ÆMINIUM: A Permission-Based Concurrent-by-Default Programming Language Approach

SVEN STORK, KARL NADEN, and JOSHUA SUNSHINE, Carnegie Mellon University
MANUEL MOHR, Karlsruhe Institute of Technology
ALCIDES FONSECA and PAULO MARQUES, University of Coimbra
JONATHAN ALDRICH, Carnegie Mellon University

Writing concurrent applications is extremely challenging, not only in terms of producing bug-free and maintainable software, but also for enabling developer productivity. In this article we present the ÆMINIUM concurrent-by-default programming language. Using ÆMINIUM programmers express data dependencies rather than control flow between instructions. Dependencies are expressed using permissions, which are used by the type system to automatically parallelize the application. The ÆMINIUM approach provides a modular and composable mechanism for writing concurrent applications, preventing data races in a provable way. This allows programmers to shift their attention from low-level, error-prone reasoning about thread interleaving and synchronization to focus on the core functionality of their applications. We study the semantics of ÆMINIUM through μÆMINIUM, a sound core calculus that leverages permission flow to enable concurrent-by-default execution. After discussing our prototype implementation we present several case studies of our system. Our case studies show up to 6.5X speedup on an eight-core machine when leveraging data group permissions to manage access to shared state, and more than 70% higher throughput in a Web server application.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]; D.1.3 [**Concurrent Programming**]; D.1.5 [**Object-Oriented Programming**]

General Terms: Languages, Theory, Performance

Additional Key Words and Phrases: Access permissions, permissions, data groups, concurrency

ACM Reference Format:
Stork, S., Naden, K., Sunshine, J., Mohr, M., Fonseca, A., Marques, P., and Aldrich, J. 2014. ÆMINIUM: A permission-based concurrent-by-default programming language approach. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 2 (March 2014), 42 pages.
DOI: http://dx.doi.org/10.1145/2543920

1. INTRODUCTION

In recent years concurrency has become ubiquitous in a wide range of software systems, from high-performance computers to ordinary laptops, smart phones, and even

This work was partially supported by the Portuguese Research Agency – FCT, through a scholarship (SFRH/BD/33522/2008), CISUC (R&D Unit 326/97) and the CMU/Portugal program (R&D Project Aeminium CMU-PT/SE/0038/2008). Supporting work on the Plaid language was funded through the US NSF grant no. CCF-1116907.
Authors' addresses: S. Stork (corresponding author), K. Naden, and J. Sunshine, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA; email: svens@cs.cmu.edu; M. Mohr, Karlsruhe Institute of Technology, Karlsruhe, Germany; A. Fonseca and P. Marques, University of Coimbra, Coimbra, Portugal; J. Aldrich, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

DOI: http://dx.doi.org/10.1145/2543920

embedded systems. The concurrency models used by applications running on these systems differ widely, including parallel number crunching, task synchronization, and inter-thread communication for hiding I/O latency, among many others.

The problem of concurrency cannot be successfully solved without considering software engineering concerns. Today most software leverages libraries, frameworks, and other reusable software components, and is large enough to be difficult for a single programmer to fully understand. This often leads to cases where a small change in one component breaks a completely unrelated component. In addition to those correctness concerns comes the question of efficiency. Adve and Boehm [2010] show that correct and efficient concurrency support requires programming language support. In particular it is shown that *race freedom*, at the very least, must be supported in programming languages to allow efficient cooperation between hardware and software.

In this article we present ÆMINIUM [Stork 2013], which is to our knowledge the first system to combine automatic parallelization with type-based safe deterministic and nondeterministic concurrency. The ÆMINIUM type system is based on *access permissions*, which express constraints on program aliasing, allowing us to overcome one of the major obstacles in prior automatic parallelization work. This aliasing information allows the compiler to easily build a dependency graph and then to parallelize the code. A novel *permission splitting* operation allows programmers to express when two operations that access the same data are conceptually independent, allowing the compiler to safely extract nondeterministic concurrency in addition to deterministic parallelism.

Our approach permits the user to expose potential parallelism in a predictable way through permissions, but puts the runtime system in charge of the highly platform-dependent task of scheduling that potential parallelism onto hardware resources. Library code can also be more reusable, as the programmer only exposes potential parallelism with permissions, rather than committing to a particular parallelization strategy which may conflict with client code.

The main contributions of this article are as follows.

- A concurrent-by-default programming language leverages permissions and data groups to automatically, safely, and deterministically parallelize applications based on permission flows. While an initial sketch of the approach was presented in Stork et al. [2009], this article fills in the sketch to show how the system actually works, and provides a different (and more workable) design for data group permissions.
- We present a safe approach to integrating nondeterminism into the implicit parallelism model mentioned before. Our approach leverages access permissions to data groups, allowing developers to explicitly specify when nondeterminism is permisible while ensuring the absence of data races.
- A core calculus called μÆMINIUM makes the model just described precise and allows formal reasoning about the system. The formal system consists of:
  - a type system that extracts dependency information and ensures the absence of race conditions;
  - a concurrent-by-default evaluation semantics which models dataflow parallelism at a fine granularity, in contrast to prior type-based concurrency models that used threads or explicit fork-join parallelism; and
  - a proof of type soundness and race freedom.
- We provide a detailed description of our prototype implementation in the Plaid programming language infrastructure.

— Several case studies evaluate our initial implementation which show the benefits and applicability of our system to selected example programs.

1.1. Approach

In ÆMINIUM the programmer uses permissions to specify which data he is accessing and in which way he needs to access the data (e.g., if he is willing to share access to the data with other parts of the code or if he wants exclusive access). Encoding this permission information allows the system to check for the correctness of each function as well as their composition in a modular way. Based on the permission flow through the application ÆMINIUM infers potential concurrent executions by computing a *dataflow graph* [Rumbaugh 1975] which can then be executed by exploiting available, and potentially concurrent, computation resources. ÆMINIUM's type system prevents data races by either enforcing synchronization when accessing shared data or by correctly computing dependencies to ensure a happens-before relationship (meaning conflicting accesses will be ordered according to their lexical order).

Note. ÆMINIUM is implemented in Plaid [Aldrich et al. 2009] which already has first-class support for permissions. We therefore present all examples in ÆMINIUM/Plaid syntax. Plaid's syntax is sufficiently close to Java's syntax to be readily understood. We ignore Plaid's special features (such as typestate) and for the purposes of this article, we consider Plaid's states to be equivalent to Java's classes.

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

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

For this example we assume that the order in which we perform the withdraw and deposit operations does not matter. In particular, they could be executed concurrently because both the withdraw and deposit operations should only affect the specified bank account and no other. To encode this extra information ÆMINIUM uses permission annotations. Permissions [Boyland 2003] specify aliasing and access information for objects. The `transfer` method specifies that it requires a *unique* permission to both bank accounts and an *immutable* permission to the amount parameter. The unique permission means that there is only one valid reference to the specified object in the whole system at the moment of a function call, and modifications to the object within the function are possible. The immutable permission specifies that there might be multiple aliases to this object but none of them can be used to change the object.

Assuming the method declarations for the `deposit` and `withdraw` methods given shortly, ÆMINIUM is now able to compute the permission flow within the `transfer` method. The unique permission of the "to" parameter flows to the `deposit` method while the unique permission of the "from" parameter flows to the `withdraw`. But we only have one immutable permission to the "amount" object while both `withdraw` and `deposit` require one each. Because immutable permissions explicitly allow aliasing

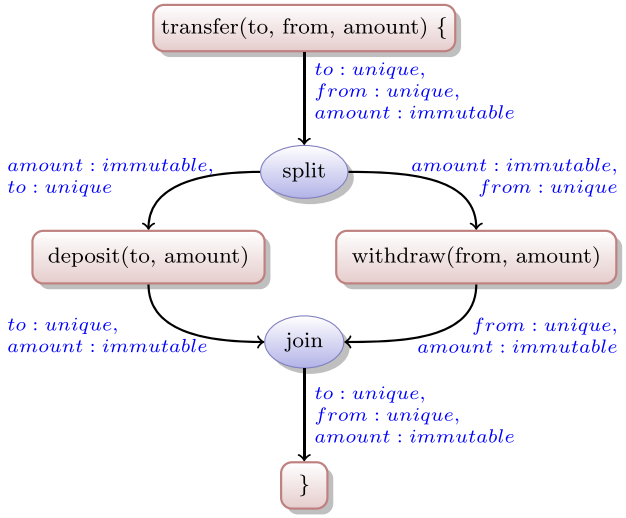


Fig. 1. Permission flow in the transfer example. We use the notation *var* : *perm* to indicate that we have permission "perm" for variable "var".

ÆMINIUM automatically *splits* the one immutable permission into two permissions, which are then passed to the two method calls.

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

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

The permission flow of the `transfer` method is shown in Figure 1. After the split operation the *unique* "to" and *immutable* "amount" permissions are passed to `deposit` method while the unique "from" permission and immutable "amount" permission flow to the `withdraw` method. After those methods complete ÆMINIUM will automatically *join* the previously split immutable permissions. The permission flow graph corresponds to the dataflow graph which is used to execute the `transfer` methods. Although this example illustrates only unique and immutable data, we will later show how ÆMINIUM supports shared mutable data with *shared* permissions and an `atomic` synchronization primitive.

Note that in this example, passing a *unique* `Account` object to be modified by a method is isomorphic to passing an *immutable* `Account` object as an argument and receiving an updated `Account` as the result of the method. One can thus think of state being threaded through the program following the permissions. In this sense, permissions allow us to treat an imperative program as if it were purely functional, with corresponding benefits for reasoning and parallelization. An analogy can be made to monads [Moggi 1991] such as the state monad in Haskell, which conceptually threads the state of the heap through the program computation. However, embedding permissions in a linear logic and providing splitting rules, as discussed shortly, adds flexibility compared to a monadic approach. While we do not explore the monad analogy further in the article, we believe some readers may find it helpful.

In the following example we explore a hypothetical mistake, in which the programmer tries to implement the `available_balance` method to compute the available balance of a given account. For this the caller must pass in the account object along

with an immutable permission. Due to a mistake the programmer adds a call to the `withdraw` method, which attempts to withdraw the specified amount from the given account. The `withdraw` method, though, requires a unique permission to the account and we only have an immutable permission to the specified account. This will result in a typechecking error because an immutable permission cannot be converted into the required unique permission—and fortunately so, because an immutable object can be accessed in parallel, so allowing a modifying access could result in a race condition.

```
method immutable Amount available_balance(immutable Account account) {
    // ...
    withdraw(account, amount);      // typecheck error
    // ...
}
```

1.2. Outline

The article is organized as follows: Section 2 provides an overview of the concept of the ÆMINIUM language; Section 3 gives a detailed description of the core calculus; Section 4 presents an overview of our initial prototype implementation; and Section 5 presents its evaluation; Section 6 discusses the current limitations of our prototype system and future work; Section 7 compares our approach with previous approaches and, finally, Section 8 concludes.

2. OVERVIEW

In this section we describe the ÆMINIUM programming language, which realizes a concurrent-by-default programming model [Stork et al. 2009] with a concrete design and precise semantics. ÆMINIUM uses *access permissions* [Beckman et al. 2008] for objects and data group permissions for *data groups* [Leino 1998] to compute the permission flow throughout the code (explained in the next sections). The compiler uses this information to compute a dataflow graph, which can then be executed in parallel on available computing resources.

While the general ÆMINIUM approach is language agnostic, we use an extended Java syntax for presenting the examples in this section. This requires extending the Java syntax with the missing language constructs and permission annotations. We are currently working on a prototype implementation in the Plaid [Aldrich et al. 2009] language. Plaid has permissions built-in as a first-class language construct and therefore requires only minor extensions to support ÆMINIUM.

2.1. Access Permissions

Access Permissions (AP) have been studied in the past for checking interface protocol compliance and verifying the correct use of synchronization [Beckman et al. 2008]. In ÆMINIUM we use access permissions, and more precisely the flow of the access permissions through the application, to model possible concurrent execution strategies for a program. Access permissions are abstract capabilities associated with object references. The primary purpose of access permissions is to keep track of how many references to a given object exist in a moment in time, and to specify what kind of operations are permitted on the object at that moment. In ÆMINIUM we adopted the following three permissions kinds.

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

— *Shared*. A *shared* access permission to an object reference indicates that there are an arbitrary number of references to the object in the system and *all* the permissions are *shared*. A *shared* access permission allows the client to read and modify the object.
— *Immutable*. An *immutable* access permission to an object reference indicates that there are an arbitrary number of references to the object in the system and *all* of them are *immutable*. An *immutable* access permission allows only read access to the object.

Access permissions follow the rules of *linear logic* [Girard 1987]. They are analogous to physical resources that are unavailable once consumed. Permissions can be converted from one type to another as long as the previously described invariants hold. For instance, a unique AP can be *split* into two shared APs. Because of the linearity of APs the unique AP is gone, having been replaced by two shared APs. Each of the shared APs can be further split into more shared APs, but not into unique or immutable permissions. Using *fractions* [Boyland 2003] for keeping track of the individual AP allows permissions to be *joined*, eventually enabling the recovery of a unique access permission.

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

2.2. Data Groups

Although pure APs define a clean execution model for *unique* and *immutable* data, our permission splitting rules will allow all operations on shared data to proceed concurrently. We need a way to express when one operation on a *shared* data structure depends on another. Furthermore, we'd like to control these dependencies, as well as synchronization on *shared* data, at a granularity greater than one object at a time.

