OpenMP version for parallelizing programs across multiple machines called *Cluster OpenMP* [31].

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

FIGURE 2-3: A simple PLINQ program for find all persons of the given name that are over 21. Note that the only difference to a normal (sequential) LINQ programs is in line 26, where the program uses the `AsParallel` method to retrieve a parallel collection.

Given the higher overhead of the underlying *distributed shared memory* (DSM) model, this approach seems rather inefficient for most cases.

Language Integrated Query (LINQ) [32] is an extension to the C# programming language, which allows *Structured Query Language* (SQL) [33] like operations on data objects. Any object that implements IEnumerable<T> can the used as source in an LINQ query. This allows LINQ to expression work with a variety of data objects, ranging from simple arrays, over complex collection objects to objects that represent remote databases. The high-level declarative nature SQL is used inside databases for various optimizations, including parallel execution. With *Parallel LINQ* (PLINQ) [34] the same idea is transformed to LINQ. The PLINQ extension allows queries to be executed in parallel, as long as there are no data dependencies between the different computations. Figure 2-3 shows a simple example for concurrently filtering a list of persons that match certain criteria. It is worth to mention that most of (P)LINQ is implemented as library. The main languages changes for supporting (P)LINQ have been the introduction of lambda functions and some synthetic sugar to write the queries in a more SQL style way.

Fortess [35] is one of the most closely related projects to our approach. Fortress has been funded by DARPA for high-performance computing. The syntax of Fortress closely resembles Java's syntax, Fortress employs a significantly different evaluation semantics. In Fortress many evaluation contexts, like for instance tuples and for-loops, execute concurrently by default. In the possible case of data-races, the programmer either has to force a sequential execution or protect critical accesses with an atomic-block. Fortress does not support any mechanism to detect possible data-races. It is the responsibility of the programmer to localize potential data-races and take appropriate counter actions. A very useful feature of Fortress is the usage of UNICODE symbols (e.g the sum or integral symbol), to render formulas and program code in 'pseudo-code' style format. This feature explicitly targets the target user groups of scientist, which have a solid understanding of their domain, but might have limited programming skills.

Several *Automatic Parallelization* approaches and techniques for compilers have been proposed. In general these approaches focus on instruction level parallelism (ILP) by exploiting special vector units or improving pipeline utilization. Nowadays all mainstream compilers [36, 37, 38, 39] support ILP at different levels. While this approaches improve single threaded performance, they do not parallelize the program across multiple CPU cores. Therefore more coarse grain approaches for the automatically parallelization of programs have been investigated. Hall et al. [40] describe SUIF, an automatically parallelizing compiler for coarse grain parallelism. SUIF uses a scalar, an array and an inter-procedural analysis to automatically parallelize loops. A similar approach is used by T-Systems Cell-Compiler [41]. The T-System Cell-Compiler automatically extract parallelism at loop level to execute on a Cell processor [42]. Because both approaches rely on highly regular problem, they are a good fit scientific computing but have limited applicability for irregular, general purpose programs.

CHAPTER
THREE

Research Objectives and Approach

This chapter discusses the overall objectives and approach of our system. First elaborate the placed objectives, describing which features and attributes we expect to hold in our system. Then we discuss in detail how we expect to realize those attributes in our system.

3.1 Objectives

With our Æminium language we intend to show the implications of a 'concurrent by default' programming language. In particular we focus on the following objectives:

Impact We want to evaluate the impact of an concurrent by default programming language. In particular we want evaluate if such an programming language can be used as a general purpose programming language. Additionally we want to evaluate if a concurrent by default programming language provides better support to programmers writing concurrent application than current major programming languages.

Runtime Another objective is the evaluation of implications for the runtime system of a parallel by default programming language. In particular we expect the questions of granularity and implementation techniques will be one of the major challenges for the runtime system.

Scalability The question if our system is capable of scaling to dozens or hundreds of core depends mainly on the implementation of the runtime system. But the best runtime system in the world cannot scale if there is not enough concurrency available. Therefore the evaluation of available concurrency in code application will be another objective of our work.

Over the past year we have worked on the core principles of the Æminium language. The next section provides an overview of the approach.

3.2 Approach

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

```
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 3-1: Example: Unique and Immutable Permissions

constraints, execute the program with the maximum possible concurrency. By following this approach our system resembles a *dataflow architecture* [43]. 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* [2] 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.

3.2.1 Access Permissions for Concurrency

Unique and Immutable Permissions

Consider the application in Figure 3-1 which computes over a collection of data. Starting with line 20 the main function creates a collection containing some random 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 those. Those objects are later needed by code we omitted (line 27). After that, we remove existing duplicates from the collection and then print the updated collection to the screen (line 26).

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. To specify exactly those requirements access permissions are used.

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.

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* [44] 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 not 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 a 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 [45], this function would have the type '`Integer` $\rightarrow$ `String`', stating that the method takes an Object as input and returns an Object. In our system, the same function would have the following access permission signature:

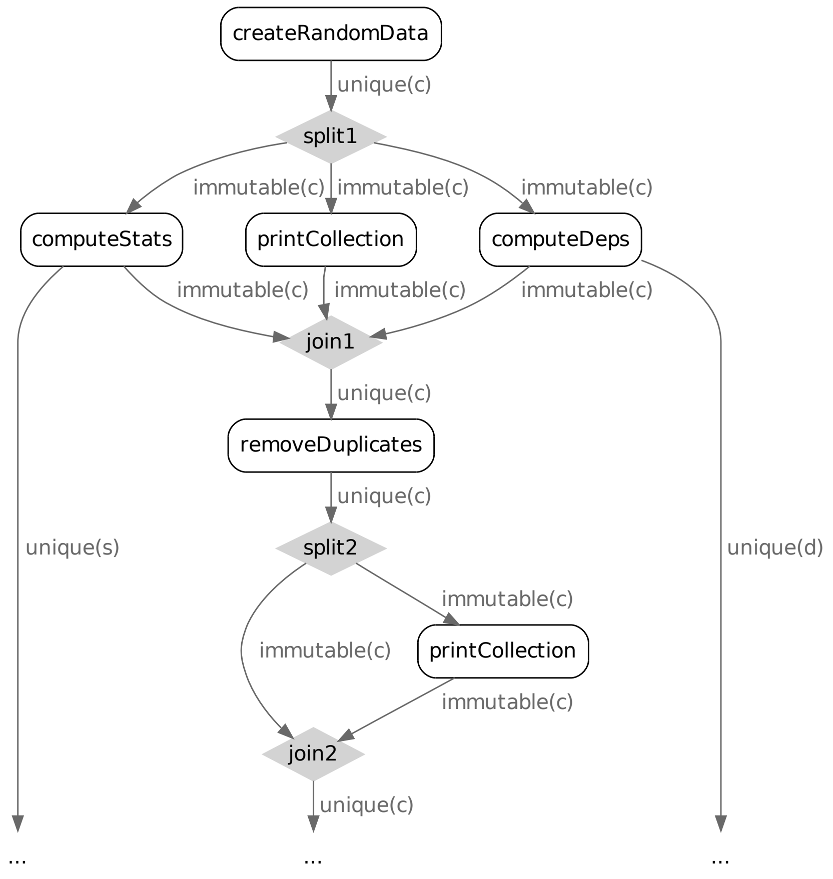


FIGURE 3-2: Example: Unique and Immutable Permissions Flow

immutable(a) ⤇ **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, 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 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`[1] (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.

[1]For accessing the output device our system also requires a permission. To keep the example simple and because data groups (explained in section 3.2.2) offer a better abstraction for dealing with this kind of problems, we omit the I/O-related permissions in this example.

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 3-1, the `createRandomData` method generates a unique permission to the returned collection. The `printCollection`, `compStats` and `compDeps` functions require only a 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 [2], our system performs those 'conversions' by automatically splitting and joining permissions utilizing *fractions* [46]. This means that 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 is no longer available. The reverse works in a similar way. Once all shared or immutable permissions have been collected, the system is able to form a unique permission again by consuming all fractional permissions.

The splitting of the unique permission into three immutable permissions is shown in Figure 3-2 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 3-2, '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 3-2, 'split2'). After the completion of the second `printCollection` method the system will automatically recover the unique permission to the collection[2] (see Figure 3-2, '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.

Shared Permissions

In the previous section we saw how unique and immutable permissions can be used 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

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

