μÆMINIUM Language Specification

Sven Stork **Jonathan Aldrich** **Paulo Marques**

February 2012
CMU-ISR-10-125R2

This report updates CMU-ISR-10-125R to reflect the renaming of share blocks to split blocks.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

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 paper we present μÆMINIUM: a core calculus for 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, provably preventing data races. 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.

This work was partially supported by the Portuguese Reearch 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).

Keywords: programming languages, concurrency, access permissions

Contents

List of Figures

List of Tables

1 Introduction

In recent years concurrency has spread to all areas of software engineering and system design. These systems range from high performance computers to ordinary laptops, smart phones and even embedded systems. The concurrency models used by applications running on these systems are widely different. They include parallel number crunching to task synchronization, and inter-thread communication to I/O and latency hiding.

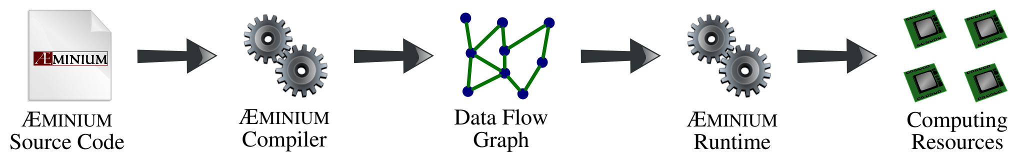


Figure 1: The ÆMINIUM Approach

The problem of concurrency cannot be successfully solved without having software engineering concerns in mind. 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. As a first step to address this issue we have developed ÆMINIUM Stork et al. [2009]. ÆMINIUM is a concurrent-by-default programming language that uses permissions to express data dependencies. 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 *data flow graph* Rumbaugh [1975] which can then be executed by exploiting available, and potentially concurrent, computation resources (cf. Figure 1).

To illustrate this concept, consider the `transfer` function shown below, 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.

```
public 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 a *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 below, 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 an immutable permission to the 'amount' object while both `withdraw` and `deposit` require one each. Because immutable permissions explicitly allow aliasing ÆMINIUM automatically *splits* the one immutable permission into two permission, which are then passed to the two method calls:

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

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

The permission flow of the `transfer` method is shown in Figure 2. 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 data flow 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.

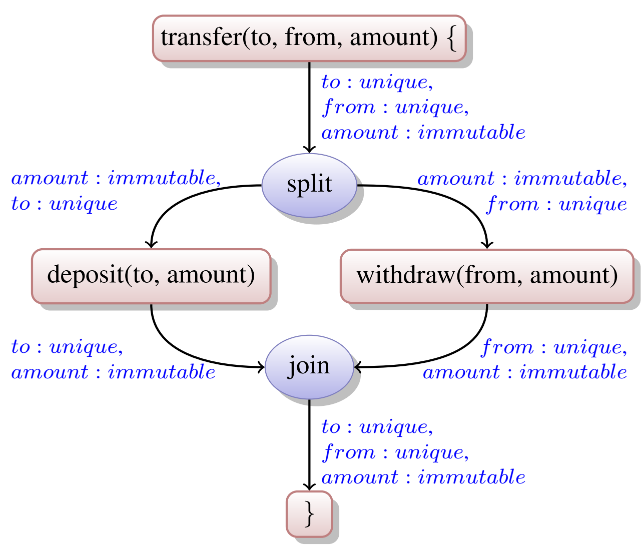


Figure 2: Permission Flow in the Transfer Example. We use the notation $var : perm$ to indicate that we have permission 'perm' for variable 'var'.

In a sense the high-level goals of ÆMINIUM are somewhat similar to the goals of garbage collection. Garbage collection automates memory management, allowing programmers to focus on the functionality of their applications. ÆMINIUM automates the management of concurrency, freeing the programmers from the nasty issues of synchronization and race conditions. Given the design of ÆMINIUM, programmers can start out with an initial version of the program and then incrementally increase its concurrency by refining permission annotations of critical components to allow more concurrent execution.

In this paper we present μÆMINIUM, a core calculus for the ÆMINIUM language. The main contributions of our work are:

- A concurrent-by-default programming language that leverages permissions and data groups to automatically, safely, and deterministically parallelize applications based on permission flows.
- An approach to integrating nondeterministic parallelism into the model above through access permissions to shared data groups, in which developers explicitly specify when nondeterminism is permisible, and which eliminates data races.
- A core calculus called μÆMINIUM which allows formal reasoning about permission flow information and concurrent-by-default execution semantics. The formal system consists of: *i*) a type system that extracts dependency information and avoids race conditions *ii*) a concurrent-by-default evaluation semantics *iii*) a type soundness proof.

2 Overview

In this section we provide an overview of the ÆMINIUM programming language. ÆMINIUM uses *access permissions* Bierhoff and Aldrich [2007] for objects and data group permissions for *data groups* Leino [1998] to compute the permission flow throughout the code (explained in the next sub-sections). The compiler uses this information to compute a data flow graph, which can then be executed in parallel on available computing resources.

2.1 Access Permissions

Access Permissions (AP, Bierhoff and Aldrich [2007]) 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 adapted 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). A *unique* access permission allows read and modifying access to the object.