To address this challenge we leverage *Data Groups* (DG, [Leino 1998]). A data group represents an abstract collection of objects. Using data groups for grouping multiple objects differs from previous work [Leino et al. 2002], which used data groups exclusively to partition the state of one object. When an object is part of a data group, we say that this object is *owned* by that data group. In ÆMINIUM each *shared* object must be part of exactly one data group. The specific data group an object is in can change during runtime execution. To transfer a shared object from one data group to another one, all shared permissions to the object must be joined into a unique permission. Only when a unique permission has been reassembled is it possible to split this unique permission into shared permissions associated with a different data group. We write *shared*⟨*myGroup*⟩ to indicate that the shared object is part of the data group *myGroup*. Data groups need to be declared in a state but are instance specific (like instance-specific fields). When an object is allocated, the data groups associated with it are instantiated by the compiler/runtime system. The global set of data groups partitions the heap of shared objects into disjoint parts, which do not overlap.

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

— *Exclusive*. There is at most one *exclusive* GP to a data group in the whole system at a time. This resembles a unique AP. Similar to a unique permission, an exclusive

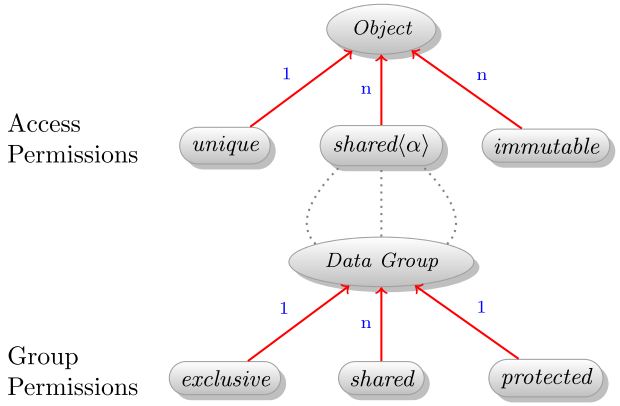


Fig. 2. Permissions in ÆMINIUM. Shows different permission kinds and what each permission controls (including arity). Access permissions control access to objects and group permissions control access to data groups of shared objects. There can only exist one unique, exclusive, or protected permission to an object or data group at a time in the system, while there can be an arbitrary number of shared and immutable permissions. Shared permissions refer to the data group to which they belong (e.g., *shared*⟨α⟩ means the object belongs to data group α).

GP represents the only currently existing permission through which the data of the data group can be accessed.
An *exclusive* group permission behaves like "thread-local" data (although we do not have the notions of threads in ÆMINIUM). An execution path that holds an *exclusive* group permission can safely access the associated shared objects of the group without synchronization. This is an important feature as many data structures intrinsically require shared access permissions to the objects they are composed of (e.g., a doubly linked list which requires at least two valid references to the linked node objects).

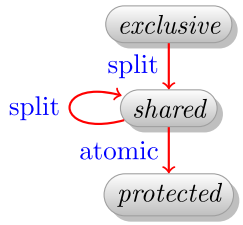
— *Shared*. A *shared* GP resembles a shared AP: there can be an arbitrary number of shared GP in the system. Having a shared GP does not grant any kind of access to the associated data because there is the danger of data races.
— *Protected*. A *protected* GP indicates that access to the shared data is safe because the access to the *shared* data group has been protected by a corresponding `atomic` block. The semantics of protected permissions is that there can only be one protected permission per data group at a time. This is enforced by the runtime system. In contrast to an *exclusive* permission, a *protected* permission cannot be split into *shared* permissions; doing so would be tantamount to requesting concurrency within an atomic block, likely with confusing and even error-prone semantics.

Figure 2 provides a global overview of all available permissions in the ÆMINIUM system. Access permissions are used to classify object references and consist of *unique*, *shared*, and *immutable*. By definition every shared object must be associated with a data group (e.g., α) for which we use a data group permission *exclusive*, *shared*, and *protected*.

2.2.1. Management of Data Group Permissions. Unlike the automatic splitting of access permissions, data group permissions are split and joined manually to provide the programmer with better control over dependencies between operations. By default, each operation on a data group depends on the previous operation on that data group; when the operations are conceptually independent, an explicit `split` block is used to split an *exclusive* GP into an arbitrary number of *shared* GPs (see Figure 3). The `split` block

```
1  // gr : gp with
2  // gp ∈ {exclusive, shared}
3  split ⟨gr⟩ {
4      // gr : gp with
5      // gp : shared
6      atomic ⟨gr_i⟩ {
7          // gr_i : protected
8      }
9      // gr : gp with
10     // gp : shared
11 }
12 // gr : gp with
13 // gp ∈ {exclusive, shared}
```

(a) split/atomic block



(b) group permission conversion diagram

Fig. 3. Group permission splitting/joining via shared and atomic blocks. The notation $gr : gp$ means that we have group permission gp for data group gr.

specifies data groups for which it splits the available permission (either exclusive or shared) into more shared permissions (one for each statement in the body). Group permissions to data groups not mentioned are simply passed into its body. The available permissions inside the body are partitioned into disjoint sets. Each one of those permission subsets flows to one statement of the body. This means that if multiple statements in the block require the same unique AP, or any GP that is not mentioned in the `split` block, then the code will not typecheck because permissions cannot be duplicated. After the completion of all body statements, the split block joins the generated shared permissions back to the permission that existed before the block was entered.

In order to give programmers control over the granularity of synchronization, each `atomic` block protects access to objects in the particular data groups that are specified at the `atomic` block entry point. It will provide a *protected* GP for the specified data group to its body expression. The specification of the data group is optional as the compiler can automatically infer the required data groups from the provided arguments at the call site. This is similar to C++ which can deduce template parameter type from the provided arguments. In ÆMINIUM's case the type of the arguments encodes which data groups the shared objects are associated with and the compiler can use this information to deduce the required data group parameter information. Providing an explicit annotation, however, provides useful documentation of the programmer's intent and helps catch unintended data accesses. In particular, the semantics of the `atomic` block is that its body is executed as if it has exclusive access to the shared data associated with the specified data group. Similar to the `split` block, the `atomic` block will upon its completion revert the GP to the state it was in before entering the `atomic` block. The semantics of `split` and `atomic` blocks is illustrated by example in Figure 3.

Data groups are declared inside states in a similar way to fields (see Figure 4, line 6). Data groups are only visible inside states and their substates (similar to Java's `protected`). Before accessing data associated with those inner groups, the programmer must gain access to those data groups via an "**unpackInnerGroups** {...}" construct. The `unpackInnerGroups` block, similar to the *focus* operation from Fahndrich and DeLine [2002], will trade the permission to the owner group of the receiver object for permissions to inner groups defined in the receiver's state. This exchange prohibits recursive method calls from accessing the same inner groups, which would violate the permission invariants (e.g., only one exclusive data group permission per data group). What happens is that when `unpackInnerGroups` is called, the exclusive permission for the "owner" is replaced by exclusive permissions for the inner data groups of the receiver object (i.e., the "this" object). This approach transitively avoids the need for

```
1  state DoubleLinkedListItem⟨data⟩ {
2    ... // standard double linked list item
3  }
4
5  state DoubleLinkedList⟨data⟩ {
6    group⟨internal⟩ // inner data group
7
8    // 'head' belonging to inner data group 'internal'
9    shared⟨internal⟩ DoubleLinkedListItem⟨internal, data⟩ head;
10
11   method void
12   add⟨exclusive owner, shared data⟩(shared⟨data⟩ Object⟨data⟩ o)
13     : shared⟨owner⟩ // shared permission to the receiver
14   {
15     // owner : exclusive, data : shared
16     unpackInnerGroups {
17       // internal : exclusive, data : shared
18       // access internal data directly
19     }
20     // owner : exclusive, data : shared
21   }
22
23   method void
24   add⟨shared owner, shared data⟩(shared⟨data⟩ Object⟨data⟩ o)
25     : shared⟨owner⟩ // shared permission to the receiver
26   {
27     // owner : shared, data : shared
28     unpackInnerGroups {
29       // internal : shared, data : shared
30       atomic ⟨internal⟩ {
31         // internal : protected, data : shared
32         // need protection to access internal data
33       }
34     }
35     // owner : shared, data : shared
36   }
37   ...
38 }
```

Fig. 4. A doubleLinkedList with data groups. The example has two `add` methods. The first one requires an exclusive permission to the owner and transitively provides an exclusive permission to the inner groups, and does not require synchronization. The second version only requires a shared permission to the owner and only provides shared permissions to the inner groups, requiring synchronization, that is, `atomic` blocks. In comments "//" we show which permissions we currently hold via the notation $dg : gp$, meaning for data group dg we have permission gp.

synchronization. Analogously, when the client has either a shared or protected permission to the owner (rather than exclusive), the owner permission is replaced by a shared permission to the inner groups. The `unpackInnerGroups` block could automatically be inferred by the compiler (by simply determining which statements need inner data groups and wrapping them in an `unpackInnerGroups` block), but adding it explicitly aids in documenting the programmer's intent. Despite the manual group permission management ÆMINIUM's type system guarantees the absence of race conditions.

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

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

To make this more clear, consider the doubly linked list example in Figure 4. In line 5, the `DoubleLinkedList` state is defined with group parameter `data`, using the same syntax as *Java* type parameters. The `data` group parameters specifies the data group to which the objects stored in the list belong. Line 6 defines a new data group called "internal". Line 9 declares the "head" field pointing to the chain of "DoubleLinkedListItems" which are all associated with the "internal" data group of the surrounding "DoubleLinkedList". Because inner groups are not visible outside the state it is impossible for these objects to leave the scope of the state. This strong encapsulation resembles ownership types [Clarke et al. 1998], and allows ÆMINIUM developers to *incrementally* refine their internal data structures to increase internal concurrency (e.g., in our case study that follows, modifying a hash table that uses one data group for all hash buckets to an implementation that uses one data group per hash bucket).

Lines 12 and 24 show the definitions of two add functions that specify data group parameters along with their required permissions. The signatures of the two `add` methods are identical, with the exception that the `add` method in line 12 requires an exclusive permission to the data group that owns the receiver, while the add method in 24 requires a shared GP. The effect of this difference can be observed in the implementation of the corresponding bodies. In the case of the `add` method that requires an exclusive permission to the receiver's data group, the `unpackInnerGroups` can provide an exclusive permission to the inner data groups, which in turn allows the programmer to access the shared inner state without any synchronization. In the case of the `add` method that requires a shared permission to the receiver's data group, the `unpackInnerGroups` can only provide a shared permission to the inner data groups, requiring the programmer to synchronize on the inner data group (line 30).

Note that the current design of ÆMINIUM only protects against race conditions and not against deadlocks. The latter has been handled in prior work [Boyapati et al. 2002], which is orthogonal to our approach and is left out of this discussion for simplicity.

2.3. Producer/Consumer Example

After the discussion of access permissions, data groups, and their relationships we now present a producer/consumer example in ÆMINIUM (see Figure 5). The program starts execution at the global entry method `main` (line 19). When entering the body it has an exclusive permission to a data group α. This permission will first flow into the `createQueue` method call (line 21). The exclusive permission matches the method permission requirements as specified in line 16. After the `createQueue` call returns the exclusive permission to α, the permission flows into the *split* block at line 23. As previously described, the `split` block will replace the exclusive permission with one corresponding shared permission for each statement in its body. This leads to the fact that one shared permission to α is flowing in parallel to the `producer` and `consumer` method calls (line 24 and 25). After those calls have been completed and therefore have returned their shared permissions to α, the share block will collect them and join them back together to an exclusive permission (line 26). This newly gained exclusive permission is then fed to the `disposeQueue` method call. Note that if either `producer`

```
1  state ProducerConsumer {
2    method void producer⟨shared γ⟩(shared⟨γ⟩ Queue⟨γ⟩ q) {
3      // α : shared
4      atomic ⟨γ⟩ {
5            // α : protected
6            ...
7      }
8    }
9    method void consumer⟨shared γ⟩(shared⟨γ⟩ Queue⟨γ⟩ q) {
10     // α : shared
11     atomic ⟨γ⟩ {
12           // α : protected
13           ...
14     }
15   }
16   method shared⟨γ⟩ Queue⟨γ⟩ createQueue⟨exclusive γ⟩(){...}
17   method void disposeQueue⟨exclusive γ⟩(shared⟨γ⟩ Queue⟨γ⟩ q){...}
18
19   method void main⟨exclusive α⟩() {
20     // α : exclusive
21     shared⟨α⟩ Queue⟨α⟩ q = createQueue⟨α⟩()
22
23     split ⟨α⟩ {
24         producer⟨α⟩(q)   // α : shared
25         consumer⟨α⟩(q)  // α : shared
26     }
27     // α : exclusive
28     disposeQueue⟨α⟩(q)
29   }
30 }
```

Fig. 5. Producer/consumer example.