```
class Queue {
  void enqueue (Object o)
    : unique(this), shared(o)  ⤇  unique(this)

  Object dequeue()
    : unique(this)  ⤇  unique(this), shared(result)
}

Queue createQueue() : unit  ⤇  unique(result)

void disposeQueue(Queue q) : unique(q)  ⤇  unit

void producer(Queue q) : shared(q)  ⤇  shared(q)
{ atomic { q.enqueue(...) ... } }

void consumer(Queue q) : shared(q)  ⤇  shared(q)
{ atomic { Object o = q.deqeueu() ... } }

void main() {
  Queue q = createQueue()
  producer(q)
  consumer(q)
  disposeQueue(q)
}
```

FIGURE 3-3: Example: Producer/Consumer with Shared Permissions

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.

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* [13] 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. Our system

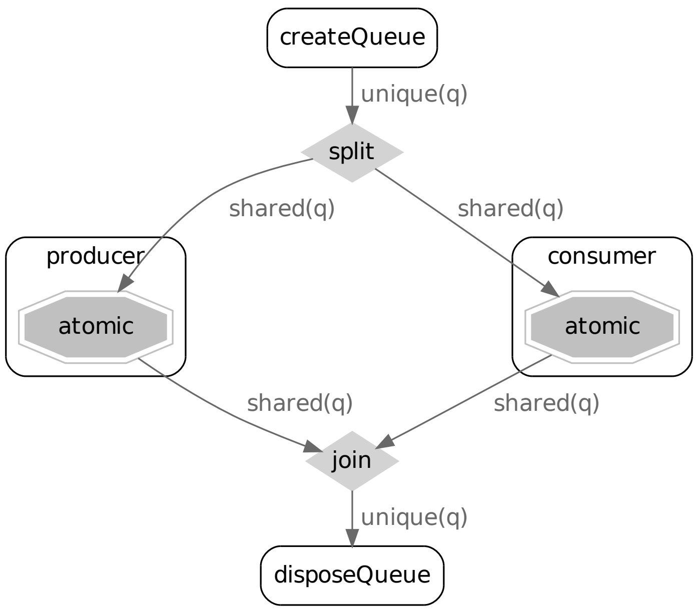


FIGURE 3-4: Example: Producer/Consumer with Shared Permission Flow

can adapt the approach described in [24] to verify and enforce the correct usage of atomic blocks.

Figure 3-3 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 `producer` 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 3-4. 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 3-4, '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 3-4, 'join').

3.2.2 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

```
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 3-5: Example: Concurrent Observer

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* [47] to allow the specification of such high-level dependencies.

Consider the simple observer example shown in Figure 3-5. The program creates a new subject which is then passed to newly created observers and to several `update` method calls. The observer's 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 3-6. 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 unacceptable in other situations (e.g., when the observer depends on the

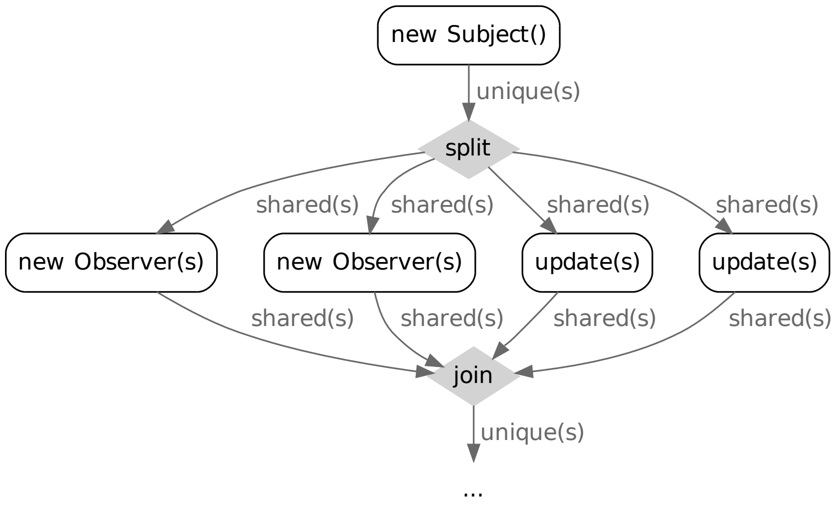


FIGURE 3-6: Example: Concurrent Observer Flow

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.

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 an immutable 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 [48].