shared A *shared* access permission to an object reference indicates that there is 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 to the program and automatically splits/joins APs as needed. In ÆMINIUM we define a concurrent execution model based on the non-interference of the permission flow. We define that the permission flows of two code fragments do not interfere with one another if 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, which provides adequate protection. 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 later one.

2.2 Data Groups

Although pure APs are enough to define a concurrent execution model, there are cases where this approach breaks down. In particular there are circumstances in which shared APs are inevitable, for instance in the case of a double-ended, linked list-based queue.

For almost all linked items in the list there exist at least two references in the system (i.e., from the previous and the next elements in the list) which cannot be unique and must therefore be shared. Access to these items must be coordinated, however, as the entire structure must be updated consistenly and so the trival approach of performing a synchronization operation on individual objects is likely to be unsafe.

To overcome these problems we leverage *data groups* (DG, Leino [1998]). A data group represents an abstract collection of objects. Using data groups for grouping multiple objects is a divergence to some previous work that uses 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 all *shared* objects must be part of exactly one data group. We write $shared\langle myGroup\rangle$ to indicate that the shared object is part of the data group *myGroup*.

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, a exclusive GP represents the only currently existing permission through which the data of the data group can be accessed. This allows access to shared data group objects without synchronization.

shared A *shared* GP resembles a shared AP: there can be an arbitrarily 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 the access to a 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.

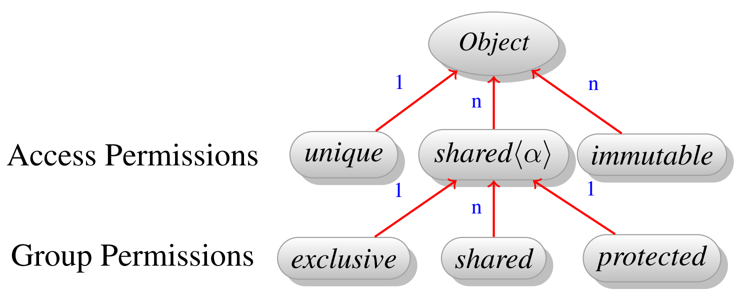


Figure 3: Permissions in ÆMINIUM. Shows different permission kinds and what each permission controls (including arity). Access permissions control access to objects and group permissions control access to data groups of shared objects. There can only exist one unique, exclusive and protected permission 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 to (e.g., $shared\langle\alpha\rangle$ means the object belongs to data group α).

Figure 3 provides an 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*.

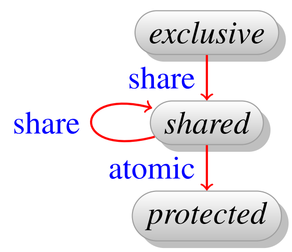
2.2.1 Management of Data Group Permissions

Unlike access permissions, data group permissions are manually split and joined to allow the programmer better control of how accesses to shared data is parallelized. To split an *exclusive* GP into an arbitrarily number of *shared* GPs, ÆMINIUM uses a `split` block (see Figure 2.2.1). The `split` block 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 through its body. The available permissions inside the body are partitioned into disjoint sets. Each one of the those permission sub sets flows to one statement of the body. This means that if multiple statements in the block require the same unique AP or an exclusive GP (which is not mentioned in the `split` block) then the code will not type-check 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.

We already discussed the `atomic` block as a protection mechanism for shared data. In light of data groups, we extend the `atomic` block to refer to the data group for which it provides protection. It will provide a protected GP for the specified data group to its body expression. In particular, the semantics of the `atomic` block is that its body is executed as if it has exclusive access to the shard 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 2.2.1.

// $\overline{gr : gp}$ with
// $\overline{gp} \in \{exclusive, shared\}$
split $\langle \overline{gr} \rangle$ {
    // $\overline{gr : gp}$ with
    // $\overline{gp} : shared$
    atomic $\langle gr_i \rangle$ {
        // $gr_i : protected$
    }
    // $\overline{gr : gp}$ with
    // $\overline{gp} : shared$
}
// $\overline{gr : gp}$ with
// $\overline{gp} \in \{exclusive, shared\}$

(a) Split/Atomic Block



(b) Group Permission Conversion Diagram

Figure 4: Group Permission Splitting/Joining via Split and Atomic blocks. The notation $gr : gp$ means that we have group permission gp for data group gr.

Data groups are declared inside classes in a similar way to fields (see Figure 5, line 6). Data groups are only visible inside classes and their sub classes (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** $\{\ldots\}$' construct. The `unpackInnerGroups` block will trade the permission to the owner group of the receiver object for permissions to inner groups defined in the receiver's class. 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 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.

2.2.2 Discussion