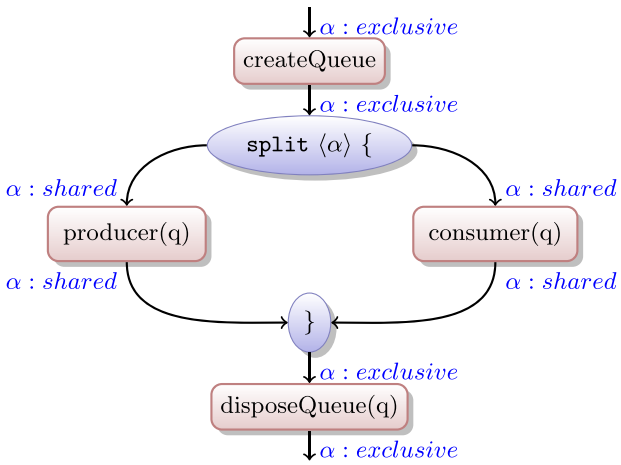


Fig. 6. Dataflow graph for producer/consumer example.

or `consumer` want to access the shared queue, they first have to protect their access to this data group via an atomic block (lines 4 and 11). Figure 6 shows the resulting permission flow and the derived dataflow graph for this example program.

```
1  method void exchange⟨exclusive S,
2                       exclusive I,
3                       exclusive O⟩(shared⟨S⟩ Socket s,
4                                    shared⟨I⟩ Packet inp,
5                                    shared⟨O⟩ Packet outp) {
6      receivePacket⟨S, I⟩(s, inp);
7      checkPacket⟨I⟩(inp);
8      updatePacket⟨O⟩(outp);
9      sendPacket⟨S, O⟩(s, outp);
10 }
```

Fig. 7. Exchange source code.

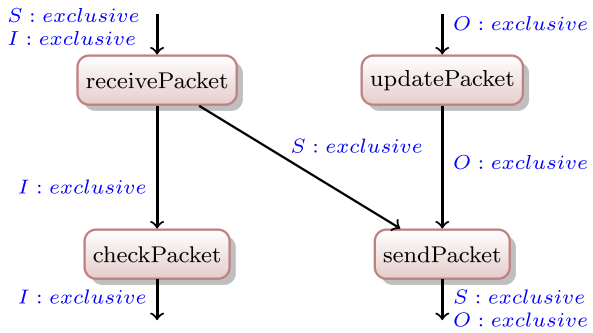


Fig. 8. Dataflow graph for `exchange` function (for simplicity we show only the flow of data group permissions as the access permissions do not cause additional dependencies).

2.4. Dataflow is not Fork/Join

ÆMINIUM supports both *dataflow* and *fork-join* parallelism. To better understand the difference between those concepts, consider the example shown in Figure 7. The `exchange` function, which could be part of a bidirectional ring network implementation, receives a new packet via the provided socket *s* into the packet *inp*. It then checks the newly received packet *inp* for errors (e.g., that checksums match). The function then updates the outgoing packet *outp* (e.g., updates header fields and recomputes checksums), before this packet is sent through the socket.

Assuming that all functions called in the `exchange` method require *exclusive* permissions to the corresponding data groups, the permission flow forms a graph as shown in Figure 8. The graph shows that receiving the incoming packet can be performed in parallel to updating the outgoing packet. As soon as the incoming packet has been received the newly received packet can be checked. When additionally the updates of the outgoing packet have completed, the outgoing packet can be sent in parallel to checking of the incoming packet. This kind of parallelism is naturally supported by ÆMINIUM's dataflow approach, but cannot be directly expressed in a fork-join paradigm unless extra dependencies or synchronization is used.

3. FORMAL LANGUAGE

This section formalizes the object-oriented μÆMINIUM core language. We briefly discuss the syntax of the language and then elaborate on how the static and dynamic semantics of the calculus prohibit race conditions. We conclude this section by describing the soundness properties we have proved for μÆMINIUM. The goal of μÆMINIUM is to explore a simple, efficient mechanism to track data dependencies via permission flow and to guarantee the absence of race conditions. Because only shared data can

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

$C, D, E \in$ CLASSES — $m \in$ METHODS
$f \in$ FIELDS — $x, y, \texttt{this} \in$ VARS
$\alpha, \beta, \gamma \in$ GROUP VARS — $o \in$ OBJ. REFS.
$gn \in$ GROUP NAMES

Fig. 9. μÆMINIUM grammar.

lead to race conditions and the tracking of object permissions and data group permissions can be done using similar mechanisms, we focused the core calculus on modeling data groups and data group permissions, assuming that all data is implicitly shared and omit *immutable* and *unique* permissions from our formal system (note that our implementation has support for all discussed permissions). μÆMINIUM's typechecking rules generate a *data group configuration* representing the graph of dependencies between primitive expressions in the language; this configuration is used along with runtime permissions to model parallel execution in the dynamic semantics.

3.1. Syntax

The grammar of μÆMINIUM is shown in Figure 9 and is formulated as an extension to *Featherweight Java* (FJ, [Igarashi et al. 2001]). Our extensions are highlighted in red.

In a nutshell the major extensions to FJ are: (*i*) addition of data group parameters to method calls, and class and method declarations; (*ii*) addition of group types, and

extension of object types to be parameterized with group parameters; (*iii*) new language constructs to deal with data groups and to support assignment.

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

A program consists of a set of classes and a `main` method. In μÆMINIUM the global starting expression of *FJ* is explicitly wrapped in a `main` method, to provide an initial data group for the top-level objects. A class declaration (*CL*) gives the class a unique name *C* and defines its data group parameters, internal data groups (*G*), fields (*F*), and methods (*M*). Note that the sequence of data group parameters may not be empty, and instead of having an explicit `owner` parameter, the first data group parameter specifies the data group to which the class instances belong. μÆMINIUM does not provide an explicit constructor. Upon creation of a new object all its fields are initialized to `null` and must later be explicitly set. Fields (*F*) are declared with a name and type. Data groups (*G*) are declared by name, which is passed to the group constructor. Methods (*M*) specify their result type, the data group permissions they require, their formal parameters, and a body expression.

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

3.2. Static Semantics

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

3.2.1. Typing Context. The *typing context* Γ contains all the typing information for object references and data group references. We use $\mathbb{G}$ as the type for all data group references.

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

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

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

3.2.3. Data Group Configuration. The *data group configuration* $\mathcal{G}$ hierarchically tracks the data group requirements of an expression, including any ordering or concurrency among those requirements. It vaguely resembles NESL's [Blelloch and Greiner 1996] approach for tracking profiling information, but instead of tracking operation costs we

track permission requirements. A data group configuration can either be empty ($\bullet$); a collection of group references ($\{\overline{gr}\}$), indicating the permission requirements of the current expression; the sequential composition of data group configurations ($\oplus$), used to combine data group configurations of expressions that are sequentially ordered, or the parallel composition of data group configurations ($\|$), used to combine data group configurations of expressions that are executed in parallel. We also define a global data group configuration table ($\mathcal{GT}$) which maps class and method tuples to data group configurations.

(DG configuration) $\mathcal{G} ::= \bullet \mid \{\overline{gr}\} \mid (\mathcal{G}_1 \oplus \mathcal{G}_2) \mid (\mathcal{G}_1 \parallel \mathcal{G}_2)$

($\mathcal{G}$ table) $\mathcal{GT} ::= \bullet \mid \mathcal{GT}, \langle (C, m) \mapsto \mathcal{G} \rangle$

Example. Let us consider a simplified example to provide an intuition for how the data group configuration is used to control execution. Let us assume we have a given expression e which represents a normal let binding with a corresponding data group configuration $\mathcal{G}$. It consists of the sequential composition of the data group configurations of its subexpressions (i.e., $\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2)$ where $\mathcal{G}_1$ and $\mathcal{G}_2$ are data group configurations of subexpressions e_1 and e_2). Furthermore, assume without loss of generally that the required data groups for those subexpressions are $requiredPerms(\mathcal{G}_1) = \{gr_0, gr_1\}$ and $requiredPerms(\mathcal{G}_2) = \{gr_0\}$.

$\mathcal{G} := (\mathcal{G}_1 \oplus \mathcal{G}_2)$ $\quad requiredPerms(\mathcal{G}_1) = \{gr_0, gr_1\}$, $requiredPerms(\mathcal{G}_2) = \{gr_0\}$

$e :=$ **let** x = e_1 **in** e_2

For the moment consider the simple evaluation judgment $\delta | \mathcal{G} \vdash e \mapsto e' \dashv \mathcal{G}'$, meaning, given the runtime permissions δ and the expression e with its data configuration $\mathcal{G}$, the expression e steps to a new expression e' with its new data group configuration $\mathcal{G}'$.

$\{gr_0, gr_1\} \mid \mathcal{G} \vdash$ **let** x= e_1 **in** $e_2 \mapsto$ **let** x = e_1' **in** $e_2 \dashv \mathcal{G}'$

$\mathcal{G}' := (\mathcal{G}_1' \oplus \mathcal{G}_2)$ $\quad requiredPerms(\mathcal{G}_1') = \{gr_1\}$, $requiredPerms(\mathcal{G}_2) = \{gr_0\}$

$e' :=$ **let** x = e_1' **in** e_2

The first subexpression e_1 requires all available runtime permissions, and because of the sequential composition operator $\oplus$ the runtime system needs to satisfy its requirements first. Therefore there are no runtime permissions for the second expressions e_2 left. The system steps e_1 to e_1' and updates its data group configuration to $\mathcal{G}_1'$. As shown before, assume that with this step all remaining operations in e_1' solely depend on the runtime permission gr_1 indicated by $requiredPerms(\mathcal{G}_1') = \{gr_1\}$. In the next execution step, the runtime system again first needs to satisfy the dependencies of e_1' before e_2. But this time e_1' does not require all available runtime permissions, which allows the system to provide the remaining runtime permissions to e_2. This allows the system to step e_1' and e_2 in parallel as shown next.

$\{gr_0, gr_1\} \mid \mathcal{G}' \vdash$ **let** x = e_1' **in** $e_2 \mapsto$ **let** x = e_1'' **in** $e_2' \dashv \mathcal{G}''$

$\mathcal{G}'' := (\mathcal{G}_1'' \oplus \mathcal{G}_2')$

$e'' :=$ **let** x = e_1'' **in** e_2'

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

Fig. 10. μÆMINIUM helper functions.

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

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

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

3.2.5. Helper Functions. Throughout the typing and evaluation rules we use several helper functions to abbreviate common functionality. For space reasons we delegate the full definitions of these functions to a companion technical report (submitted as supplementary material) and just provide a short overview of their effects in Figure 10.

3.2.6. Typing Rules. The typing rules are shown in Figure 11. Most rules are straightforward; we highlight the most interesting ones. T-PROGRAM starts the checking with a top-level data group α. The T-UNPACKGROUPSIN-* rules exchange a permission to the data group of an object for a permission to the inner groups of that object. In the case that we have a *unique* permission to the receiver object we get *exclusive* group permissions (i.e., T-UnpackGroupsIn-Exclusive) in all other cases we get shared group permissions (i.e., T-UnpackGroupsIn-Shared). We could always unpack inner group permissions to shared group permissions, but making the distinction allows us to avoid unnecessary synchronization overhead in the case we know that we do not need it (i.e., in the case of a unique object). T-SPLIT splits the incoming permission context in two, duplicating the named *shared* permissions, while T-ATOMIC allows the protected expression to treat a *shared* data group as *protected*. T-LET supports sequential composition, as specified by the group configuration $\mathcal{G}_1 \oplus \mathcal{G}_2$, while T-SHARE specifies parallel use of any shared groups, as specified by the group configuration $\mathcal{G}_1 \parallel \mathcal{G}_2$. T-FIELD-READ and T-FIELD-ASSIGN require an *exclusive* or *protected* permission to the first data group parameter (gr_0) of the object being read or assigned. This ensures that either a data group is unshared, or it is locked with an `atomic` section before being used. Field reads and writes generate a data group configuration that is just the group being read or assigned. Finally, T-CALL ensures that the data groups required by the called function are provided by the caller. For a more detailed description of each rule refer to Stork et al. [2010].

T-PROGRAM

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

T-FIELD

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

T-METHOD

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

T-CALL

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

T-CLASS

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

T-UNPACKGROUPSIN-EXCLUSIVE

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

T-UNPACKGROUPSIN-SHARED

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

T-SPLIT

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

T-ATOMIC

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

T-INATOMIC

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

T-LET

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

T-REFERENCE

$$\frac{\Gamma|\Sigma \vdash r : D\langle\overline{gr}\rangle}{\Gamma|\Sigma|\Delta \vdash_C r : D\langle\overline{gr}\rangle\ |\bullet}$$

T-FIELD-READ

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

T-FIELD-ASSIGN

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

T-NEW

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

Fig. 11. Static semantics of μÆMINIUM.

3.3. Dynamic Semantics

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

3.3.1. Store. The *store* μ is a mapping of object references o to objects obj. A store can either be a potentially empty set of object mappings or `race`, which indicates the case that a race condition occurred during the execution (our soundness theorem will show that these races cannot occur in well-typed code). An object is a record consisting of all instance fields. The inner groups (i.e., data groups that are declared by every object) along with their corresponding state are managed separately in the group access token context (refer to Section 3.3.3).