Figure 3-7 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 example, 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 3-8. The second 'split' block (line 31) requires an atomic permission to the 'SubG' data group and therefore depends on the completion of the 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 3-8. 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

```
class Subject<SG> {
  void add(Observer<SG> o)
    : shared(this|SG), shared(o|SG)  ⤇  shared(this|SG)

  void update()
    : shared(this|SG)  ⤇  shared(this|SG)
}

class Observer<SG> {
  Observer(Subject<SG> s)
   : shared(s|SG)  ⤇  shared(s|SG), shared(result|SG)
  { s.add(this) }

  void notify(Subject<SG> s)
   : shared(this|SG), shared(s|SG)
       ⤇  shared(this|SG), shared(s|SG)
}

void update(Subject<SG> s)
   : shared(s|SG)  ⤇  shared(s|SG)
{ s.update() }

void main() {
  group <SubG>

  split (SubG) {
    Subject<SubG> s = new Subject<SubG>()
    Observer<SubG> obs1 = new Observer<SubG>(s)
    Observer<SubG> obs2 = new Observer<SubG>(s)
  }
  split (SubG) {
    update<SubG>(s)
    update<SubG>(s)
  }
  ...
}
```

FIGURE 3-7: Example: Concurrent Observer with Data Groups

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.

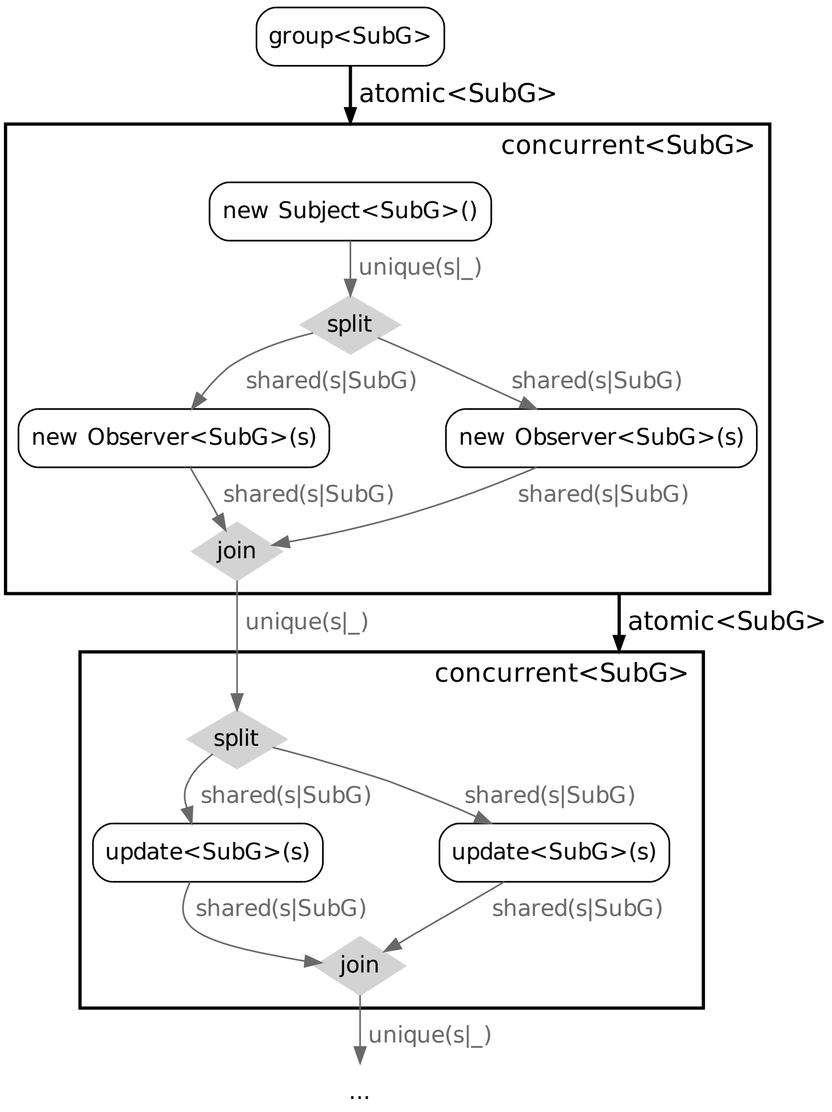


Figure 3-8: Example: Concurrent Observer with Data groups Flow

CHAPTER
FOUR

Current Work and Preliminary Results

So far we started by defining what a concurrent-by-default language could like. In particular what features and attributes we expect such a language to support. We decided to start with the language design aspect first, because the runtime system depends to a certain amount on specifics of the language it is supposed to support (e.g. support for groups or permissions at runtime). To get a better understanding of possible problems and required features, we started to develop a minimum core language calculus based on *Featherweight Java* (FJ) [49]. We extended FJ with unique and immutable permissions and assignments[1]. We called this new calculus *Featherweight Java with Annotations* (FJA). To bridge the gap between FJA and the runtime system, we defined an intermediate language, called *Concurrent FJ* (CFJ), in which data dependencies are explicit represented. We also defined re-writing rules to transfer programs from FJA to CFJ. While developing this basic systems, we encountered several shortcomings which lead us to extend the system to the language that has been described in Section 3.2. In particular we extended the FJA core language with shared permissions and data-groups, resulting in our ÆMINIUM language. Figure 4-1 shows a graphical representation of the relationship between all those languages. The next section provides a short overview of the latest ÆMINIUM grammar. The grammars of all precursor languages, including the rewriting rules, can be found in Appendix A-D.

4.1 Latest Core-Language Specification

While developing the core-langauage which only supported unique and shared permissions, we soon experienced the limitations of this approach. Therefore, we developed ÆMINIUM by extending our core-language with shared permissions and data-groups, as described in chapter 3. Figure 4-2 shows the latest grammar reflecting those extensions in our core language[2]. The major changes compared to the previous core-language grammar (Figure 6-2) are:

Permission-Types As described in Section 3.2.1 we promoted permission to first class citizens. All permission information are now collected in the Method-Specification (MS) and Field-Specifications (FS) (shown in Figure 4-3).

[1]FJ is a pure object calculus, therefore we need to some way of mutable state.