```
class DoubleLinkedListItem⟨owner, data⟩ {
  ... // standard double linked list item
}

class DoubleLinkedList⟨owner, data⟩ {
  group⟨internal⟩ // inner data group

  // ‘head’ belonging to inner data group ‘internal’
  shared⟨internal⟩ DoubleLinkedListItem⟨internal, data⟩ head;

  void
  add⟨exclusive owner, shared data⟩(shared⟨data⟩ Object⟨data⟩ o)
    : shared⟨owner⟩ // shared permission to the receiver
  {
    // owner : exclusive, data : shared
    unpackInnerGroups {
      // internal : exclusive, data : shared
      // access internal data directly
    }
    // owner : exclusive, data : shared
  }

  void
  add⟨shared owner, shared data⟩(shared⟨data⟩ Object⟨data⟩ o)
    : shared⟨owner⟩ // shared permission to the receiver
  {
    // owner : shared, data : shared
    unpackInnerGroups {
      // internal : shared, data : shared
      atomic ⟨internal⟩ {
        // internal : protected, data : shared
        // need protection to access internal data
      }
    }
    // owner : shared, data : shared
  }
  ...
}
```

Figure 5: 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 requires synchronization. The second version only requires a shared permission to the owner and only provides shared permissions to the inner groups, requiring synchronization i.e. `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.

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 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 5. In line 5, the `DoubleLinkedList` class is defined with group parameters `owner` and `data`, using the same syntax as *Java* type parameters. The `owner` parameter represents the data group with which the current object is associated, and `data` 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 class it is impossible for these objects to leave the scope of the class. 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., 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 signature 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. Future work may address this issue.

2.3 Producer/Consumer Example

After the discussion of access permission, data groups and their correlation we now present an example for a producer/consumer in ÆMINIUM (see Figure 6). 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

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

Figure 6: Producer/Consumer Example

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 + 25). After those calls have been completed, and therefore returned their shared permissions to α, the split 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` 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 7 shows the resulting permission flow and the derived data flow graph for this example program.

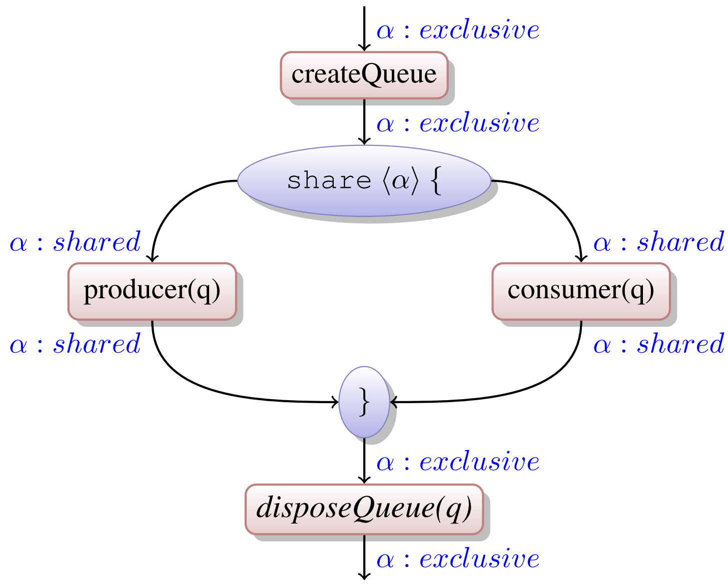


Figure 7: Data Flow Graph for Producer/Consumer Example

3 Grammar

The grammar of μÆMINIUM is shown in Figure 8 and is formulated as an extension to *Featherweight Java* (FJ, Igarashi et al. [2001]). In a nutshell the major extensions are:

i) addition of data group parameters to method calls, class and method declarations. *ii*) addition of group types and extensions of the object types to be parametrized with group parameters *iii*) new language constructs to deal with data groups and allow side effects.

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 • 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 (*F*) 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 objects 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`),

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

Figure 8: μÆMINIUM Language Grammar

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 (`share`), 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 non-empty. 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 and a global data group configuration table ($\mathcal{GT}$) which maps class and method tuples to data group configurations.

4 Static Semantics

4.1 Γ - 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)	Γ	$::=$	$\bullet \mid \Gamma, r : C\langle\overline{gr}\rangle \mid \Gamma, gr : \mathbb{G}$
(Domain)	$dom(\Gamma)$	$::=$	$\{X \mid (X : T) \in \Gamma\}$

4.2 Δ - 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)	Δ	$::=$	$\bullet \mid \Delta, gr : gp$
(Domain)	$dom(\Delta)$	$::=$	$\{gr \mid (gr : gp) \in \Delta\}$

4.3 Σ - Store Typing Context

The store typing context Σ contains typing information for all objects inside the store.

(Store Typing)	Σ	$::=$	$\bullet \mid \Delta, o : T$
(Domain)	$dom(\Sigma)$	$::=$	$\{o \mid (o : T) \in \Gamma\}$

4.4 $\mathcal{G}$ - Data-Group Configuration

The *data group configuration* $\mathcal{G}$ hierarchically tracks the data group requirements of an expression. 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 combined 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.

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

Example: Let us consider a simplified example to provide an intuition on how the data group configuration is used to control the 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 sub-expressions (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 sub-expressions are $requiredPerms(\mathcal{G}_1) = \{gr_0, gr_1\}$ and $requiredPerms(\mathcal{G}_2) = \{gr_0\}$.

$$
\begin{aligned}
&\mathcal{G} \;:=\; (\mathcal{G}_1 \oplus \mathcal{G}_2) \qquad requiredPerms(\mathcal{G}_1) = \{gr_0, gr_1\}\\
&\qquad\qquad\qquad\qquad\quad requiredPerms(\mathcal{G}_2) = \{gr_0\}\\
&e \;:=\; \textbf{let}\ \ \mathtt{x}\ =\ e_1\ \ \textbf{in}\ \ e_2
\end{aligned}
$$

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 $\mathcal{G}'$.

$$
\begin{aligned}
&\{gr_0, gr_1\}|\, \mathcal{G} \vdash\ \textbf{let}\ \ \mathtt{x} =\ e_1\ \ \textbf{in}\ \ e_2\ \mapsto\ \textbf{let}\ \ \mathtt{x} = e_1'\ \textbf{in}\ e_2 \dashv \mathcal{G}'\\
&\mathcal{G}' \;:=\; (\mathcal{G}_1' \oplus \mathcal{G}_2) \qquad requiredPerms(\mathcal{G}_1') = \{gr_1\}\\