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

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

$$\mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2} = \begin{cases} \mu_{\delta_1}, \mu_{\delta_2} & dom(\mu_{\delta_1}) \cap dom(\mu_{\delta_2}) = \bullet \\ \texttt{race} & \text{OTHERWISE} \end{cases}$$

$$\mu' = [\,\mu_\delta\,]\,\mu = \begin{cases} \texttt{race} & \mu_\delta = \texttt{race} \\ [\,o \mapsto obj\,]\,\mu & \forall \langle o \mapsto obj \rangle \in \mu_\delta \end{cases}$$

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

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

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

3.3.3. Group Access Token Context. The *group token context* Ψ is a set of group access tokens, that is, group references along with their current locking state $S = \{U|L\}$. A locking state U indicates an unlocked state, meaning that one atomic block referring to that data group can be entered. A locking state L indicates a locked state meaning that an atomic block referring to that data group is currently executing. There is a controversial discussion [Boehm 2009] regarding the correct semantics for atomic blocks. Some argue that transactional semantics should be used while others argue that lock-based semantics should be used. We decided to use a lock-based approach for its simplicity of implementation and semantics. In the future we might reconsider this decision and evaluate a transactional semantics [Moore and Grossman 2008].

There exists exactly one group access token for every data group in the system and unlike runtime permissions, group access tokens *cannot* be split. In several rules the unlocked group access token context is split in a nondeterministic way. This models nondeterminism of how atomic blocks can lock data groups. Locked group access

$$\text{E-TRANS-Z}\quad \frac{}{(\mu|\delta|\Psi|\mathcal{G}|e) \mapsto (\mu|\delta|\Psi|\mathcal{G}|e)}$$

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

Fig. 12. μÆMINIUM program state transitions rules.

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

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

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

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

Fig. 13. Dynamic semantics of μÆMINIUM atoms.

tokens are forced to flow into the expression that contains the corresponding `inatomic` . This approach is not strictly necessary but allows us to formulate a stronger preservation induction hypothesis.

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

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

3.3.5. Program State. A program state is a quintuple of the form $(\mu|\delta|\Psi|\mathcal{G}|e)$, consisting of a store (μ), a runtime permission context (δ), a group access token context (Ψ) of available tokens, a data group configuration ($\mathcal{G}$), and an expression (e). A program state represents a consistent state of the execution. To transition from one program state to another, the expression takes a step following the evaluation judgment and then generates a new global store (see E-TRANS-N in Figure 12).

3.3.6. Evaluation Rules. The evaluation rules for atoms are shown in Figure 13 and the rules for expressions are shown in Figure 14 and 15. Once again we describe the most interesting rules. E-FIELD-READ demonstrates the basic approach: we look up the permissions required based on the group context $\mathcal{G}$ (which was computed by the typechecking rules), and the read cannot execute unless and until the required permission is in the permission context δ. Other atom rules are similar. The E-UNPACKGROUPSOF-* rules make the inner permissions available to the enclosed expression if and only if the permission to the outer object is available; otherwise the enclosed expression can

$$\textsc{E-UnpackGroupsOf-Replace}\quad \frac{\mathcal{G} = (\{v'.gn', \overline{v_r.gn}\} \oplus \mathcal{G}_e) \qquad \delta = \delta', v'.gn', \qquad \mu|\delta'|\Psi|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'_e \qquad \mathcal{G}' = (\{v'.gn', \overline{v_r.gn}\} \oplus \mathcal{G}'_e)}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e \mapsto\ \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e'\ \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

$$\textsc{E-UnpackGroupsOf-None}\quad \frac{\mathcal{G} = (\{v'.gn', \overline{v_r.gn}\} \oplus \mathcal{G}_e) \qquad v'.gn' \notin \delta \qquad \mu|\delta|\Psi|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'_e \qquad \mathcal{G}' = (\{v'.gn, \overline{v_r.gn}\} \oplus \mathcal{G}'_e)}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e \mapsto\ \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e'\ \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

$$\textsc{E-Let-1}\quad \frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2) \qquad \delta_1 = \delta \cap \mathit{requiredPerms}(\mathcal{G}_1) \\ \Psi = \Psi_1, \Psi_2 \qquad \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \\ \mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e'_1 \dashv \mu_\delta|\Psi'_1|\mathcal{G}'_1 \\ \mathcal{G}' = (\mathcal{G}'_1 \oplus \mathcal{G}_2) \qquad \Psi' = \Psi'_1 \cup \Psi_2\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mapsto\ \texttt{let}\ x = e'_1\ \texttt{in}\ e_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

$$\textsc{E-Let-2}\quad \frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2) \quad \delta_2 = \delta - \mathit{requiredPerms}(\mathcal{G}_1) \\ \Psi = \Psi_1, \Psi_2 \quad \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \\ \mathit{requiredTokens}(e_2) \subseteq \Psi_2 \\ \mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e'_2 \dashv \mu_\delta|\Psi'_2|\mathcal{G}'_2 \\ \Psi' = \Psi_1 \cup \Psi'_2 \qquad \mathcal{G}' = (\mathcal{G}_1 \oplus \mathcal{G}'_2)\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mapsto |\ \texttt{let}\ x = e_1\ \texttt{in}\ e'_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

$$\textsc{E-Let-12}\quad \frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2) \qquad \delta_1 = \delta \cap \mathit{requiredPerms}(\mathcal{G}_1) \qquad \delta_2 = \delta - \delta_1 \\ \Psi = \Psi_1, \Psi_2 \qquad \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \qquad \mathit{requiredTokens}(e_2) \subseteq \Psi_2 \qquad \mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e'_1 \dashv \mu_{\delta_1}|\Psi'_1|\mathcal{G}'_1 \\ \mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e'_2 \dashv \mu_{\delta_2}|\Psi'_2|\mathcal{G}'_2 \qquad \Psi = \Psi'_1 \cup \Psi'_2 \qquad \mathcal{G}' = (\mathcal{G}'_1 \oplus \mathcal{G}'_2) \qquad \mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2}\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mapsto\ \texttt{let}\ x = e'_1\ \texttt{in}\ e'_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

$$\textsc{E-Let-Value}\quad \frac{\mathcal{G} = (\bullet \oplus \mathcal{G}_2) \qquad \mathcal{G}' = [^v/_x]\,\mathcal{G}_2}{\mu|\delta|\Psi|\mathcal{G} \vdash\ \texttt{let}\ x = v\ \texttt{in}\ e_2 \mapsto [^v/_x]\,e_2 \dashv \bullet|\Psi|\mathcal{G}'} \qquad \textsc{E-UnpackGroupsOf-Value}\quad \frac{}{\mu|\delta|\Psi|\mathcal{G} \vdash\ \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ v \mapsto v \dashv \bullet|\Psi|\bullet}$$

$$\textsc{E-Split-1}\quad \frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2) \qquad \delta_1 = \delta \cap \mathit{requiredPerms}(\mathcal{G}_1) \qquad \Psi = \Psi_1, \Psi_2 \qquad \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \\ \mathit{requiredTokens}(e_2) \subseteq \Psi_2 \qquad \mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e'_1 \dashv \mu_\delta|\Psi'_1|\mathcal{G}'_1 \qquad \Psi' = \Psi'_1 \cup \Psi_2 \qquad \mathcal{G}' = (\mathcal{G}'_1 \parallel \mathcal{G}_2)\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash\ \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto\ \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e'_1 \parallel e_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

$$\textsc{E-Split-2}\quad \frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2) \qquad \delta_2 = \delta \cap \mathit{requiredPerms}(\mathcal{G}_2) \qquad \Psi = \Psi_1, \Psi_2 \qquad \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \\ \mathit{requiredTokens}(e_2) \subseteq \Psi_2 \qquad \mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e'_2 \dashv \mu_\delta|\Psi'_2|\mathcal{G}'_2 \qquad \Psi' = \Psi_1 \cup \Psi'_2 \qquad \mathcal{G}' = (\mathcal{G}_1 \parallel \mathcal{G}'_2)\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash\ \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto\ \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1 \parallel e'_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

$$\textsc{E-Split-12}\quad \frac{\begin{array}{c}\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2) \qquad \delta_1 = \delta \cap \mathit{requiredPerms}(\mathcal{G}_1) \qquad \delta_2 = \delta \cap \mathit{requiredPerms}(\mathcal{G}_2) \\ \Psi = \Psi_1, \Psi_2 \qquad \mathit{requiredTokens}(e_1) \subseteq \Psi_1 \qquad \mathit{requiredTokens}(e_2) \subseteq \Psi_2 \qquad \mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e'_1 \dashv \mu_{\delta_1}|\Psi'_1|\mathcal{G}'_1 \\ \mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e'_2 \dashv \mu_{\delta_2}|\Psi'_2|\mathcal{G}'_2 \qquad \mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2} \qquad \Psi' = \Psi'_1 \cup \Psi'_2 \qquad \mathcal{G}' = (\mathcal{G}'_1 \parallel \mathcal{G}'_2)\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash\ \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto\ \texttt{split}\ \langle\overline{v.gn}\rangle\ \texttt{between}\ e'_1 \parallel e'_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

Fig. 14. Dynamic semantics of μÆMINIUM expressions [1/2].

only take steps for which these permissions are not required. There are three variants of the let and share rules: one where the first expression takes a step, one where the second steps, and one where both expressions step (this can occur even in the sequentializing LET construct if the permissions required do not overlap). The rules for split differ in that LET divides the permissions without duplicating any, while SPLIT duplicates the permissions named in the `split` block. Finally, the rules for the `atomic` block do not pass a permission to the named data group inwards until a lock is acquired, at which point the state of the lock changes to $@L$ and the expression changes to `inatomic`

$$\textsc{E-Atomic-Step1}\quad \frac{\mathcal{G} = (\{v.gn\} \oplus \mathcal{G}_e) \qquad v.gn \notin \delta \qquad \mu|\delta|\Psi|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'_e \qquad \mathcal{G}' = (\{v.gn\} \oplus \mathcal{G}'_e)}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{atomic}\ \langle v.gn\rangle\ e \mapsto \texttt{atomic}\ \langle v.gn\rangle\ e' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

$$\textsc{E-Atomic-Step2}\quad \frac{\mathcal{G} = (\{v.gn\} \oplus \mathcal{G}_e) \qquad \delta = \delta', v.gn \qquad v.gn@U \notin \Psi \qquad \mu|\delta'|\Psi|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'_e \qquad \mathcal{G}' = (\{v.gn\} \oplus \mathcal{G}'_e)}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{atomic}\ \langle v.gn\rangle\ e \mapsto \texttt{atomic}\ \langle v.gn\rangle\ e' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

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

$$\textsc{E-InAtomic-Step}\quad \frac{v.gn \in \delta \qquad \Psi = \Psi'', v.gn@L \qquad \mathcal{G} = (\{v.gn\} \oplus \mathcal{G}_e) \qquad \mu|\delta|\Psi''|\mathcal{G}_e \vdash e \mapsto e' \dashv \mu_\delta|\Psi'''|\mathcal{G}'_e \qquad \Psi' = \Psi''', v.gn@L \qquad \mathcal{G}' = (\{v.gn\} \oplus \mathcal{G}'_e)}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{inatomic}\ \langle v.gn\rangle\ e \mapsto \texttt{inatomic}\ \langle v.gn\rangle\ e' \dashv \mu_\delta|\Psi'|\mathcal{G}'}$$

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

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

Fig. 15. Dynamic semantics of μÆMINIUM expressions [2/2].

for tracking purposes. For a more detailed description of each rule refer to Stork et al. [2010].

3.4. Proof

We prove the correctness of our system by induction on the derivation of program state transitive rules (refer to Figure 12). We prove the type *safety* following the standard approach [Pierce 2002] by proving *progress* and *preservation* separately.

Our definition of correctness means that every well-formed program is free of data races. As outlined in Section 2.2.2 ÆMINIUM currently does not handle deadlocks. Therefore a correct ÆMINIUM program, while free of deadlocks, might still have potential deadlocks.

The intuitive idea behind the proof is that to avoid race conditions at runtime our type system checks that all accesses to shared data groups are correctly protected using an `atomic` block. Accessing the same object of the heap in a conflicting manner would result in a `race` heap. Our proof shows that using ÆMINIUM's type system no such conflicting operations can occur at runtime.

3.4.1. Type Safety. We state type safety as follows: If $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$ and $(\mu|\delta|\Psi|\mathcal{G} \vdash e) \mapsto^* (\mu'|\delta'|\Psi'|\mathcal{G}'|e')$ then $\Gamma|\Sigma'|\Delta \vdash_{wf} (\mu'|\delta'|\Psi'|\mathcal{G}'|e')$ and *not* stuck. In words this means that every well-formed (refer to Definition 3.1) program state can take an arbitrary amount of steps and will result in another well-formed program state. We prove this theorem through induction by leveraging our progress and preservation lemma (refer to Sections 3.4.2 and 3.4.3).

Definition 3.1 (*Well-Formed Program State*). A program state is well typed, written as $\cdot|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$, if :

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

3.4.2. Progress. Our progress lemma is stated as follows.

LEMMA 3.2 (PROGRESS). *If* $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$ *(i.e., a well-formed program state) then either:*

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

In other words, for every well-formed program state, the expression e is either a value, or can take a step to e', caused a null pointer execption, or is waiting for being able to run (i.e., waiting until all the previous expressions it depends on have executed). We prove the correctness of our progress lemma through induction on $\Gamma|\Sigma|\Delta \vdash_C e : T\ |\mathcal{G}$ (refer to Stork [2013]).