[2]Note, that this grammar represents a reduced feature-set core-language and the syntax might diverge from previously shown Java-style example code

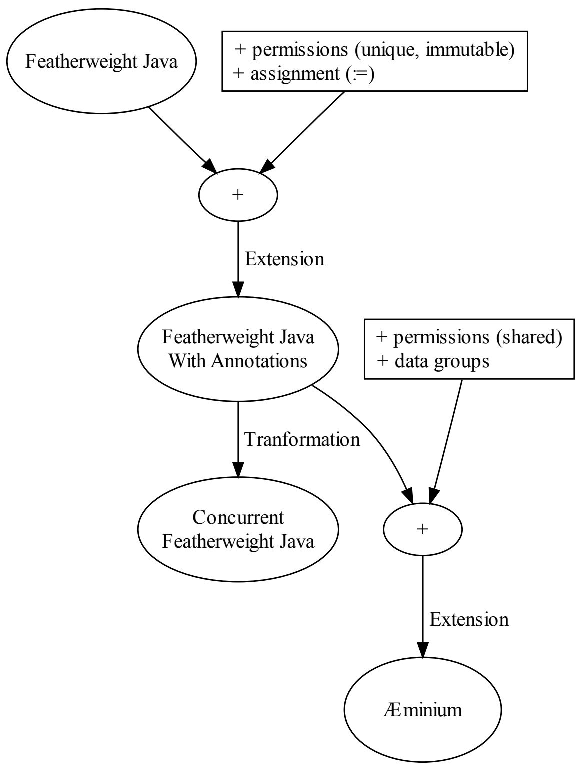


FIGURE 4-1: Language Development Overview/Relationship

$$
\begin{array}{llcl}
\text{(programs)} & P & ::= & \langle \overline{CL}, e \rangle \\
\text{(class decl.)} & CL & ::= & \text{class } C\langle GS' GS \rangle \text{ extends } C'\langle GS' \rangle \ \{ \overline{G}\ \overline{F}\ I\ \overline{M}\ \} \\
\text{(field decl.)} & F & ::= & C\ f : FS \\
\text{(group decl.)} & G & ::= & \texttt{group}\langle gn \rangle \\
\text{(group specs)} & GS & ::= & \overline{gn} \\
\text{(constructor decl.)} & I & ::= & C(\overline{C\ f}) : MS\ \{\ \text{super}(\overline{f});\ \overline{\text{this}.f = f;}\ \} \\
\text{(method decl.)} & M & ::= & D\ m\langle \overline{gk\ gn} \rangle(\overline{C\ x}) : MS\ \{\ \text{return } e;\ \} \\
\text{(references)} & r & ::= & x \mid f \\
\text{(expressions)} & e & ::= & x \mid e.f \mid e.m\langle \overline{gref} \rangle(\overline{e}) \mid \text{new } C\langle \overline{gref} \rangle(\overline{e}) \mid e_1.f := e_2 \\
 & & \mid & \texttt{split}(gref)\{e\} \mid \texttt{atomic}(gref)\{e\} \\
\text{(variables)} & & & \{x, \texttt{this}\}
\end{array}
$$

FIGURE 4-2: Core-Language

Shared Permission The *shared* permission has been added to the set of supported permissions. As described in Section 3.2.1, only shared permission are associated with data-groups. Therefore only shared permissions have a group-reference in their definition.

Data-Groups The core-language now supports the declaration of the data-groups as described in Section 3.2.2. The class and method declaration have been extended to support data-group parameters (enclosed in angle-brackets).

Split-Block The core-language now supports the split-block as described in Section 3.2.2.

Ext. Atomic-Block The core-language has support for the extended atomic block, as describes in Section 3.2.2.

(access permissions)	ap	$::=$	$access(ak, aref, gref)$
(permission kinds)	ak	$::=$	$unique \mid immutable \mid shared$
(permission reference)	$aref$	$::=$	$r \mid$ result
(group kinds)	gk	$::=$	$atomic \mid concurrent \mid locked$
(group refs)	$gref$	$::=$	gn
(permission decl.)	P	$::=$	$P, ap \mid unit$
(methods specs)	MS	$::=$	$P \Rrightarrow P$
(field specs)	FS	$::=$	ap

FIGURE 4-3: Permission Specification

New Symbol We replaced the $\multimap$ with $\Rrightarrow$ symbol, to avoid the impression that those annotation are linear implications that themselves can only consumed once.

4.2 Summary

The preliminary results, namely a concurrent-by-default language with just read/write operations, provided us with insight into possible problems and pitfalls for a concurrent-by-default programming language. In particular this early work allowed use develop the concepts for ÆMINIUM. This early work also lead to two publications: 'Reducing STM Overhead with Access Permissions' [50] describing how permissions can be used to optimize Software Transactional Memory systems and 'Concurrency by Default' [51] describing the concurrency-by-default approach of ÆMINIUM.

Currently we are working on the formal theory to describe ÆMINIUM and how to bridge to the runtime system. In particular we are investigating if there is still a need for an intermediate language (like CFJ) for ÆMINIUM or not. Along this direction we have also to start to think about concrete runtime system implications.

CHAPTER
FIVE

Work Plan and Implications

In this chapter we discuss the overall workplan and anticipated implications. As this dissertation is carried out under the CMU|Portugal program, we first will provide an overview of the program and its implications on this dissertation. For a whole overview of the program refer to the official webpage [52]. The CMU|Portugal program is a collaboration between Carnegie Mellon University (CMU) [53] and several Portuguese universities (Coimbra in the case of this dissertation). The CMU|Portugal PhD program has a duration of 5 years. two of the five years are spend at CMU while the remaining time is spent at the corresponding Portuguese university. Given this time constraint and the different research focuses, we will discuss in the next section a possible workplan for conducting the remaining tasks. In Section 5.2 we will discuss possible target conferences for publishing our results.

5.1 Schedule

As shown in Chapter 4, the design of a new programming language is not a straight forward process. The process, as show in Figure 5-1, is highly iterative. After one round of designing, implementing and evaluation, the newly gathered experience is fed back into the next design round. With every iteration, similar to a spiral, the system gets one step closer to the target system.

Given this exploratory and iterative character of our research, we just provide a coarse-grain schedule shown in Figure 5-2. We do not separately note each iteration, because there is no general rule to determine how many iterations are necessary or how much time each iteration take. Note that we abbreviate Fall 2009 with F08, Spring 2010 with S10 and so on. The presented schedule uses the Portuguese semester regular cycle of 6 month per semester. Figure 5-2 also contains a tentative location schedule, for stays in Carnegie Mellon University (CMU) and the University of Coimbra (UC) respectively.

5.2 Target Conferences

We intend to publish in conferences and workshop of the areas: programming languages , operating systems and virtual machines, parallel/concurrent programming, and high performance computing. Table 5-1 summarizes the target conferences and workshops alongside their usual deadlines.