&\qquad\qquad\qquad\qquad\quad requiredPerms(\mathcal{G}_2) = \{gr_0\}\\
&e' \;:=\; \textbf{let}\ \ \mathtt{x}\ =\ e_1'\ \ \textbf{in}\ \ e_2
\end{aligned}
$$

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 above, 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 needs again first 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 below.

$$
\begin{aligned}
&\{gr_0, gr_1\}|\, \mathcal{G}' \vdash\ \textbf{let}\ \ \mathtt{x} = e_1'\,\textbf{in}\ \ e_2\ \mapsto\ \textbf{let}\ \ \mathtt{x} = e_1''\ \textbf{in}\ e_2' \dashv \mathcal{G}''\\
&\mathcal{G}'' \;:=\; (\mathcal{G}_1'' \oplus \mathcal{G}_2')\\
&e'' \;:=\; \textbf{let}\ \ \mathtt{x}\ =\ e_1''\ \ \textbf{in}\ \ e_2'
\end{aligned}
$$

4.5 Typing Judgements

Judgement	Description
$T_f\ f$ ok in C	Field f checks in the context of class C.
$T_r\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x})\ \{\ e\ \}$ ok in C	Method m checks in the context of the class C.
$\Gamma\|\Sigma\|\Delta \vdash_C e : T \mid \mathcal{G}$	Given the typing context Γ, the store typing Σ, the permission context Δ, the expression e checks in the context of class C with type T and produces data group configuration $\mathcal{G}$.

4.6 Typing Rules

4.6.1 Sub-Typing

Standard sub-typing rules. Extended to cover data group parameters.

$$\textsc{ST-Class}\quad \frac{\texttt{class}\ C\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends}\ D\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\}}{\Gamma \vdash C\langle\overline{gr_D},\overline{gr_C}\rangle <: D\langle\overline{gr_D}\rangle}$$

$$\textsc{ST-Refl}\quad \frac{}{\Gamma \vdash C\langle\overline{gr_C}\rangle <: C\langle\overline{gr_C}\rangle}$$

$$\textsc{ST-Trans}\quad \frac{\Gamma \vdash C\langle\overline{gr_C}\rangle <: D\langle\overline{gr_D}\rangle \qquad \Gamma \vdash D\langle\overline{gr_D}\rangle <: E\langle\overline{gr_E}\rangle}{\Gamma \vdash C\langle\overline{gr_C}\rangle <: E\langle\overline{gr_E}\rangle}$$

$$\textsc{ST-Bottom}\quad \frac{}{\Gamma \vdash \bot <: C\langle\overline{gr}\rangle}$$

4.6.2 Well-formed types

$$\textsc{WF-Var}\quad \frac{x : T \in \Gamma}{\Gamma \vdash x\ ok}$$

$$\textsc{WF-Group}\quad \frac{\alpha : \mathbb{G} \in \Gamma}{\Gamma \vdash \alpha\ ok}$$

$$\textsc{WF-Group-Name}\quad \frac{x : C\langle\overline{gr}\rangle \in \Gamma \qquad gn \in groupDecls(C)}{\Gamma \vdash x.gn\ ok}$$

$$\textsc{WF-Object}\quad \frac{\Delta \vdash gr\ ok}{\Delta \vdash Object\langle gr\rangle\ ok}$$

$$\textsc{WF-Class}\quad \frac{CT(C) = \ \texttt{class}\ C\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends}\ D\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \qquad \Gamma \vdash \overline{gr_\alpha},\overline{gr_\beta}\ ok}{\Gamma \vdash C\langle\overline{gr_\alpha},\overline{gr_\beta}\rangle\ ok}$$

4.6.3 T-Program

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

T-PROGRAM checks that a program is valid by requiring that all its classes are valid and the main function is valid.

4.6.4 T-Class

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

T-CLASS checks that a class declaration is valid by requiring that its fields and methods are valid.

4.6.5 T-Field

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends}\ D\langle\overline{\alpha}\rangle\ \{\overline{G}\ \overline{FM}\} \\ (this : C\langle\overline{\alpha},\overline{\beta}\rangle, \overline{\alpha : \mathbb{G}}, \overline{\beta : \mathbb{G}}, \overline{G : \mathbb{G}}) \vdash E\langle\overline{gr_E}\rangle\ ok\end{array}}{E\langle\overline{gr_E}\rangle\ f\ \text{ok in C}}\ \textsc{T-Field}$$

T-FIELD checks that the field declaration is valid in class C by requiring that the groups of the field type only mention either group parameters of the class C or internal groups of class C (i.e. `this`.gn).

4.6.6 T-Method

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

T-METHOD checks that the method declaration is valid in class C by requiring that if the method overrides another method of a parent class the declaration is a valid override. The definition of a valid override follow the same definition of *FJ* regarding formal parameter and return types and additionally requires the data group parameters to match exactly. For simplicity reasons μÆMINIUM does not support polymorphism for data group parameters as shown in Section 2.

4.6.7 T-UnpackGroupsOf

$$\frac{\begin{array}{c}\Gamma|\Sigma \vdash r : C\langle\overline{gr}\rangle \qquad \Delta = \Delta', (gr_0 : exclusive) \\ groupDecls(C) = \overline{gn} \qquad \Gamma, (\overline{r.gn : \mathbb{G}})|\Sigma|\Delta', (\overline{r.gn : exclusive}) \vdash e : T \mid \mathcal{G}\end{array}}{\Gamma|\Sigma|\Delta \vdash_C \texttt{unpackGroupsOf}\ r\ \texttt{in}\ e : T \mid (\{gr_0, \overline{r.gn}\} \oplus \mathcal{G})}\ \textsc{T-UnpackGroupsOf-Exclusive}$$

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

T-UNPACKGROUPSOF-EXCLUSIVE applies in the case when we have an *exclusive* permission to the data group gr_0 by which the object r is owned. In that case we replace the *exclusive* permission of the owner group with *exclusive* permissions to the inner groups of r and type check the sub-expression. Because we first have to unpack the object r, we sequentially combine ($\oplus$) the data group configuration of the sub-expression $\mathcal{G}$ with the set of the owner data group and the inner permissions of r.

T-UNPACKGROUPSOF-SHARED follows the same principle as the T-UNPACKGROUPSOF-EXCLUSIVE rule, with the difference that it only applies if we have either a *shared* or *protected* permission of the owner data group. In this case we have to substitute the owner permissions with a *shared* permission for the inner data groups. Replacing it with *exclusive* permission would violate soundness, as multiple expression might simultaneously unpack the inner groups of the same object.

4.6.8 T-Split

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