3.4.3. Preservation. We state our preservation lemma as follows.

LEMMA 3.3 (PRESERVATION). *If* $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$ *with* $\Gamma|\Sigma|\Delta \vdash e : T\ |\mathcal{G}$ *and* $\mu|\delta|\Psi|\mathcal{G} \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'$ *and* $\mu' = [\mu_\delta]\,\mu$ *then there exists:*

— *$\Sigma' \supseteq \Sigma$;*
— *T';*

such that:

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

In other words, if we start with a well-formed program state and the expression e steps to e' we end in a well-formed program state again. We prove this lemma by induction on $(\mu|\Psi_{\mathcal{F}}|\Psi_{\mathcal{L}}|\mathcal{G}|e) \mapsto^* (\mu|\Psi'_{\mathcal{F}}|\Psi'_{\mathcal{L}}|\mathcal{G}'|e')$ (refer to Stork [2013]).

4. IMPLEMENTATION

Our implementation is based on the Plaid programming language and is publicly available in our Google Code repository [Stork et al. 2012]. The overall system architecture is shown in Figure 16. The compiler user writes *Plaid* code and feeds it into our compiler. The compiler first translates the *Plaid* source code into an *Abstract Syntax Tree (AST)*. The newly generated AST is then used by the typechecker to check that the input program does not violate Plaid's typing rules. In addition to typechecking the program, the typechecker also computes a sequential dependency graph based on the permission flow. The dependency graph design and optimizations follow the general idea of Cliff Click's *sea of nodes* [Click and Paleczny 1995] in which he replaces

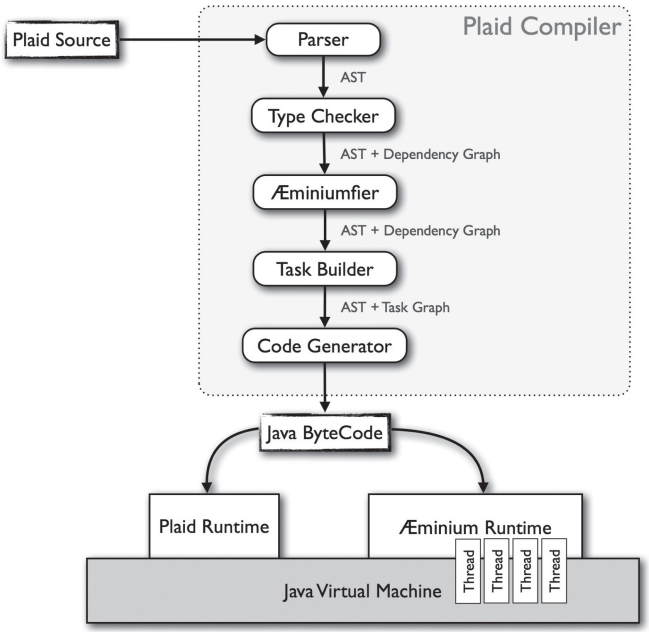


Fig. 16. System architecture.

the AST representation with a graph structure. The AST and the dependency graph are then used by the *Æminiumfier* which analyses and transforms the sequential dependency graph into a parallel dependency graph. The parallel dependency graph and the AST are then used by by the *task builder* to cluster operations into more coarse tasks. The generated task graph and AST are used by the *code generator* to generate the final *Java* bytecode.

The generated code uses the *Plaid* and ÆMINIUM runtime libraries to create and manage objects and parallelism. The *Plaid* runtime is responsible for managing states, objects, and *Java* interoperability. The ÆMINIUM runtime is responsible for managing the execution of the tasks generated by the program. The following sections elaborate on the extensions we made to the Plaid compiler.

4.1. Plaid Primer

This section provides a short introduction to the Plaid programming language, explaining all necessary constructs required for this article. Please refer to the official *Plaid* language specification [Aldrich et al. 2012] for a more in-depth overview of Plaid. By design, the *Plaid* language resembles the *Java* language as much as possible. The main conceptional difference between *Plaid* and *Java* is the usage of states instead of classes. Conceptionally, *Plaid* uses state abstractions to naturally encode the various states an object can be in a direct and checkable way. We discuss state composition and state change semantics in Sunshine et al. [2011]. An overview of *Plaid's* type system is given in Naden et al. [2012]. Those concepts are orthogonal to ÆMINIUM's parallelization approach and we therefore limit ourselves to a subset of *Plaid* which most closely resembles normal *Java*.

Listing 1 shows simple counter code emphasizing the commonalities with *Java*. In line 1 we define a new state `Object`. States, similar to *Java* classes, consist of a collection of fields and methods that operate on those fields. Instead of using the `class`

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

Listing 1. Basic Plaid Example.

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

Listing 2. Plaid Fibonacci Example.

keyword *Plaid* uses the `state` keyword to declare such a collection. As in *Java*, we call the instances of states *objects*. Line 2 shows that the `Object` state defines only one method called `toString`. *Plaid*'s method declaration follows the same syntax as a *Java* method declaration, with the following exceptions. All method declarations in *Plaid* start with the keyword `method` to indicate the start of a new method declaration. Note that *Plaid* does not support *Java's* modifiers (i.e., `public`, `final`, `abstract`, etc.) but has its own (discussed later). After the `method` keyword we have the return type of the method followed by the method name and its parameter list. After the parameter list we have the so-called *environment* of the method declared in square brackets. The environment is an implicit parameter list specifying all the variables that are implicitly passed into the method or are captured from the enclosing lexical environment. As shown in the example, the environment contains the declaration of the `this` reference. Note the additional `local` keyword in front of the `immutable` permission of the `this` reference. `local` is a permission modifier that allows the caller of a method to recover the permission passed in, without requiring the user to worry about concrete fractions (refer to Naden et al. [2012]). The `this` reference is implicitly passed into the method and therefore we need to specify which permissions we need. After the environment we usually would declare the method body in curly braces, but in

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

Listing 3. Plaid Boolean.

this case we finish the declaration with a semicolon to indicate an *abstract* method declaration.

In line 5 we define a new state `Counter` as a substate of `Object`. Plaid uses the `case of` instead of *Java's* `extends` to declare subtyping. The `Counter` defines a local field in line 6. All fields and variable declarations start with either `val` (immutable) or `var` (mutable). In lines 8 and 12 the `Counter` defines various methods to increase, decrease, or retrieve the current counter value. In *Plaid*, like in *Smalltalk* [Goldberg and Robson 1983], everything is an object. This means, unlike in *Java*, there are no primitive types (like `int`, `boolean`, etc.). The addition operation "`this.count + 1`" in line 8 is translated into a method call with the first operand as the receiver, namely "`this.count.+(1)`". This is possible because *Plaid* supports methods named after operator symbols. Another important observation is the absence of the `return` statement in *Plaid*. *Plaid* automatically returns the value of the last statement in a method body. In line 20 the `Counter` object implements the abstract `toString` method as defined by its superstate.

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

Note that in *Plaid* booleans are not part of the language and are implemented as part of the standard library. Listing 3 shows an abbreviated version of *Plaid's* boolean declaration. Line 1 defines the top-level `Boolean` type. Lines 3 and 5 define two orthogonal subtypes, one for true values and one for false values. The definition of the `Boolean` state also demonstrates *Plaid's default permission*. The state declaration is annotated with an *immutable* permission. This allows the user to omit the permission annotation for the `Boolean` type and the *Plaid* compiler will automatically extend it with default permission specified on the state declaration (in this case an *immutable* permission).

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

4.2. Typechecker Extensions

Because Plaid's typechecker already had support for access permissions, our first extension was adding support for data groups and data group permissions. The overall implementation of data groups and permissions is straightforward and analogous

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

Fig. 17. Parallelizing peephole optimizations.

to the existing implementation of access permissions (with the exception that access permissions are automatically split/merged, while group permissions are manually split/merged). The second extension we made to the typechecker was the generation of a permission flow graph. Because of Plaid's eager typechecker implementation (i.e., access permissions are merged back as soon as possible) the resulting permission flow graph does not capture all the possible parallelism. Instead of reimplementing Plaid's typechecker in a noneager way, we decided to remove the eagerness-induced sequentiality via an extra compiler pass (refer to Section 4.3).

4.3. Æminiumfier

The ÆMINIUM parallelizing pass runs directly after the typechecking pass and transforms the sequential dependency graph inferred by the typechecker into a parallel version by applying multiple *peephole optimizations* [McKeeman 1965]. A peephole optimization searches for specific patterns inside generated code (in our case the "code" is the dependency graph) and replaces those patterns by a simpler or more efficient one. The following sections explain each optimization and Figure 17 provides a short summary.

4.3.1. Simplification of Chained Splits. Typechecking follows a bottom-up approach. This leads to cases where multiple subsequent permissions can be split off the same variable before they get merged back. A simple example of such a case would be typechecking a method call where the same variable is passed multiple times as a parameter to the call. This chaining of permission splits is unnecessary and can be optimized. Instead of having a binary split node and building chains of them we simply merge those nodes to create one n-ary split node. Figure 18 illustrates this operation. The graph on top shows a chain of split nodes along with the nodes depending on them $(\delta_1, ..., \delta_{n+1})$. The optimization is applied locally to individual nodes. For every node in the graph the algorithm checks whether the current node is a split node. If it is a split node it will check if the input permission is the same as the output permission and if the current node depends on another split block. If all conditions hold the algorithm deletes the current split block from the graph while preserving its dependencies (see Figure 19).

4.3.2. Simplification of Chained Joins. Similar to chained splits, the typechecker can generate chained join nodes that merge the chained split permissions back into the original permission. Therefore the same principle can be applied and we can reduce these chains to a single join node. Figure 20 shows the approach and the algorithm. The algorithm operates on individual nodes. It first selects all join nodes. Then for every join node the algorithm checks whether the node joins the input permission into

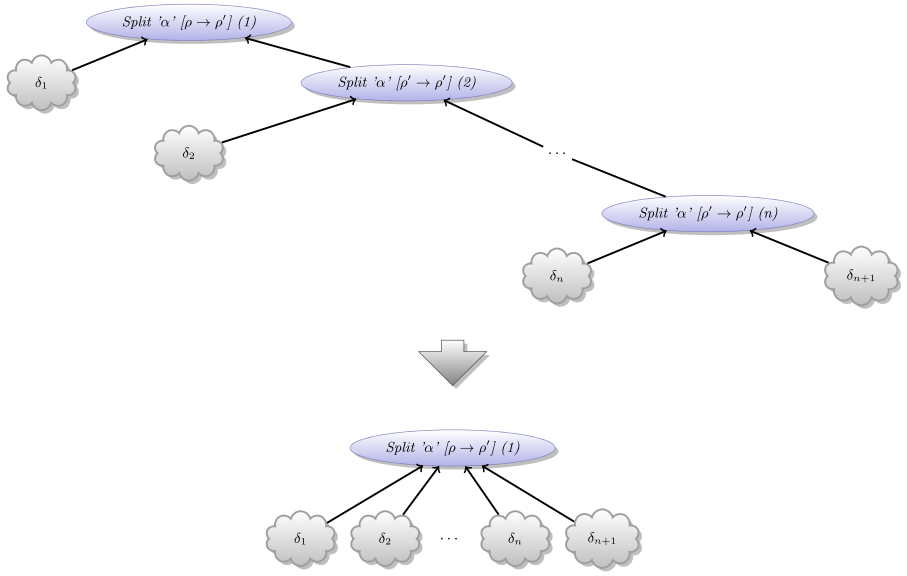


Fig. 18. Chained split block optimization.

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

Fig. 19. Node delete operation.

the same kind of permission. If the node does, the algorithm checks if there is any other join node depending on the current node. If all conditions hold the algorithm deletes the current node, again while preserving dependencies.

4.3.3. Simplification of Unique Split/Join Sequences. The typechecker may sometimes need to split off a `unique` permission from a variable, leaving a `none` permission associated with the variable. Later, when the `unique` permission is returned to the variable, the typechecker merges the incoming `unique` permission with the available `none` permission. This is a typical scenario for method calls where the permission gets conceptually split off from the variable and later (after the method call) merged back. Figure 21 shows the scenario on the left-hand side where a unique permission from α has been split off to satisfy the operations $\overline{\delta}_2$. Figure 21 also shows the algorithm to implement this optimization, which simply removes those unnecessary nodes.

4.3.4. Simplification of Symmetric Join/Split Sequences. The current version of the typechecker implements a greedy approach for merging permissions back. For every operation, the greedy approach splits off the required permissions and joins them back as soon as they become available again (i.e., the operation completes). This leads to the problem that if two operations require a symmetric permission the typechecker creates unnecessary dependencies.

To solve this issue we want to detect such unnecessary join/split patterns and eliminate them such that both operations can operate in parallel. Figure 22 shows how we

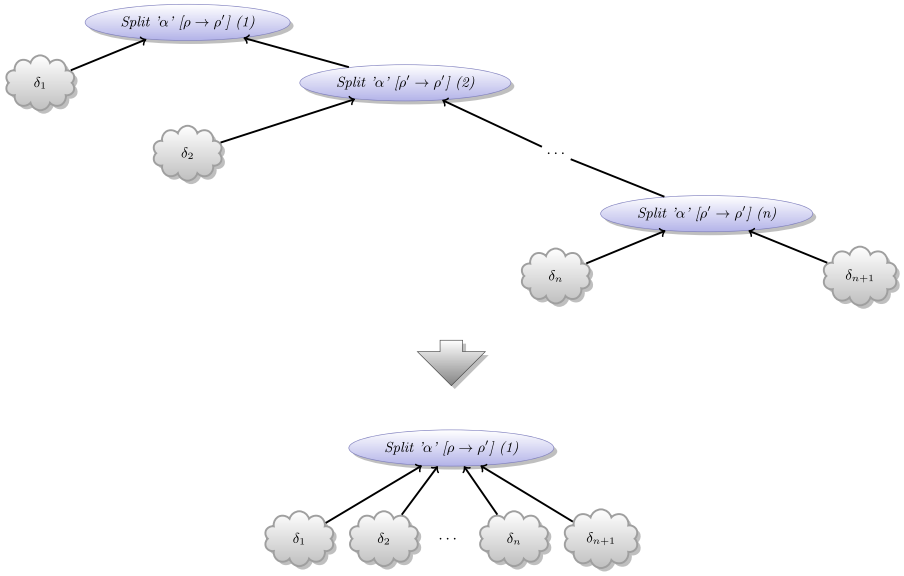


Fig. 20. Chained join block optimization.