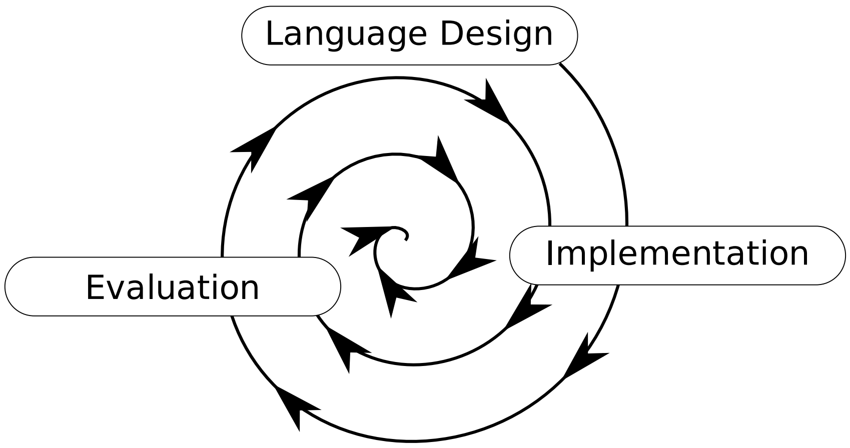


FIGURE 5-1: ÆMINIUM Iterative Design Process

Task	F08	S09	F09	S10	F10	S11	F11	S12	F12	S13
Formal System		▨	▨	▨	▨	▨	▨	▨	▨	
Implementation		▨	▨	▨	▨	▨	▨	▨	▨	
Evaluation		▨	▨	▨	▨	▨	▨	▨	▨	
Writing Thesis									▨	▨
CMU	▨		▨	▨			▨			
UC		▨			▨	▨		▨	▨	▨

FIGURE 5-2: Coarse-Grain Gantt Diagram

Conference	Field	Deadline	Takes Place
Principles of Programming Languages (POPL)	Programming Languages	July	January
Programming Language Design and Implementation (PLDI)	Programming Languages	November	June
European Conference on Object-Oriented Programming (ECOOP)	Programming Languages	December	July
Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)	Programming Languages	March	October
ACM Symposium on Operating Systems Principles (SOSP)	Operting Systems	March	October
Workshop on Hot Topics in Parallelism (HotPar)	Parallel Programming	October	March
IEEE International Parallel and Distributed Processing Symposium (IPDPS)	Parallel Programming	October	May
Principles and Practice of Parallel Programming (PPoPP)	Parallel Programming	September	March
Super Computing (SC)	High Performance Programming	April	November
International Super Computing (ISC)	High Performance Computing	February	June

TABLE 5-1: Target Conferences

CHAPTER

SIX

Conclusions

We proposed a new programming paradigm called: *concurrency-by-default*. In this new paradigm the all parts of a program, to the extends of not violating dependencies, can be executed concurrently by default. Therefore programmer do not longer need to reason about, complicated and error prone, ordering constrains. Programmer simply reason about dependencies and leave the execution and scheduling to the runtime system.

We presented Æminium, a new programming language, designed after the concurrency-by-default paradigm. Æminium uses access permissions and data groups to specify and verify dependencies. So far our investigation focused on the core features of Æminium, resulting in the core language grammar presented in Chapter 4. This core language grammar establishes the base for our future work. In particular we are proceeding with the formalization of the Æminium language, the implementation of an efficient runtime system and investigation of practical solutions to the granularity problem.

Bibliography

[1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 16–20, 2005.

[2] K. Bierhoff and J. Aldrich, "Modular typestate checking of aliased objects," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, (Montreal, Quebec, Canada), pp. 301–320, ACM, 2007.

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

[4] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A Theory of Communicating Sequential Processes," *J. ACM*, vol. 31, no. 3, pp. 560–599, 1984.

[5] Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, Juni 1995. http://www.mpi-forum.org.

[6] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, Juli 1997. http://www.mpi-forum.org.

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

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

[9] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 378–391, ACM, 2005.

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

[11] Y. Yang, A. Gringauze, D. Wu, and H. Rohde, "Detecting data race and atomicity violation via Typestate-Guided static analysis," Tech. Rep. MSR-TR-2008-108, Microsoft Research, Aug. 2008.

[12] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Softw. Eng.*, vol. 12, no. 1, pp. 157–171, 1986.

[13] J. Larus and R. Rajwar, *Transactional Memory.* Morgan & Claypool Publishers, 1 ed., 2007.

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: an efficient multithreaded runtime system,” *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.

[15] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark, “Detecting data races in cilk programs that use locks,” in *SPAA ’98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, (New York, NY, USA), pp. 298–309, ACM, 1998.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the cilk-5 multithreaded language,” *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, 1998.

[17] S. Srinivasan and A. Mycroft, “Kilim: Isolation-typed actors for java,” in *ECOOP ’08: Proceedings of the 22nd European conference on Object-Oriented Programming*, (Berlin, Heidelberg), pp. 104–128, Springer-Verlag, 2008.

[18] G. C. Hunt and J. R. Larus, “Singularity: rethinking the software stack,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, 2007.

[19] Microsoft Corporation, *Axum Programmer’s Guide*, 2009. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *EuroSys ’07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, (New York, NY, USA), pp. 59–72, ACM, 2007.

[21] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language,”

[22] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, “Scope: easy and efficient parallel processing of massive data sets,” *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, 2008.

[23] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, 2005.

[24] N. E. Beckman, K. Bierhoff, and J. Aldrich, “Verifying correct usage of atomic blocks and typestate,” *SIGPLAN Not.*, vol. 43, no. 10, pp. 227–244, 2008.

[25] D. B. Loveman, “High performance fortran,” *IEEE Parallel Distrib. Technol.*, vol. 1, no. 1, pp. 25–42, 1993.

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

[27] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," in *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (San Diego), pp. 102–111, May 1993.

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

[29] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, May 2008. http://openmp.org.

[30] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Oct. 2007.

[31] J. P. Hoeflinger, *Extending OpenMP to Clusters*. Intel Corporation. http://www.intel.com.

[32] E. Meijer, B. Beckman, and G. Bierman, "LINQ: reconciling object, relations and XML in the .NET framework," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 706–706, ACM, 2006.

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

[34] J. Duffy and E. Essey, "Running Queries On Multi-Core Processors." online, October 2007. http://msdn.microsoft.com/en-us/magazine/cc163329.aspx.

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

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

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

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

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

[40] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the suif compiler," *Computer*, vol. 29, no. 12, pp. 84–89, 1996.

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

[42] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*, (New York, NY, USA), pp. 9–20, ACM, 2006.

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

[44] J.-Y. Girard, "Linear logic," *Theor. Comput. Sci.*, vol. 50, no. 1, pp. 1–102, 1987.

[45] R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1990.

[46] J. Boyland, "Checking interference with fractional permissions," in *SAS*, pp. 55–72, Springer, 2003.

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

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

[49] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: a minimal core calculus for Java and GJ," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, 2001.

[50] N. E. Beckman, Y. P. Kim, S. Stork, and J. Aldrich, "Reducing STM Overhead with Access Permissions," in *In Proceedings of the International Workshop on Aliasing, Confinement and Ownership*, July 2009.