T-SPLIT checks if the two sub-expressions type check under the assumption that the permissions to specified groups $\overline{gr}$ have been replaced/split with shared permissions. This rule only applies if the available permissions to the specified data groups are either *exclusive* or *shared*. Because the two sub-expression are only sharing *shared* permissions we construct a data group configuration by parallel composition (‖) of the data group configurations of the two sub-expressions.

4.6.9 T-Atomic

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

T-ATOMIC checks if its sub-expression type checks under the assumption that the permission of the specified data group gr is converted/splitted to a *protected* permission. Because the sub-expression can only execute once the atomic block guarantees the protection of the execution we sequentially compose the set of the specified data group with data group configuration of the sub-expression.

4.6.10 T-InAtomic

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

T-INATOMIC Identical to T-ATOMIC.

4.6.11 T-Let

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

T-LET Checks if the first sub-expression e_1 type checks with a non-strict sub-set of the available permissions and if the second expression type checks with all available permissions and the fact that the variable x is bound to the value of e_1. Because of the lexical order of the two expressions we combine their data configurations sequentially ($\oplus$).

4.6.12 T-Reference

$$\textsc{T-Reference}\quad \frac{\Gamma|\Sigma \vdash r : D\langle\overline{gr}\rangle}{\Gamma|\Sigma|\Delta \vdash_C r : D\langle\overline{gr}\rangle \mid \bullet}$$

T-REFERENCE checks if the reference is well typed. Because no data access is occurring the data group configuration is empty ($\bullet$).

4.6.13 T-Field-Read

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

T-FIELD-READ checks if the receiver reference is well typed, the field is valid and we either have an *exclusive* or *protected* permission to the data group gr_0 to which the receiver belongs to. Because of the data access the resulting group configuration consists of the singleton set formed by gr_0.

4.6.14 T-Field-Assign

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

T-FIELD-ASSIGN checks if the receiver reference is well typed, the field is valid , the assigned reference r_v is assignment compatible and we either have an *exclusive* or *protected* permission to the data group gr_0 to which the receiver belongs to. Because of the data access the resulting group configuration consists of the singleton set formed by gr_0.

4.6.15 T-New

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

T-NEW checks if the provided data groups are well typed. Because no data is accessed the resulting data group configuration is empty ($\bullet$).

4.6.16 T-Call

T-CALL

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

T-CALL checks that the receiver, the provided data groups and parameter values are well typed and are compatible. It also checks that the by the method declaration required permissions are available. The resulting data group configuration is formed by the set of data groups that are passed into the function call.

5 Dynamic Semantics

5.1 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. 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 (cf. Section 5.3)

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

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.

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

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

5.2 Runtime Permission Context

The *runtime permission context* δ is used to model permission flows at runtime. Similar to static permissions, the runtime permission can be split and flow along different paths. But, unlike static permissions, runtime permissions do not carry any additional information about their specific type. The runtime permissions are used to model the permission flow at runtime and therefore specify which expressions currently have which permissions.

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 of inner groups.

(runtime permission context)	δ	$::=$	$\bullet \mid \delta, o.gn$
(domain)	$dom(\delta)$	$::=$	$\{o.gn \mid o.gn \in \delta\}$

5.3 Group Token Context

The *group token context* Ψ is a set of group access tokens, i.e., 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 locking based approach for its simplicity of implementation and semantics. In 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 non-deterministic way. This models non-determinism of how atomic blocks can lock data groups. Locked group access 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)	Ψ	$::=$	$\bullet \mid \Psi, o.gn@S$
(domain)	$dom(\Psi)$	$::=$	$\{o.gn \mid o.gn@S \in \Psi\}$

5.4 Judgements

Judgement	Judgement Form	Description
Evaluation	$\mu\|\delta\|\Psi\|\mathcal{G} \vdash e \mapsto e' \dashv \mu_\delta\|\Psi'\|\mathcal{G}'$	Given the store μ, the runtime permissions δ, the group token context Ψ, the data group configuration $\mathcal{G}$, the expression e steps to e' by producing the differential heap μ_δ, the group token context Ψ' and data group configuration $\mathcal{G}$.

5.4.1 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 Section 5.5).

5.5 Transitive Evaluation Rule

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

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

5.6 Evaluation Rules

5.6.1 E-UnpackGroupsOf

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

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

$$\textsc{E-UnpackGroupsOf-Value}\quad \frac{}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ v \mapsto v \dashv \bullet|\Psi|\bullet}$$

E-UNPACKGROUPSOF-REPLACE If we have the runtime permission ($v'.gn'$) to the data group by which the reference (v_r) is owned, then we execute the sub-expression by replacing the permission to the owner group with permissions to the inner groups ($\overline{v_r.gn}$).