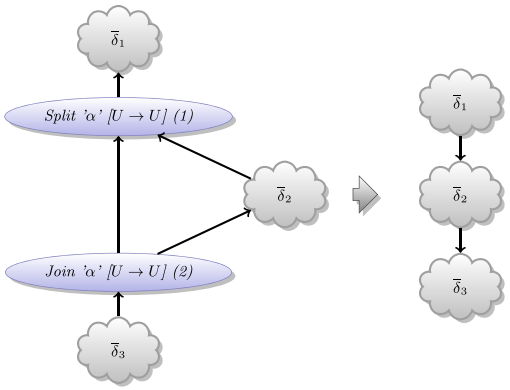


Fig. 21. Simplify unique join/split sequences.

remove those inner join/split nodes and reorganize the graph so that we initially split multiple symmetric permissions off the original permission and execute the operations in parallel.

4.4. Taskbuilder

Generating a new task for every node in the dependency graph (i.e., one task per operation) is prohibitively expensive because the ratio of task work to task creation overhead is too small. Therefore, we developed the *Taskbuilder Pass*, which combines multiple operations into bigger tasks. Figure 23 shows the basic idea. The taskbuilder takes as input a dependency graph (see Figure 23(a)) and then computes which operations can be mapped into the same task without losing parallelism. Figure 23b shows the input graph with the task clustering. The taskbuilder outputs a graph consisting only of tasks (see Figure 23(b)).

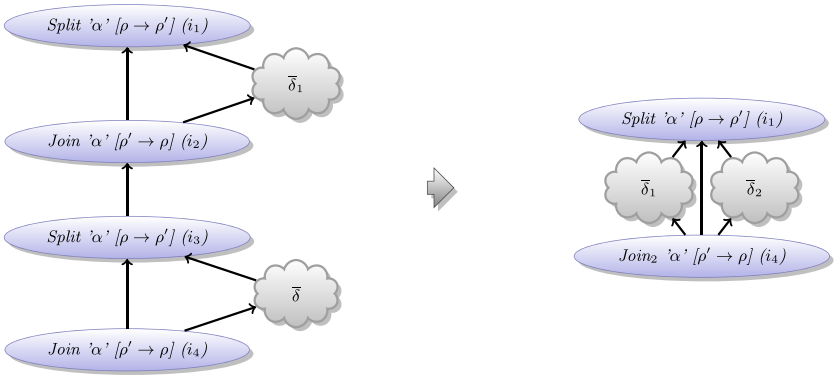


Fig. 22. Symmetric join/split optimization.

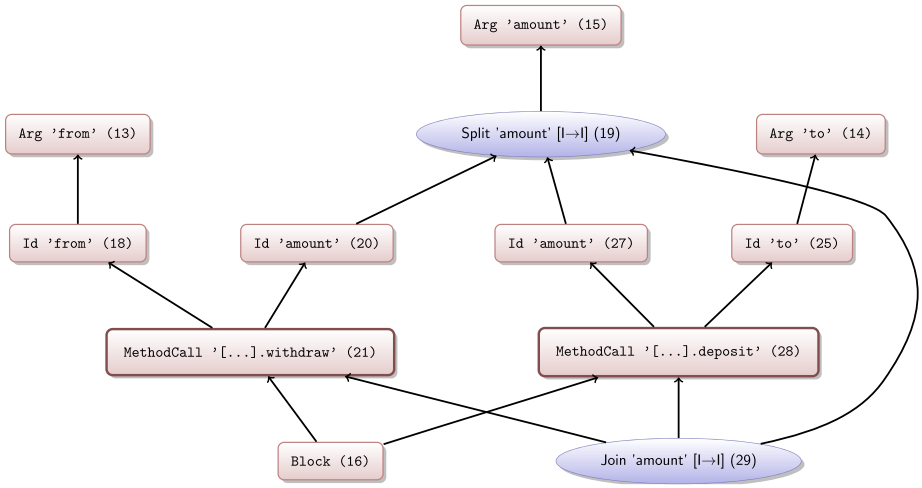
The general idea behind the taskbuilder is called *edge zeroing*. The taskbuilder uses a cost metric to estimate the overall execution costs of a specific dependency graph. The algorithm then analyses, for every edge in the dependency graph, how removing the edge and merging the connecting nodes would affect the execution cost of the whole graph. If the execution cost does not increase, the taskbuilder removes the current edge from the graph and merges together the nodes formerly connected by that edge. The following sections explain the taskbuilder in more details.

Our taskbuilder algorithm is based on *Sarkar's Algorithm* (SA, [Sarkar 1989]). To work properly, SA needs to know the runtime costs for every operation in the graph. This cost can be easily estimated for all operations except method calls. To enable SA to perform more aggressive optimizations, we provide a simple categorization of the methods. We differentiate between normal methods and *cheap* methods. Cheap methods, defined via `cheap` annotations on their declarations, are relatively short in their execution and do not justify the creation of parallelism by themselves. We prefer annotations to inference for modularity reasons, but the compiler verifies that methods annotated as cheap call only other cheap methods. Other static [Blelloch and Greiner 1996] or dynamic [Acar et al. 2011] approaches to determine runtime costs have been proposed and are generally applicable to our system.

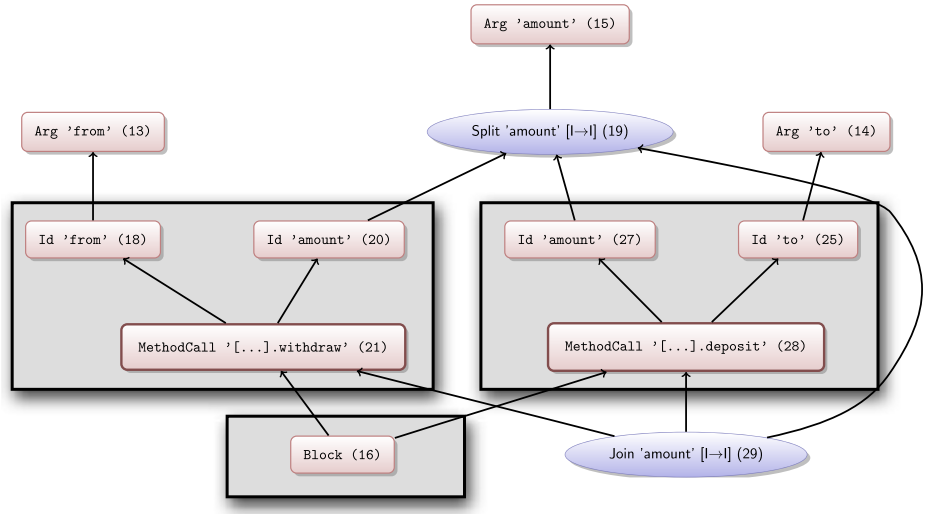
4.5. Code Generator

While the taskbuilder tries to minimize the number of tasks, there are still a few optimizations that can be performed during code generation to further reduce the number of created tasks. The following sections present several optimizations that can help in this regard (refer to Figure 24 for a summary overview). We discuss each optimization separately to focus on its core idea. We present all optimizations in the context of method calls, but notice that the optimizations are also applicable to optimizing other constructs, such as case statements in a match block. To focus on the optimization techniques, and for brevity reasons, we use the generic scheduling algorithm as a basis for our extensions when we present those optimizations.

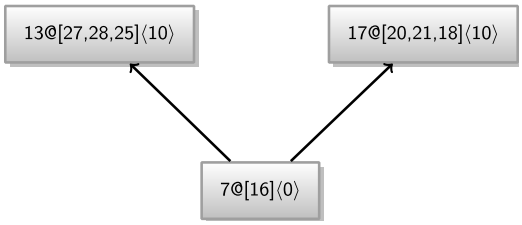
Sequentializing Single Task Graphs. If the taskbuilder manages to reduce the task graph of a whole method body to a single task, then code generation will inline this task. This results in the generation of a sequential method body, equivalent to the sequential method body that would have been generated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



(a) dependency graph



(b) dependency graph with task overlay



(c) task graph

Fig. 23. Taskbuilder approach.

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

Fig. 24. Overview of code generation optimizations.

```
method PlaidObject m(...) {
    τ@[δ̄] ⟨ω⟩
}
```

Listing 4. Single Task Function Graph.

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

Listing 5. Single Task Function Code.

```
method PlaidObject m(...) {
    τ̄\τb
    τb
}
```

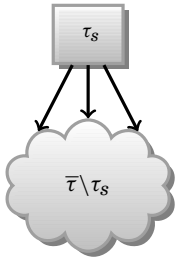
Listing 6. Inline Body Task Graph.

```
public PlaidObject m(...) {
    // create variables
① PlaidObject[] _ = new PlaidObject[|{VarDecl(x) ∈ {δ̄ : τ@[δ̄] ⟨ω⟩}}|];

    // create task objects
② ∀τi ∈ {τ̄\BODY_TASK(τ̄)} : Task Tτi = new Task(|DEPS(τi)|) {
        public void run() {
            IS_CASE_TASK(τi) ⟹ if ( CASE_MATCH_COND(τi) ) { δ̄τi }
            ¬IS_CASE_TASK(τi) ⟹ δ̄τi
            ∀τ' ∈ RDEPS(τi) : if ( Tτ'!= BODY_TASK(τ̄) &&
                                   Tτ'.decDepCount() == 0 ) {
                schedule(Tτ');
            }
        }
    };

    // compute dependencies and schedule tasks
③ ∀τi ∈ START_TASKS(τ̄) : schedule(Tτi);

    // wait for dependencies of the body task to finish
④ ∀τi ∈ DEPS(BODY_TASKS(τ)) : Tτi.wait();

⑤ return δ̄BODY_TASKS(τ);

}
```

Listing 7. Inline Body Task Code.

Inlining Body Tasks. Because the method always has to wait for the main body task to complete, we can inline this task into the method body and avoid the creation and synchronization overhead for this task. Listing 6 shows our code generation strategy, which is comprised of the following steps.

① **Variable extraction.** No changes.
② **Task Creation.** We create all tasks except the body task.
③ **Task scheduling.** No changes.
④ **Wait for dependencies.** Wait for all tasks the body task depended on to complete.



Listing 8. Inlining Start Task Graph.

```
public PlaidObject m(...) {
  // create variables
① PlaidObject[] __ = new PlaidObject[|{VarDecl(x) ∈ {δ̄ : τ@[δ] ⟨ω⟩}}|];

  // execute start task code
② δ̄START_TASK(τ̄)

  // create task objects
③ ∀τi ∈ {τ̄\START_TASK(τ̄)} : Task Tτi = new Task(|DEPS(τi)|) {
      public void compute() {
        IS_CASE_TASK(τi) ⟹ if ( CASE_MATCH_COND(τi) ) { δ̄τi }
        ¬IS_CASE_TASK(τi) ⟹ δ̄τi
        ∀τ' ∈ RDEPS(τi) : if ( Tτ'.decDepCount() == 0) { schedule(Tτ'); }
      }
  };

  // compute dependencies and schedule tasks
④ ∀τi ∈ RDEPS(START_TASKS(τ̄)) : schedule(Tτi);

  // wait for dependencies of the body task to finish
⑤ return TBODY_TASK(τ̄).wait();
}
```

Listing 9. Inlining Start Task Code.

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

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

① **Variable extraction.** No changes.
② **Execute start task code**. Execute the operations associated with the start task directly in method body.
③ **Task Creation.** We create all tasks except the start task.
④ **Task scheduling.** Schedule all start tasks which depend on the original start task.
⑤ **Wait for body task.** No changes.

4.5.1. Dynamic Load Balancing. Despite the optimizations discussed earlier, our system can produce significantly more tasks than we have parallel execution units. To eliminate the high costs of task creation and scheduling we implemented the dynamic load-balancing approach shown in Listing 10. Every method that supports parallel execution first performs a check whether we have enough parallelism (i.e., enough generated tasks to utilize the available computation units) or not by calling the `PARALLELIZE` method. If this method returns `false` it means that we have enough work and should not generate new work. In this case we simply execute the sequential method body instructions. If the return value is `true` we need to generate more parallel work and we execute the parallel method body implementation as described earlier.

The `PARALLELIZE` method implementation checks whether there are threads without work. Because we call the `PARALLELIZE` method on every method invocation, determining all the threads' current state is prohibitively expensive. To overcome this problem we guard the check with a global variable estimating the lack of parallel work. This global variable is updated when threads create new tasks and when threads are running out of work. To further optimize runtime overhead, all accesses to this variable are not synchronized. The lack of synchronization obviously leads to race conditions and lost updates. In the scheme we apply when updating the variable, lost updates

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

Listing 10. Dynamic Load Balancing.