[51] S. Stork, P. Marques, and J. Aldrich, "Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs," in *In Proceedings of Onward! Conference*, October 2009.

[52] "The CMU|Portugal Program." http://www.cmuportugal.org/.

[53] "Carnegie Mellon University." http://www.cmu.edu/.

[54] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java (TM) Language Specification*. Addison-Wesley Professional, 2005.

Appendix

A Featherweight Java

Featherweight Java [49] is a small object-oriented language modeled after Java[1][54]. FJ's goals is to provide a minimal object oriented core-language, as starting point for newly designed object-oriented programming languages. To be as small as possible FJ omits many features of Java. The most noticeable missing feature in FJ is the lack of modifiable state, which makes FJ a pure functional programming language. The grammar of FJ is shown in Figure 6-1. As shown, FJ consists of classes which themselves consist of exactly one constructor and an arbitrarily amount of methods and fields. Note that $\overline{\alpha}$ stands for the sequence of $\alpha_1, \alpha_2, \ldots$ and $\overline{\alpha}\ \overline{\beta}$ for the sequence $\alpha_1\ \beta_1, \alpha_2\ \beta_2, \ldots$. This The only operations that FJ permits are the reading of fields, method invocation and creation of new objects.

$$\begin{array}{lcl}
\text{(Classes)} & \text{CL} & ::= \text{class } C \text{ extends } C' \ \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \} \\
\text{(Constructors)} & \text{K} & ::= C(\overline{C}\ \overline{f})\ \{\ \text{super}(\overline{f});\ \text{this}.\overline{f} = \overline{f};\ \} \\
\text{(Methods)} & \text{M} & ::= D\ m(\overline{C}\ \overline{x})\ \{\ \text{return } e;\ \} \\
\text{(Expressions)} & \text{e} & ::= x \mid e.f \mid e.m(\overline{e}) \mid \text{new } C(\overline{e})
\end{array}$$

Figure 6-1: Featherweight Java Grammar

B Featherweight Java with Annotations

As described in the previous section is FJ purely functional. Therefore we have to extend FJ with the possibility of mutable state. Additional we extend FJ with the ability to annotate the parameter of methods and constructors. Methods have an additional annotation for specifying the required permission to the receiver object. The extended grammar for *FJ with Annotations* (FJA) is shown in Figure 6-2.

C Concurrent Featherweight Java

The *Concurrent FJ* (CFJ) language is designed to explicit represent statement dependencies. CFJ can be derived from FJA by the rules described in Appendix D. To express the

[1]FJ is strictly speaking a subset of Java

$$\begin{array}{lcll}
\text{(Permissions)} & \mathrm{p} & ::= & \mathit{unique} \mid \mathit{immutable} \mid \mathit{shared} \\
\text{(Classes)} & \mathrm{CL} & ::= & \text{class } C \text{ extends } C' \;\{\; \overline{p}\; \overline{C}\; \overline{f};\; K\; \overline{M} \;\} \\
\text{(Constructors)} & \mathrm{K} & ::= & C(\overline{p}\; \overline{C}\; \overline{f})\; \{\; \text{super}(\overline{f});\; \text{this}.\overline{f} = \overline{f}; \} \\
\text{(Methods)} & \mathrm{M} & ::= & D\; m(\overline{p}\; \overline{C}\; \overline{x})\; p_{this}\; \{\; \text{return } e;\; \} \\
\text{(Expressions)} & \mathrm{e} & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \text{new } C(\overline{e}) \\
 & & \mid & \underbrace{e_1.f := e_2}_{e_1.f:=e_2 \mapsto e_2}
\end{array}$$

FIGURE 6-2: Featherweight Java with Annotations Grammar

dependencies between statements, the language uses an extended let-normal-form [45] as show in Figure 6-3. Every statement is associated with a unique label, which can be used as reference by other statements to declare a dependency. The dependency of a statement is specified via a set of label between the `sync` and `let`.

$$\texttt{synch}\;\; \underbrace{\overline{\mathit{labels}}}_{\mathit{dependencies}}\;\; \texttt{let}\; \underbrace{\mathit{label}}_{\mathit{name}}\; \mathrm{x} = \langle \text{atom} \rangle\; \texttt{in} \ldots$$

FIGURE 6-3: Let-Synch-Form for Concurrent FJ

The grammar of the concurrent FJ is shown in Figure 6-4. The main changes are the introduction of the extended let-synch-form and the renaming of expressions into their atomic base elements.

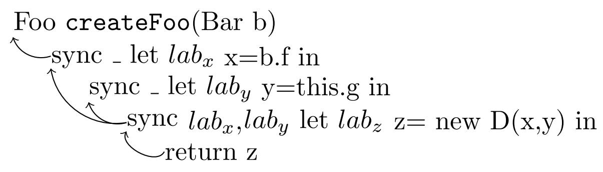
$$\begin{array}{lcll}
\text{(Classes)} & \mathrm{CL} & ::= & \text{class } C \text{ extends } C' \;\{\; \overline{C}\; \overline{f};\; K\; \overline{M} \;\} \\
\text{(Constructors)} & \mathrm{K} & ::= & C(\overline{C}\; \overline{f})\; \{\; \text{super}(\overline{f});\; \text{this}.\overline{f} = \overline{f};\; \} \\
\text{(Methods)} & \mathrm{M} & ::= & D\; m(\overline{C}\; \overline{x})\; \{\; S;\; \} \\
\text{(Sync)} & \mathrm{S} & ::= & \text{synch } \overline{\mathit{lab}} \text{ let } \mathit{lab}\; x = a \text{ in } S \mid \text{return } x \\
\text{(Atom)} & \mathrm{a} & ::= & x \mid x.f \mid x.m(\overline{x}) \mid \text{new } C(\overline{x}) \mid x.f := x
\end{array}$$

FIGURE 6-4: Concurrent Featherweight Java

Imaging the following method of a class: 'Foo `createFoo`(Bar b){ return new Foo(b.f, this.g) }'. The 'createFoo' method passes the 'f' field of the Bar parameter b and the 'g' field of the current object to the constructor of Foo. Then the newly created object is returned. Figure 6-5 shows this method after the transformation into the let-synch-form. The small arrows visualize the data-dependencies between the different let-synch statements (as stated by the label dependencies) and the method parameters.