E-UNPACKGROUPSOF-NONE If we do *not* have the runtime permission ($v'.gn'$) to the data group by which the reference (v_r) is owned, then we execute the sub-expression right away, carrying forward operations that do not depend on the inner groups of (v_r).

E-UNPACKGROUPSOF-VALUE reduces to the value generated by its sub-expression.

5.6.2 E-Let

E-LET-1

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

E-LET-2

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

E-LET-12

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

E-LET-VALUE

$$\frac{\mathcal{G} = (\bullet \oplus \mathcal{G}_2) \qquad \mathcal{G}' = [^v/_x]\mathcal{G}_2}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{let } x = v \texttt{ in } e_2 \mapsto [^v/_x]e_2 \dashv \bullet|\Psi|\mathcal{G}'}$$

E-LET-1 Executes the first sub-expression (e_1) by supplying it with a sub set (δ_1) of runtime permissions that are required by the first expression, a non-deterministic sub-set of available group access tokens (Ψ_1) and its data configuration ($\mathcal{G}_1$).

E-LET-2 Executes the second sub-expression (e_2) by supplying it with a sub-set (δ_2) of runtime permissions that are *not* required by the first expression, a random sub-set of available group access tokens (Ψ_1) and its data configuration ($\mathcal{G}_2$).

E-LET-12 Simultaneously executes both sub expressions. The first sub-expression (e_1) executes by supplying it with a sub-set (δ_1) of runtime permissions that are required by the first expression, a random sub-set of available group access tokens (Ψ_1) and its data configuration ($\mathcal{G}_1$), while the second sub-expression gets the remaining runtime permissions (δ_2) and data group tokens (Ψ_2) along with its data group configuration ($\mathcal{G}_2$).

E-LET-VALUE follows the standard let-normal-form semantics by substituting the value of e_1 for x in e_2 and $\mathcal{G}_2$.

5.6.3 E-Split

E-SPLIT-1

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

E-SPLIT-2

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

E-SPLIT-12

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

E-SPLIT-VALUE

$$\frac{\mathcal{G} = (\bullet \parallel \bullet)}{\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{ share } \langle\overline{v.gn}\rangle \texttt{ between } v_1 \parallel v_2 \mapsto \texttt{null} \dashv \bullet|\Psi|\bullet}$$

E-SPLIT-1 Executes the first sub-expression (e_1) by supplying it with a sub set (δ_1) of runtime permissions that is required by the first expression, a random sub-set of available group access tokens (Ψ_1) and its data configuration ($\mathcal{G}_1$).

E-SPLIT-2 The symmetric rule to E-SPLIT-1 which lets e_2 take a step instead of e_1.

E-SPLIT-12 Executes both of its sub-expressions (e_1, e_2) but giving the sub-set of runtime permissions they require (δ_1, δ_2), disjoint sub-sets of group access tokens (Ψ_1, Ψ_2) and their corresponding data group configurations ($\mathcal{G}_1, \mathcal{G}_2$). Upon completion the sub-differential stores ($\mu_{\delta_1}, \mu_{\delta_2}$) of both sub-evaluations will be merged accordingly to Section 5.1.

E-SPLIT-VALUE Reduces `null` in the case both of its sub expressions have been evaluated to values. The rule throws away the resulting values, because both sub-expressions have been primarily evaluated for their side effects.

5.6.4 E-Atomic

E-ATOMIC-STEP1

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

E-ATOMIC-STEP2

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

E-ATOMIC-INATOMIC

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

E-INATOMIC-STEP

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

E-INATOMIC-VALUE

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

E-ATOMIC-STEP1 If we do *not* have the runtime permission ($v.gn$) for the data group mentioned in the `atomic` block we execute its sub-expression (e). The general absence of the require runtime permission does not allow the execution of any code that access data of this data group.

E-ATOMIC-STEP2 If we have the runtime permission ($v.gn$) for the data group mentioned in the `atomic` block but *not* the corresponding group access token in an *unlocked* state, we execute its sub-expression (e) without this runtime permission to prevent the execution of operations that access data of this data group.

E-ATOMIC-INATOMIC If we have the runtime permission ($v.gn$) for the data group mentioned in the `atomic` block and the corresponding group access token in an *unlocked* state, then the whole expression step to `inatomic` and change the access token state to *locked* (entering the `atomic` block).

E-INATOMIC-STEP We keep evaluating the sub expression.

E-INATOMIC-VALUE Once the sub-expression has been reduced to a value, the whole expression reduces to that value and the group access token is switched back to *unlocked* (leaving `atomic` block).

5.6.5 E-Field-Read

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

E-FIELD-READ If we have the runtime permission to the data group by which the receiver is owned we return the value of the specified field and add the receiver object to the differential store.

5.6.6 E-Field-Assign

$$\textsc{E-Field-Assign}\;\frac{\begin{array}{c}\mathcal{G} = \{v_g.gn\} \qquad v_g.gn \in \delta \qquad \mu \vdash \langle v_r \mapsto obj_r\rangle \qquad obj_r = C[\overline{f_r = v_{f_r}}, f_{ri} = v_{fi}, \overline{f_r = v_{f_r}}] \\ obj_r' = C[\overline{f_r = v_{f_r}}, f_{ri} = o_v, \overline{f_r = v_{f_r}}] \qquad \mu_\delta = \langle v_r \mapsto obj_r'\rangle\end{array}}{\mu|\delta|\Psi|\mathcal{G} \vdash v_r.f_{ri} := o_v \mapsto o_v \dashv \mu_\delta|\Psi|\mathcal{G}}$$

E-FIELD-ASSIGN If we have the runtime permission to the data group by which the receiver is owned we return the value of the specified field and add the receiver object to the differential store.

5.6.7 E-New

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