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

Listing 11. Atomic Block Translation.

only ever lead to the creation of additional tasks and never to starving threads (refer to our implementation for the exact details).

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

4.5.2. Atomic Block Implementation. Our implementation allows seamlessly mixing code with and without data groups. If we use code without data groups we are talking about plain *shared* permissions and atomic blocks without any data group parameters. In this data-group-less mode we implicitly pass a share data group permission to an anonymous global data group into every method. Figure 11 shows that we simply translate an atomic block into an `enterAtomic` and `leaveAtomic` method call on the corresponding data group. Once we entered a global atomic block we decided for simplicity reasons to sequentialize the execution of its body. This means that when we call a method from inside a global atomic block this method needs to execute sequentially even if it could execute in parallel. There are two approaches to achieve this behavior. The first option is to have a dynamic check at runtime to force sequential execution. The second option is to have two versions of every method: one version that is called by default and another version that can only be called from inside an atomic block directly or transitively (refer to AtomJava [Hindman and Grossman 2006]). We decided to go for the dynamic approach because it can be easily merged with dynamic load balancing and avoids code explosion. Listing 12 shows the implementation of the global atomic block sequentializing check.

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

4.6. Implementation Reflection

The goal of our implementation was to be as fast as possible. During our initial experiments it became quite obvious that creating and executing fine-grained tasks on a

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

Listing 12. Global Atomic Test.

large scale was prohibitively expensive. Therefore our goal was to eliminate as many tasks as possible. In our experience, the *load-balancing* method resulted in the most dramatic reduction in number of tasks. The dynamic load-balancing approach only generates as many tasks as needed to utilize system resources. Despite this fact, all the other optimizations we described play an important part in our achieved performance (refer to Section 5). Those optimizations are important as they help to reduce the number of tasks and increase the overall task size (which helps to counteract the task switching cost). Like so many other cases, it is not a single optimization but rather a combination of several that results in the best possible performance.

5. EVALUATION

We evaluated our system by conducting several case studies of which we present only a selection in this section. The remaining case studies can be found in Stork [2013].

Inspired by the *Problem-Based Benchmark Suite*[1] we developed a dictionary benchmark to evaluate the effectiveness of data groups. Our implementation[2] is based on a hash table using separate chaining to handle collisions. We developed two versions, a *global* version which uses plain *shared* permissions for its internal data structures and a *fine* version in which every bucket has its own data group.

We evaluated two use cases, one in which we have a *unique* permission to the dictionary and one in which we have a *shared* permission. Our benchmark first inserts the identity mapping for the numbers 2^0 to 2^{16} into the dictionary (initialization). Then we look up every mapping to check for correctness (checking). We run each benchmark case 50 times on an eight-core SMP system (using *Intel Xeon* X5460 CPUs) with 16GB of memory running *Fedora* 7 using the Java HotSpot 64-bit Server VM (build 20.4-b02). We used a dictionary with 64 hash buckets. To avoid artificial patterns we randomized the sequence in which the numbers are inserted/checked with a constant seed to guarantee reproducibility.

Figure 25 shows the results of our dictionary benchmark. The first bar "global/unique" (15.12*s*) represents the results of the *global* dictionary implementation with a *unique* permission to the dictionary. The linearity of the *unique* permission sequentiallizes all insert/check operations. In the second bar "global/shared" (15.13*s*) we have a *shared* permission to the dictionary, which allows us to perform our operations in parallel. This case performs no better because each parallel operation must immediately synchronize on the entire *shared* dictionary structure, thus sequentializing all the accesses. The third bar "fine/unique" (9.99*s*) uses the implementation which utilizes data groups for its internal representation. This scenario is faster than any of the cases using the global implementation, because of the use of fine-grained data groups, one for each bucket. The *unique* receiver permission allows us to get *exclusive* group

[1]http://www.cs.cmu.edu/∼pbbs/
[2]http://goo.gl/nzvLd

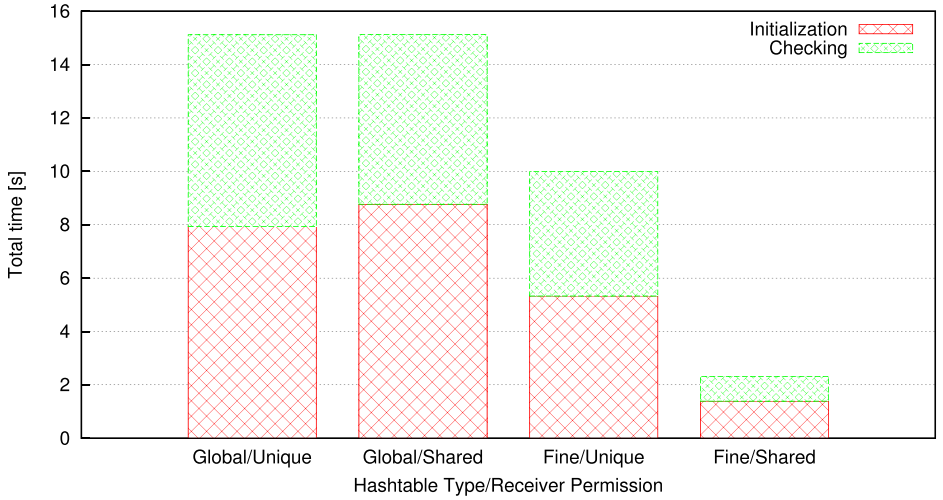


Fig. 25. Dictionary benchmark results.

permissions to the inner groups of the dictionary. This means we do not require protection to access data within those data groups and therefore we avoid unnecessary synchronization operations. The last case "fine/shared" (2.32*s*) also allows the parallel execution of our operations. Because the implementation associates each bucket with its own data group, we achieve a very fine-grained protection mechanism which allows the parallel modification of disjoint parts of the dictionary. This results in a speedup of 6.5*X* compared to the "global/shared" version.

The second case study we present consists of a Web server application[3]. We compiled the Web server in two ways. First we compiled it as a plain Plaid program (resulting in a sequential program) and second we compiled it with ÆMINIUM enabled. As a control we implemented equivalent *Java* versions (sequential and parallel). We hosted the Web server in a quad-core machine (*Intel Core 2* Q6600 with 4GB of memory running *Ubuntu 11.04* and using the OpenJDK 64-bit Server VM (build 20.0-b11)), serving the *Python* 2.7 documentation[4]. We mirrored the whole documentation three times to our local machine using the `puf`[5] tool. The `puf` tool uses up to 20 connections to parallelize the file downloads and therefore allows us to emulate multiple clients.

Figure 26 shows the average performance values measured. The *Plaid* version of the Web server is the slowest (49.1*s*) followed by the sequential *Java* version (48.5*s*). This makes sense as *Plaid* is generally slower than *Java*. The ÆMINIUM-compiled version of the Web server is the second fastest (37.4*s*) version. It is approximately 31% faster than its sequentially compiled counterpart. The reason for this is that the Web server in the ÆMINIUM-compiled version is able to handle multiple requests in parallel. This allows the overlapping of communication and computation and results in a higher throughput. The manually parallelized *Java* version delivered the best performance (31.2). The performance difference between the parallelized *Java* and the ÆMINIUM version is bigger compared to their sequential counterparts. This effect is caused by the parallel execution and the overlap of communication and computation which hides the communication costs to some degree. Because the communication effect is reduced,

[3]http://goo.gl/rU3P2
[4]http://docs.python.org/
[5]http://puf.sourceforge.net/

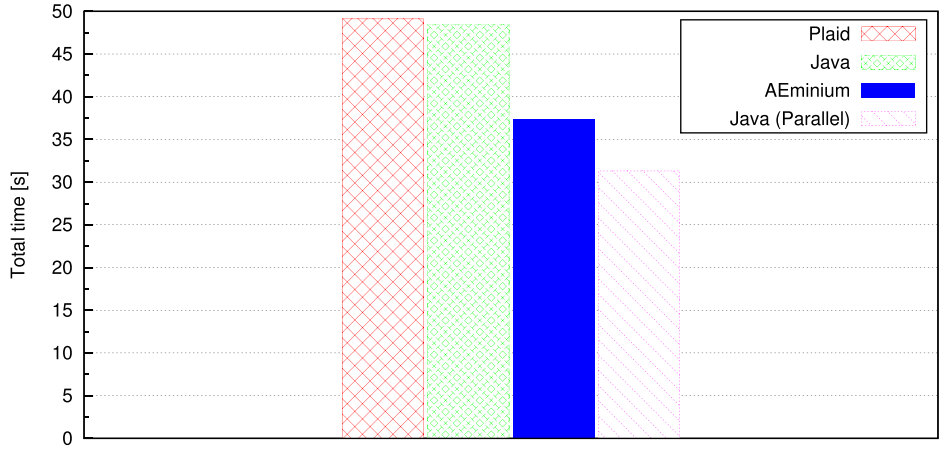


Fig. 26. Web server benchmark results.

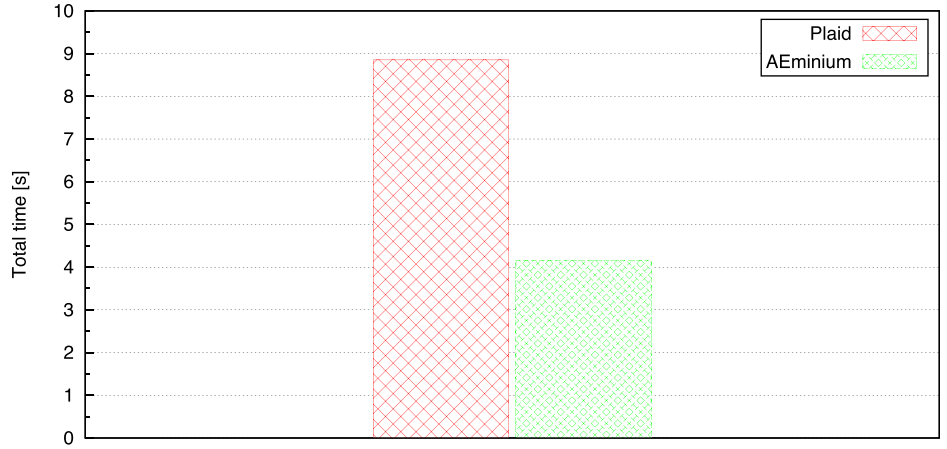


Fig. 27. Integral performance graph.

the computation part gains relatively more weight, with the result that lower base performance of the Plaid programming language has a greater impact.

In our *integral* case study we investigated ÆMINIUM's capabilities to parallelize purely functional, highly computation-intensive problems. We developed a small integral library which computes the integral of a user-defined function. The integral is computed by subdividing the overall interval into infinitesimally small intervals for which we calculate the approximate area, and then add up all fractions to compute the area of the whole integral. We evaluated the performance by computing the integral of the square function (i.e., $f(x) = x^2$) for the interval $[0, 1]$. We run the sequential *Plaid* and parallel ÆMINIUM version on our eight-core machine each 20 times. The average runtime and standard deviation of both cases are shown in Figure 27. The *Plaid* version requires $8.9s$ while the ÆMINIUM version needs only $4.2s$. This results in a speedup of 2.1 meaning that ÆMINIUM was able to parallelize the program and achieve some performance improvements. But it also means that the ÆMINIUM version was only twice as fast on an eight-core machine, which would suggest a speedup closer to eight. Our investigation revealed that the main source for this poor performance lies in the *Plaid*'s object system. As described previously, *Plaid* does not

Program	Total SLOC	Annot. SLOC	Type Annot.	Group Arg.	ÆMINIUM Constr.
webserver*	227	47 (20.7%)	59	0	0
dic/global*	169	41 (24.2%)	65	0	3
dic/fine*	251	71 (28.3%)	109	10	2
Total*	**647**	**159(24.6%)**	**233**	**10**	**5**
webserver†	227	0 (0.0%)	0	0	0
dic/global†	169	5 (3.0%)	2	0	3
dic/fine†	251	41 (18.3%)	41	10	2
Total†	**647**	**52(7.9%)**	**43**	**10**	**5**

Fig. 28. Annotation overhead over Java.

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

We evaluated our annotation overhead by comparing our ÆMINIUM programs to their equivalent *Java* versions. We counted how many lines of the source code (SLOC, measured with `wc`) we had to modify by: annotating types (i.e., add permission information to types), how often we had to specify additional group parameters to method calls, and how many ÆMINIUM-specific operations we used (e.g., `atomic` blocks). Figure 28 shows the numbers for the case studies we presented. The values marked with "$*$" are versions fully annotated and values marked with "†" are programs which use Plaid's default permission mechanism which allows omitting the permission annotation by specifying a default permission in the state declaration. This allows the compiler to automatically insert a default permission wherever the user did not specify a permission explicitly (e.g., in Java all strings are immutable by design and therefore the default permission for strings could be `immutable`, which allows the user to simply write `String` instead of `immutable String` when he specifies a string type). The numbers show that type annotations are the most common source of overhead and that Plaid's default permission helps to reduce it. The second important observation is that the more developers specify, the more performance the compiler can achieve. This means users can start with a simple version of a program and then incrementally add more annotations to increase the performance. It is worth pointing out that using Plaid's default permission approach we are able to extract concurrency in the Web server example without the need for *any* additional annotations. Overall we achieve a reasonable 7.9% annotation overhead which is comparable to the 10.7% reported by DPJ [Bocchino et al. 2009]. Further improvements to our system (e.g., type inference) should allow us to further mitigate the programmer's burden. The reader should also take into account that the access permission information in Plaid serves additional purposes (e.g., checking typestate).