D Re-writting Rules

This section describes the transformation rules, that convert from FJ with annotations to concurrent FJ. Before we discuss the rules we need to define some auxiliary constructs:



```
Foo createFoo(Bar b)
sync _ let lab_x x=b.f in
   sync _ let lab_y y=this.g in
      sync lab_x,lab_y let lab_z z= new D(x,y) in
         return z
```

FIGURE 6-5: Example Let-Synch-Form

Definition 6-1 (Global Class Permission Context)

$$\boxed{\Omega ::= \varnothing \mid \Omega, C.m.\alpha{:}permission \mid \Omega, C.C.\alpha{:}permission}$$

α selects the permission between of the method receiver or the methods parameter.

$\alpha = 0$ the permission of the receiver object

$\alpha > 0$ the permission of the N'th parameter

lookup : $\Omega(x) \rightarrow permission$

The global class permission context is defined in definition 6-1. The goal of this context is to provide information of which permissions are associated with which parameter or method receiver object. The lookup function must be used with a fully qualified name (FQN), concatenated by the permission index, to retrieve the corresponding permission (e.g. $\Omega(Foo.bar.1)$ returns `unique`). As the name indicates, this is a global context, which is build once during the parsing of the source-code.

Definition 6-2 (Variable Permission-Label Context)

$$\boxed{\Phi ::= \varnothing \mid \Phi, (var, permission, \{\overline{labels}\})}$$

lookup : $\Phi(var, permission) \rightarrow \overline{labels}$

$\Phi(var, unique) \rightarrow label$ (latest unique operation)

$\Phi(var, immutable) \rightarrow \overline{labels}$ (all read operations since the last unique)

update : $\Phi' = \left[\frac{(variable,permission,\{\Phi(variable,permission),label_{new}\})}{(variable,permission,\Phi(variable,permission))}\right]\Phi$

The variable permission-label context, as defined in definition 6-2, is used to dynamically track the relationship between variables and their corresponding read/write set of statements (identified via their labels) (e.g. $\Phi(foo, immutable) \rightarrow \{l_1, l_5, l_{29}\}$). Because the content of this context is changed dynamically, this context supports both a lookup function, also an update function. The update function returns a copy of the old context, in which the corresponding substitution has been performed.

Definition 6-3 (S-context)

$$\boxed{\mathfrak{S} ::= \Box \mid \text{synch } \overline{lab_{deps}} \text{ let } lab_x\ x = e_x \text{ in } \mathfrak{S} \mid \text{return } x}$$

$$\begin{aligned} \textit{update :} \quad & \mathfrak{S}' = \mathfrak{S}[\text{synch } lab_{deps} \text{ let } lab_x\ x = e_x \text{ in } \Box] \\ \Longleftrightarrow \quad & \mathfrak{S}' = \mathfrak{S}\left[lab_{deps} \Rightarrow lab_x(x, e_x)\right] \end{aligned}$$

The S-Context, as defined in definition 6-3, is used to dynamically construct method body for concurrent FJ methods. The update function 'plugs' the given argument into the 'hole' (☐) of the context. Given the way the S-context is designed, it starts out with a single hole, appends let-synch statements (which ends with a hole), until the last hole is filled with an return-statement.

$$\frac{}{(\Phi, \mathfrak{S}_{outer}) \vdash v \rightarrow (\Phi, \mathfrak{S}_{outer}, v)}\ \text{(R-VAR)}$$

$$\frac{\begin{array}{c}(\Phi, \mathfrak{S}_{outer}) \vdash e_1 \rightarrow (\Phi_1, \mathfrak{S}_1, y) \qquad labs_{deps} = \{\Phi_1(y, unique)\} \\ fresh\_var() = x \qquad var\_to\_label(x) = lab_x \\ \Phi_2 = add\_to\_readset(\Phi_1, y, lab_x) \qquad \Phi_3 = set\_var(\Phi_2, x, lab_x)\end{array}}{(\Phi, \mathfrak{S}_{outer}) \vdash e_1.f \rightarrow (\Phi_3, \mathfrak{S}_1\left[labs_{deps} \Rightarrow lab_x(x, y.f)\right], x)}\ \text{(R-READ)}$$

$$\frac{\begin{array}{c}(\Phi, \mathfrak{S}_{outer}) \vdash e_1 \rightarrow (\Phi_1, \mathfrak{S}_1, y) \qquad (\Phi_1, \mathfrak{S}_1) \vdash e_2 \rightarrow (\Phi_2, \mathfrak{S}_2, z) \\ labs_{deps} = \{\Phi_2(y, immutable), \Phi_2(z, unique)\} \qquad fresh\_var() = x \qquad var\_to\_label(x) = lab_x \\ \Phi_3 = set\_var(\Phi_2, y, lab_x) \qquad \Phi_4 = set\_var(\Phi_3, x, lab_x)\end{array}}{(\Phi_1, \mathfrak{S}_{outer}) \vdash e_1.f := e_2 \rightarrow (\Phi_4, \mathfrak{S}_2\left[labs_{deps} \Rightarrow lab_x(x, y.f := z)\right], x)}\ \text{(R-ASSIGN)}$$

$$\frac{\begin{array}{l}(\Phi, \mathfrak{S}_{outer}) \vdash e_1 \rightarrow (\Phi_1, \mathfrak{S}_1, y) \\ \Omega(C.C.1) = p_1 \qquad fresh\_var() = x \qquad var\_to\_label(x) = lab_x \\ (p_1 = unique)\ ?\ (\Phi_2 = set\_var(\Phi_1, y, lab_x)\ ;\ labs_{deps} = \Phi_1(y, immutable)) \\ (p_1 = immutable)\ ?\ (\Phi_2 = add\_to\_readset(\Phi_1, y, lab_x)\ ;\ labs_{deps} = \Phi_1(y, unique)) \\ \Phi_3 = set\_var(\Phi_2, x, lab_x)\end{array}}{(\Phi, \mathfrak{S}_{outer}) \vdash \text{new } C(e_1) \rightarrow (\Phi_3, \mathfrak{S}_1\left[lab_{deps} \Rightarrow lab_x(x, \text{new } C(y))\right], z)}\ \text{(R-NEW)}$$