E-NEW We allocate a new object and initialize all its fields to `null` and return that new mapping in the differential store. Additionally we add new group access tokens for all the inner groups of the newly created object in the *unlocked* state to the existing group access tokens. The initialization does not count as data access and therefore does not require the availability of the corresponding owner runtime permission. In the real implementation we obviously allocate the object in a lazy manner (i.e. only when they are needed) to avoid resource exhaustion.

5.6.8 E-Call

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

E-CALL If we have all the required runtime permissions to the data groups we lookup the body expression with its data group configuration. Then we substitute the concrete values for the formals in the expression and the data group configuration and step to the new expression and data group configuration.

Appendices

A Utility Rules

A.1 Helper Functions

$fields(C) = \overline{F}$ **returns fields of class C**

H-FIELDS-OBJ

$$\frac{}{fields(Object) = \bullet}$$

H-FIELDS

$$\frac{CT(C) = \texttt{ class } C\langle\overline{\alpha}, \overline{\beta}\rangle \texttt{ extends } D\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \qquad fields(D) = \overline{F'}}{fields(C) = \overline{F'}, \overline{F}}$$

A.1.1 groupDecls

$groupDecls(C) = \overline{gn}$ **returns the declared groups of class C**

H-GROUPDECLS-OBJ

$$\frac{}{groupDecls(Object) = \bullet}$$

H-GROUPDECLS

$$\frac{CT(C) = \texttt{ class } C\langle\overline{\alpha}, \overline{\beta}\rangle \texttt{ extends } D\langle\overline{\alpha}\rangle\{\overline{\texttt{group}\langle gn\rangle}\ \overline{F}\ \overline{M}\} \qquad groupDecls(D) = \overline{gn'}}{groupDecls(C) = \overline{gn'}, \overline{gn}}$$

A.1.2 override

$override(C, m)\ ok$ **checks if a method correctly overrides another method**

H-OVERRIDE

$$\frac{\begin{array}{c} CT(C) = \texttt{ class } C\langle\overline{\alpha}, \overline{\beta}\rangle \texttt{ extends } D\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \qquad T_{Cr}\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_{Cx}\ x})\ \{\ e_C\ \} \in M \\ mdecl(D, m) = T_{Dr}\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_{Dx}\ x})\ \{\ e_D\ \} \qquad \overline{T_{Cx}} <: \overline{T_{Dx}} \qquad T_{Dr} <: T_{Cr} \end{array}}{override(C, m)\ ok}$$

H-OVERRIDE-TOP

$$\frac{CT(C) = \texttt{ class } C\langle\overline{\alpha}, \overline{\beta}\rangle \texttt{ extends } D\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \qquad \neg mdecl(D, m)}{override(C, m)\ ok}$$

A.1.3 requiredPerms

$requiredPerms(\mathcal{G}) = \overline{gr}$ **returns all permissions in** $\mathcal{G}$

H-RequiredTokens-Leaf

$$\frac{\overline{o.gn} \in \{gr\}}{requiredPerms(\{\overline{gr}\}) = \{\overline{o.gn}\}}$$

H-RequiredTokens-Par

$$\frac{}{requiredPerms((\mathcal{G}_1 \parallel \mathcal{G}_2)) = requiredPerms(\mathcal{G}_1) \cup requiredPerms(\mathcal{G}_2)}$$

H-RequiredTokens-Par

$$\frac{}{requiredPerms((\mathcal{G}_1 \oplus \mathcal{G}_2)) = requiredPerms(\mathcal{G}_1) \cup requiredPerms(\mathcal{G}_2)}$$

A.1.4 requiredTokens

$requiredTokens(e) = \{\overline{o.gn@L}\}$ **the access tokens corresponding to inatomic expression in** e

H-RequiredTokens-UnpackGroupsOf

$$\frac{}{requiredTokens(\ \texttt{unpackGroupsOf}\ r\ \texttt{in}\ e\ ) = requiredTokens(e)}$$

H-RequiredTokens-Let

$$\frac{}{requiredTokens(\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ ) = requiredTokens(e_1) \cup requiredTokens(e_2)}$$

H-RequiredTokens-Atomic

$$\frac{}{requiredTokens(\ \texttt{atomic}\ \langle gr \rangle\ e\ ) = requiredTokens(e)}$$

H-RequiredTokens-InAtomic

$$\frac{}{requiredTokens(\ \texttt{inatomic}\ \langle gr \rangle\ e\ ) = \{gr@L\} \cup requiredTokens(e)}$$

H-RequiredTokens-Share

$$\frac{}{requiredTokens(\ \texttt{share}\ \langle \overline{gr} \rangle\ \texttt{between}\ e_1 \parallel e_2\ ) = requiredTokens(e_1) \cup requiredTokens(e_2)}$$

H-RequiredTokens-Atoms

$$\frac{}{requiredTokens(\ a\ ) = \bullet}$$

A.1.5 mdecl

$mdecl(C, m) = M$ **looks up the method declaration of m in class C**

H-MDECL
$$\frac{CT(C) = \texttt{class } C\langle\overline{\alpha}, \overline{\beta}\rangle \texttt{ extends } D\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \quad T_r\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x}) : T_{this}\ \{\ e\ \} \in \overline{M}}{mdecl(C, m) = T_r\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x}) : T_{this}\{\ e\ \}}$$

H-MDECL-REC
$$\frac{CT(C) = \texttt{class } C\langle\overline{\alpha}, \overline{\beta}\rangle \texttt{ extends } D\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\} \qquad T\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x}) : T_{this}\ \{\ e\ \} \notin \overline{M}}{mdecl(C, m) = mdecl(D, m)}$$

A.1.6 mbody

$mbody(C, m) = \overline{\gamma}.\overline{x}.e \times \mathcal{G}$ **looks up the method body expression m in class C**

H-MBODY
$$\frac{mdecl(C, m) = T_r\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x}) : T_{this}\ \{\ e\ \} \qquad \mathcal{GT}(C, m) = \mathcal{G}}{mbody(C, m) = \overline{\gamma}.\overline{x}.e \times \mathcal{G}}$$

B Definitions & Proofs

B.1 Definitions

DEFINITION 1 (STUCK)
An program state $(\mu|\delta|\Psi|\mathcal{G}|e)$ is stuck if e is not a value and:

- *$(\mu|\delta|\Psi|\mathcal{G}|e)$ does not take a step (i.e. $(\mu|\delta|\Psi|\mathcal{G}|e) \mapsto (\mu'|\delta'|\Psi'|\mathcal{G}'|e')$ for some $e', \mu', \delta', \Psi', \mathcal{G}'$)*
- *$(\mu|\delta|\Psi|\mathcal{G}|e)$ does not wait for resources to become available*

DEFINITION 2 (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).

DEFINITION 3 (UNIQUE ALLOCATION)
If multiple expression simultaniously allocate new objects, then every creation site will get a unique object reference.

DEFINITION 4 (WELL-FORMED PROGRAM STATE)