6. FUTURE WORK

While our current prototype system demonstrated the potential of our approach, it has a few shortcomings we would like to address in future versions. The following

paragraphs elaborate the most interesting and useful directions for future extensions.

Permissions and data groups provide a nice abstraction for many situations but there are corner cases in which they can be cumbersome or not sufficient. For instance, the programmer may want to impose an order on two operations, perhaps because the operations have effects that are not currently captured in our permission system (e.g., I/O). In this case the programmer would have to write "ghost permissions" that represent the effect. Another situation would be when the user has to write multiple versions of the the same method for different permission configurations. To solve these issues, investigation into refined permission abstractions is necessary. These new abstractions should not only allow more fine control by the programmer, but also allow the compiler to infer permissions and implementations when possible.

Our current implementation does not support global state. While global state is generally considered a bad thing, there are situations where it is extremely convenient. An example may include using I/O methods such as `println\printf` that rely on global state to access the standard output device.

In the current system we use static costs for the operations and method calls. We already distinguish between cheap and heavy functions in order to optimize the task graph. One way to improve this approach would be be to use an aggressive static analysis to try to prove a bound on method costs. Another, and more promising, approach would be to have a Just-In-Time (JIT) version of our compiler. This JIT would analyse the cost of functions at runtime and then optimize code depending on the gathered profiling information.

7. RELATED WORK

Deterministic Parallel Java (DPJ, [Bocchino et al. 2009]) is a parallel programming language with deterministic-by-default semantics. DPJ uses *regions* (which correspond to ÆMINIUM's data groups) to partition the store and provides explicit fork-join parellelism. DPJ has special language constructs (e.g., for loops, cobegin blocks, etc.) which allow parallel execution of statements that do not interfere with each other. Code outside those constructs executes sequentially. DPJ recently added support for race-free nondeterministic parallelism as well [Bocchino et al. 2011].

The most significant difference between ÆMINIUM and DPJ is that programmers in ÆMINIUM think and write code with permissions in mind. Parallelism in ÆMINIUM is implicitly inferred based on the permission flow of those permissions. Implicit parallelism means that ÆMINIUM programs are not tied to a particular amount or granularity of parallelism specified by the programmer; instead, the runtime is free to adapt to the parallelism available in the underlying hardware. Likewise, the runtime can parallelize a library, or not, depending on whether the client is already taking advantage of parallel resources.

On a technical level, our implicit parallelism uses a dataflow model, which can in some programs capture more parallelism than can be expressed in DPJ's fork-join model (refer to Section 2.4) . This dataflow computation makes our formal system quite different than prior fork-join or thread-based type systems. Our split block (developed independently of DPJ's nondeterminism; see Stork et al. [2010]) also differs conceptually from DPJ's nondeterministic parallelism construct: it does not specify that code executes in parallel, but rather that two blocks of data can be accessed independently without affecting (high-level) program semantics. Finally, Plaid's permissions and data groups are tied to individual objects, in contrast to DPJ's globally declared regions; our design is more object based, and helps express idioms such as uniqueness that are not supported in DPJ.

Craik and Kelly [2010] describe a system which uses ownership information to automatically parallelize code in a dataflow style. Craik's ownership contexts are similar

to ÆMINIUM's data groups, but they do not have the concept of *unique* or *immutable* permissions. Their system supports only deterministic parallelism. While they provide an argument for soundness, our formal model goes further in incorporating a small-step operational semantics model of parallelism and a rigorous progress/preservation proof approach.

The *FX* programming language [Gifford and Lucassen 1986] uses an implicit dataflow approach similar to ÆMINIUM. The FX language classifies every expression into one the following four categories: *producer* (i.e., can read, write, and allocate memory), *observer* (i.e., can read and allocate memory) *function* (i.e., can allocate memory), and *pure* (i.e., side-effects free). Based on the effects of each expression the system can compute a dataflow graph based on the interference on the global heap and extract concurrency. Compared to ÆMINIUM, FX only supports deterministic parallelism and computes interference using effects with a global granularity rather than fine-grained data groups.

Data-Centric Synchronization (DCS) [Vaziri et al. 2010] is an explicitly parallel system where synchronization is expressed by associating object fields with *atomic sets*. Each method declares which atomic sets it accesses and the runtime system inserts synchronization to ensure that no methods with conflicting atomic sets will be executed at the same time.

Fortress [Allen et al. 2008] has concurrent-by-default evaluation semantics for some language constructs (e.g., loops). When the programmer uses these constructs, she is indicating that it is safe to parallelize execution. ÆMINIUM takes this concurrent-by-default principle and applies it to the whole language, not just a few language constructs. Furthermore it provides a type system for controlling parallelism according to dependencies which, in the case of Fortress, might be missed by the programmer, causing errors.

ÆMINIUM's dataflow parallelism generalizes fork-join parallelism, which was notably supported by *Cilk* [Blumofe et al. 1995]. Cilk extends C with three additional keywords for explicit parallelism: `cilk`, `spawn`, and `sync`. Every method annotated with `cilk` can be asynchronously spawned-off with the `spawn` keyword. `sync` keyword is used to wait for a previously started asynchronous task. ÆMINIUM essentially attempts to infer spawn and sync points based on typed dependencies, and can also capture more general dataflow patterns of parallelism.

Axum (formerly known as *Maestro*) [Microsoft Corporation 2009] is an actor-based programming language. Axum comes with several operators to allow the explicit construction of dataflow graphs, which can be 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. Axum and ÆMINIUM share similar concepts, in particular the dataflow approach, and the use of data groups/domains combined with explicit access specifications.

Boyapati et al. [2002] describe an explicitly concurrent extension to Java that associates each object with an *owner* (related to our data groups), and checks that the owner is locked before accessing the object. Deadlocks are also prohibited via a lock ordering protocol.

Athapascan-1 [Galilée et al. 1998] is a language that dynamically computes and uses a dataflow graph to execute the code. In Athapascan-1 the user writes tasks which can be asynchronously spawned off. Tasks are annotated with information about which shared data they access and in which way. The semantics of Athapascan-1 preserves the deterministic result of execution and can roughly been seen as a dynamic version of DPJ. Compared to ÆMINIUM, Athapascan-1 uses a dynamic approach while ÆMINIUM uses a static approach for computing the dataflow graph.

SharC [Anderson et al. 2008] is a data race checker for *C* programs. SharC uses a lightweight type annotation system which bears some resemblance to ÆMINIUM's permission and data group approach. SharC has *private* and *read-only* annotations which compare to ÆMINIUM's *unique* and *immtable* permissions. In SharC, all shared data accesses need to be marked with an *locked(lock)* indicating which lock needs to be held before accessing the corresponding data. This resembles ÆMINIUM's shared permissions associated with data groups. To allow for more flexibility, SharC uses on top of a static typesystem additionally dynamic runtime checks. Unlike ÆMINIUM, SharC is a checker only and can only check that a user-parallelized program is accessing its state in a safe manner.

The biggest differentiator for ÆMINIUM is that while nearly all the systems mentioned already have explicit parallel programming constructs or libraries, in the case of ÆMINIUM code executes in parallel by default, to the extent allowed by permission dependencies. Compared to the implicitly parallel models in FX and Craik et al., ÆMINIUM supports a richer set of permissions that enables expressing the programs from our case studies.

8. CONCLUSION

We presented ÆMINIUM, an automatic parallelization methodology with type-based safe deterministic and nondeterministic concurrency. ÆMINIUM uses the *permission flow* and *data groups* to automatically parallelize code and supports *dataflow* and *fork-join* parallelism. We further presented μÆMINIUM, a core calculus for the concurrent-by-default programming language ÆMINIUM along with its soundness proof. We presented our initial prototype implementation and several case studies showing the benefits and applicability of the ÆMINIUM concept to selected use cases. The ÆMINIUM approach is modular, composable, incremental, and provably avoids race conditions. The fundamental concept of ÆMINIUM is generally applicable and not limited to object-oriented languages. With ÆMINIUM programmers can focus on the core functionality of their applications by shifting concerns about race conditions and parallelization to ÆMINIUM.

REFERENCES

Acar, U. A., Charguéraud, A., and Rainey, M. 2011. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'11)*.

Adve, S. V. and Boehm, H.-J. 2010. Memory models: A case for rethinking parallel languages and hardware. *Comm. ACM 53*, 8, 90–101.

Aldrich, J., Sunshine, J., Saini, D., and Sparks, Z. 2009. Typestate-oriented programming. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'09)*.

Aldrich, J., Beckman, N. E., Bocchino, R., Naden, K., Saini, D., Stork, S., and Sunshine, J. 2012. The plaid language: Typed core specification. Tech. rep. CMU-ISR-12-103, Carnegie Mellon University.

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J., Ryu, S., Steele Jr, G., and Tobinhochstadt, S. 2008. The fortress language specification version 1.0. Tech. rep., Sun Microsystems.

Anderson, Z., Gay, D., Ennals, R., and Brewer, E. 2008. Sharc: Checking data sharing strategies for multithreaded C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM Press, New York, 149–158.

Beckman, N. E., Bierhoff, K., and Aldrich, J. 2008. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'08)*.

Blelloch, G. E. and Greiner, J. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*. 213–225.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*.

Bocchino, Jr., R. L., Adve, V. S., Dig, D., Adve, S. V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'09)*.

Bocchino, Jr., R., Heumann, S., Honarmand, N., Adve, S., Adve, V., Welc, A., and Shpeisman, T. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. 535–548.

Boehm, H.-J. 2009. Transactional memory should be an implementation technique, not a programming interface. Tech. rep. HPL-2009-45, HP Laboratories.

Boyapati, C., Lee, R., and Rinard, M. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'02)*. 211–230.

Boyland, J. 2003. Checking interference with fractional permissions. In *Proceedings of the 10th International Symposium on Static Analysis*.

Clarke, D. G., Potter, J. M., and Noble, J. 1998. Ownership types for flexible alias protection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*. 48–64.

Click, C. and Paleczny, M. 1995. A simple graph-based intermediate representation. In *Papers from the ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*. ACM Press, New York, 35–49.

Craik, A. and Kelly, W. 2010. Using ownership to reason about inherent parallelism in object-oriented programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, and the International Conference on Compiler Construction (CC'10/ETAPS'10)*. 145–164.

Fahndrich, M. and Deline, R. 2002. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. Vol. 37, ACM Press, New York, 13–24.

Galilée, F., Cavalheiro, G. G., Louis Roch, J., and Doreille, M. 1998. Athapascan-1: On-line building data flow graph in a parallel language. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 88.

Gifford, D. K. and Lucassen, J. M. 1986. Integrating functional and imperative programming. In *Proceedings of the ACM Conference on LISP and Functional Programming (LFP'86)*. 28–38.

Girard, J.-Y. 1987. Linear logic. *Theor. Comput. Sci. 50*, 1.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing, Boston, MA.

Hindman, B. and Grossman, D. 2006. Atomicity via source-to-source translation. In *Proceedings of the Workshop on Memory System Performance and Correctness (MSPC'06)*. ACM Press, New York, 82–91.

Igarashi, A., Pierce, B. C., and Wadler, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'01)*.

Leino, K. R. M. 1998. Data groups: Specifying the modification of extended state. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*.

Leino, K. R. M., Poetzsch-Heffter, A., and Zhou, Y. 2002. Using data groups to specify and check side effects. *ACM SIGPLAN Not. 37*, 5, 246–257.

McKeeman, W. M. 1965. Peephole optimization. *Comm. ACM 8*, 443–444.

Microsoft Corporation 2009. Axum programmer's guide. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx.

Moggi, E. 1991. Notions of computation and monads. *Inf. Comput. 93*, 1, 55–92.

Moore, K. F. and Grossman, D. 2008. High-level small-step operational semantics for transactions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*.

Naden, K., Bocchino, R., Aldrich, J., and Bierhoff, K. 2012. A type system for borrowing permissions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM Press, New York, 557–570.

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

Rumbaugh, J. 1975. A parallel asynchronous computer architecture for data flow programs. Ph.D. thesis, MIT-LCS-TR-150, MIT.

Sarkar, V. 1989. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA.

Stork, S. 2013. ÆMINIUM- Freeing programmers from the shackles of sequentiality. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Stork, S., Aldrich, J., and Marques, P. 2010. Micro-AEmimium language specification. Tech. rep. CMU-ISR-10-125R2, Carnegie Mellon University.

Stork, S., Marques, P., and Aldrich, J. 2009. Concurrency by default: Using permissions to express dataflow in stateful programs. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. 933–940.

Stork, S., Naden, K., and Sunshine, J. 2012. AEminium code repository. http://goo.gl/olbMs.

Sunshine, J., Naden, K., Stork, S., Aldrich, J., and Tanter, E. 2011. First-class state change in plaid. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'11)*. ACM Press, New York, 713–732.

Vaziri, M., Tip, F., Dolby, J., and Vitek, J. 2010. A type system for data-centric synchronization. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'10)*.

Received October 2012; revised June 2013; accepted October 2013