$$\frac{\begin{array}{l}(\Phi, \mathfrak{S}_{outer}) \vdash e_1 \rightarrow (\Phi_1, S_1, y) \qquad (\Phi_1, \mathfrak{S}_1) \vdash e_2 \rightarrow (\Phi_2, \mathfrak{S}_2, z) \qquad typeof(e_1) = C_1 \\ \Omega(C_1.m.0) = p_0 \qquad \Omega(C_1.m.1) = p_1 \qquad fresh\_var() = x \qquad var\_to\_label(x) = lab_x \\ (p_0 = unique)\ ?\ (\Phi_3 = set\_var(\Phi_2, y, lab_x)\ ;\ labs_0 = \Phi_2(y, immutable))) \\ (p_0 = immutable)\ ?\ (\Phi_3 = add\_to\_readset(\Phi_2, y, lab_x)\ ;\ labs_0 = \Phi_2(y, unique))) \\ (p_1 = unique)\ ?\ (\Phi_4 = set\_var(\Phi_3, z, lab_x)\ ;\ labs_1 = \Phi_3(z, immutable))) \\ (p_1 = immutable)\ ?\ (\Phi_4 = add\_to\_readset(\Phi_3, z, lab_x)\ ;\ labs_1 = \Phi_3(z, unique))) \\ \Phi_5 = set\_var(\Phi_4, x, lab_x) \qquad labs_{deps} = \{labs_0, labs_1\}\end{array}}{(\Phi, S_{outer}) \vdash e_1.m(e_2) \rightarrow (\Phi_5, \mathfrak{S}_2\left[labs_{deps} \Rightarrow lab_x(x, y.m(z))\right], x)}\ \text{(R-CALL)}$$

$$\frac{(\varnothing, \Box) \vdash e_1 \rightarrow (\Phi_1, \mathfrak{S}_1, y)}{D\ m(C\ p)\{\text{return } e_1\} \rightarrow D\ m(C\ p)\{\mathfrak{S}_1\ [\text{ return } y;\ ]\})}\ \text{(R-METHOD)}$$

FIGURE 6-6: Re-Writing Rules

The re-writing rules for the transformation between FJ with permissions and concurrent FJ are shown in Figure 6-6. To increase readability helper functions (see Figure 6-7) have been introduced to hide the sometimes wordy syntax of the context manipulations. The rules use the following kind of judgment:

$$(\mathfrak{S}_{pre}, \Phi_{in}) \vdash e \rightarrow (\mathfrak{S}_{post}, \Phi_{post}, x)$$

$\mathfrak{S}_{pre}$	S-Context before the evaluation of e
Φ_{pre}	Permission-Context before the evaluation of e
e	expression
$\mathfrak{S}_{post}$	S-Context after the evaluation of e
Φ_{post}	S-Context after the evaluation of e
x	the variable which contains the result of e

The judgment evaluates the expression e. During the evaluation process the input contexts, $\mathfrak{S}_{pre}$ and Φ_{pre}, are transformed to the output contexts, $\mathfrak{S}_{post}$ and Φ_{post}. Additionally to the new contexts the judgment returns the variable that hold the evaluation result of the expression. The rules shown in Figure 6-6 work as follows:

R-var The R-var rule simply returns the variable itself along with the unmodified contexts.

R-read The R-read rule evaluates the sub-expression e_1 recursively. The evaluation result of e_1 will be bound to the variable y. Then the rule retrieves all labels ($labs_{deps}$) that are associated with the latest write accesses to y. The rule creates a fresh variable x, along with a corresponding label (lab_x), which will contain the evaluation result of the original expression (e). This local label is then added to the read-set of the y variable by calling the *add_to_readset* helper-method. At last the rule initializes the read-/write-set of the newly created variable to point to the local label, by calling the *set_var* helper-method and updates the S-context, by appending a corresponding let-synch-statement.

R-assign The R-assign rule recursively evaluates the sub-expressions e_1 and e_2 with the results bound to the variables y and z. Then the rule retrieves all labels ($labs_{deps}$) that are associated with the latest read accesses to y and the latest write accesses to z. Then the rule creates a fresh variable x, along with a corresponding local label (lab_x), which contains the evaluation result of the original expression (e). Then the rule reset the read-/write-set of the variables x and y to point to the local label, by calling the *set_var* helper-method and updates the S-context, by appending a corresponding let-synch-statement.

R-new The R-new rule first evaluates the sub-expression e_1 recursively. The evaluation result of the e_1 evaluation will be bound to the variable y. The rule then retrieves all labels ($labs_{deps}$) that are associated with the latest write accesses to y. The rule creates a fresh variable x, along with a corresponding local label (lab_x), which contains the evaluation result of the original expression (e). Then the rules looks-up the permission specification of the constructor parameter (p_1). If the parameter permission is unique, then the rule reset the read-/write-set of the y variable to point to the local label and retrieves all labels ($labs_{deps}$) that are associated with the latest read accesses to y. If the parameter permission is immutable, then the rule adds the local label to the read-set of the y variable and retrieves all labels ($labs_{deps}$) that are associated with the latest write accesses to y. Lastly the rule initialize the read-/write-set of the variables x to point the local label by calling the *set_var* helper-method and updates the S-context, by appending a corresponding let-synch-statement.

R-call The R-assign rule recursively evaluates the sub-expressions e_1 and e_2 with the results bound to the variables y and z. The rule creates a fresh variable x, along with a corresponding local label (lab_x), which contains the evaluation result of the original expression (e). Then the permissions to the receiver object (p_0) and the argument (p_1) is looked-up. If the receiver object requires an unique permission, the rule set the read-/write-set of the y variable to point to the local label and retrieves the old read-set ($labs_0$) of the variable y. Otherwise, if an immutable permission is required, the rule adds the local label to the read-set of the variable y and retrieves to the write-set ($labs_0$) of the variable y. If the argument requires an unique permission, the rule set the read-/write-set of the z variable to point to the local label and retrieves the old read-set ($labs_1$) of the variable z. Otherwise, if an immutable permission is required, the rule add the local label to the read-set of the variable z and retrieves to the write-set (lab_1) of the variable z. Lastly the rule reset the read-/write-set of the variables x to point the local label by calling the *set_var* helper-method, merges the depending labels of the receiver object and the argument to $labs_{deps}$ and updates the S-context, by appending a corresponding let-synch-statement.

R-method The R-method recursively evaluates the sub-expression e, with its result bound to the variable y. Then the rule updates the S-context, by appending, and therewith permanently closing the last hole, a return-statement with the variable y.

$typeof(e) = C$		(return the type/class a given expression)
$fresh\_var() = x$		(returns a fresh variable)
$var\_to\_label(id) = lab_{id}$		(returns a fresh label based on the given variable)
$add\_to\_readset(\Phi, var, label) = \Phi'$	$\iff$	$\Phi' = [(var, immutable) \rightarrow \{label, \Phi(var, immutable)\}]\Phi$
$set\_var(\Phi, var, label) = \Phi'$	$\iff$	$\Phi' = [(var, unique) \rightarrow \{label\}][(var, immutable) \rightarrow \{label\}]\Phi$

FIGURE 6-7: Helper Functions