A program state is well typed, written as $\cdot|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$*, if :*

- $\cdot|\Sigma|\Delta \vdash e : T \;|\mathcal{G}$
- μ *is well typed with respect to* Σ
- *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* $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

Lemma 1 (Weakening)
If $\Gamma' \subseteq \Gamma$, $\Sigma' \subseteq \Sigma$ and $\Delta' \subseteq \Delta$ then $\Gamma'|\Sigma'|\Delta' \vdash e : T \;|\mathcal{G}$ implies $\Gamma|\Sigma|\Delta \vdash e : T \;|\mathcal{G}$.

Lemma 2 (Store Typing)
A store μ is said to be *well typed*, written $\Gamma|\Sigma \vdash \mu$, if:

- $dom(\Sigma) = dom(\mu)$
- $\forall o \in dom(\mu) : \Gamma|\Sigma \vdash \mu(o) : \Sigma(o)$

Lemma 3 (Store Monotonicity)
If $\Gamma|\Sigma \vdash \mu$ and $\Sigma \subseteq \Sigma'$ then $\Gamma|\Sigma' \vdash \mu$.

Lemma 4 (Substitution)
If $\Gamma, x : T_x, \Gamma'|\Sigma|\Delta \vdash e : T_e \;|\mathcal{G}_e$ and $\Gamma|\Sigma|\Delta \vdash r : T_r \;|\bullet$ and $T_r <: T_x$ then $\Gamma, [^r/_x]\Gamma'|\Sigma|[^r/_x]\Delta \vdash [^r/_x](e : T_e \;|\mathcal{G}_e)$.

Lemma 5 (Progress)
If $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$ (i.e. a well-formed program state) then either:

- e is value and $\mathcal{G} = \bullet$

- $\mu|\delta|\Psi|\mathcal{G} \vdash e \mapsto e' \dashv \mu_\delta|\Psi'|\mathcal{G}'$ for some $e', \mu_\delta, \Psi', \mathcal{G}'$
- e stops execution with <u>null-dereference</u>
- e is waiting for resource to become available

Lemma 6 (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 $T' <: T$
- $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
  $(\Sigma(o) = C\langle o'.gn' \ldots\rangle$
  $\wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta$
  $\wedge (gp = shared \implies \exists \; \texttt{inatomic} \; \langle o'.gn'\rangle \ldots \in e))$

Lemma 7 (Type-Safety)
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.

Lemma 8 (Canonical-Forms)

- If v is value and has type T, then v is either `null` or o.
- If v is value and has type $\mathbb{G}$, then v is either $\texttt{null}.gn$ or $o.gn$.

Lemma 9 (Inversion)

- If $\Gamma|\Sigma|\Delta \vdash \; \texttt{let} \; x = e_1 \; \texttt{in} \; e_2 : T_2|\mathcal{G}$ then $\Delta = \Delta_1, \Delta_R$ and $\mathcal{G} = (\mathcal{G}_1 \oplus \mathcal{G}_2)$ and $\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1 \; |\mathcal{G}_1$ form some T_1 and $\Gamma, x{:}T_1|\Sigma|\Delta_1, \Delta_R \vdash e_2 : T_2 \; |\mathcal{G}_2$
- If $\Gamma|\Sigma|\Delta \vdash r.f_i : T_i|\mathcal{G}$ then $\Gamma|\Sigma|\Delta \vdash r : D\langle\overline{gr}\rangle$ for some D and $\overline{gr}$ and $\mathcal{G} = \{gr_0\}$ and $T_i \; f_i \in fields(D)$ and $\Delta \vdash gr_o : gp$ with $gp \in \{exclusive, protected\}$.
- If $\Gamma|\Sigma|\Delta \vdash r_r.f_i := r_v : T_v|\mathcal{G}$ then $\Gamma|\Sigma|\Delta \vdash r_r : D\langle\overline{gr}\rangle$ for some D and $\overline{gr}$ and $\mathcal{G} = \{gr_0\}$ and $T_i \; f_i \in fields(D)$ and $\Gamma|\Sigma|\Delta \vdash r_r : T_v \; |\bullet$ with $T_v <: T_i$ and $\Delta \vdash gr_o : gp$ with $gp \in \{exclusive, protected\}$.
- If $\Gamma|\Sigma|\Delta \vdash \; \texttt{new} \; C\langle\overline{o_g.gn_g}\rangle : T \; |\mathcal{G}$ then $CT(D) = \; \texttt{class} \; D\langle\overline{\alpha}, \overline{\beta}\rangle \; \texttt{extends} \; E\langle\overline{\alpha}\rangle\{\overline{G}\,\overline{F}\,\overline{M}\}$ and $\Gamma|\Sigma \vdash \overline{gr : \mathbb{G}}$ and $T = [^{o_g.gn_g}/_{\overline{\alpha},\overline{\beta}}]C\langle\overline{\alpha}, \overline{b}\rangle$ and $\mathcal{G} = \bullet$.

- If $\Gamma|\Sigma|\Delta \vdash r.m\langle\overline{gr}\rangle(\overline{r_p}) : T\ |\mathcal{G}$ then $\Gamma|\Sigma \vdash r : T_r, \overline{p : T_p}, \overline{gr : \mathbb{G}}$ and $\Delta \vdash \overline{gr : gp}$ and $T_r = D\langle\overline{gr_D}\rangle$ and $CT(D) =$ `class` $D\langle\overline{\alpha}, \overline{\beta}\rangle$ `extends` $E\langle\overline{\alpha}\rangle\{\overline{G}\ \overline{F}\ \overline{M}\}$ and $mdecl(D, m) = T_{result}\ m\langle\overline{gp\ \gamma}\rangle(\overline{T_x\ x})\{\ e\ \}$ and $\overline{T_p} <: [\overline{gr}, \overline{gr_D}/\overline{\gamma}, \overline{\alpha}, \overline{\beta}]T_x$ and $T_r <: [\overline{gr}, \overline{gr_D}/\overline{\gamma}, \overline{\alpha}, \overline{\beta}]D\langle\overline{\alpha}, \overline{\beta}\rangle$ and $T = [\overline{gr}, \overline{gr_D}/\overline{\gamma}, \overline{\alpha}, \overline{\beta}]T_{result}$ and $\mathcal{G} = \{\overline{gr}\}$.
- If $\Gamma|\Sigma|\Delta \vdash$ `unpackGroupsOf` r `in` e then $\Gamma|\Sigma \vdash r : C\langle\overline{gr}\rangle$ and $\Delta = \Delta', (gr_0 : qp)$ and $groupDecls(C) = \overline{gn}$ and $\Gamma, (\overline{r.gn : \mathbb{G}})|\Sigma|\Delta', (\overline{r.gn : qp'}) \vdash e : T \mid \mathcal{G}_e$ and $G = (\{gr_o, \overline{r.gn}\} \oplus \mathcal{G}_e)$.
- If $\Gamma|\Sigma|\Delta \vdash_C$ `atomic` $\langle gr\rangle\ e : T \mid \mathcal{G}$ then $\mathcal{G} = (\{gr\} \oplus \mathcal{G}_e)$ and $\Delta = \Delta', (gr : shared)$ and $\Gamma|\Sigma \vdash gr : \mathbb{G}$ and $\Gamma|\Sigma|(\Delta', gr : protected) \vdash_C e : T \mid \mathcal{G}$
- If $\Gamma|\Sigma|\Delta \vdash_C$ `inatomic` $\langle gr\rangle\ e : T \mid \mathcal{G}$ then $\mathcal{G} = (\{gr\} \oplus \mathcal{G}_e)$ and $\Delta = \Delta', (gr : shared)$ and $\Gamma|\Sigma \vdash gr : \mathbb{G}$ and $\Gamma|\Sigma|(\Delta', gr : protected) \vdash_C e : T \mid \mathcal{G}$
- If $\Gamma|\Sigma|\Delta \vdash_C$ `share` $\langle\overline{gr}\rangle$ `between` $e_1 \parallel e_2 : \perp \mid \mathcal{G}$ then $\{\overline{gp}\} \subseteq \{exclusive, shared\}$ and $\Delta = \Delta_1, \Delta_2, \Delta_r, (\overline{gr : gp})$ and $\Gamma|\Sigma|(\Delta_1, \overline{gr : shared}) \vdash_C e_1 : T_1\ |\mathcal{G}_1$ and $\Gamma|\Sigma|(\Delta_2, \overline{gr : shared}) \vdash_C e_2 : T_2\ |\mathcal{G}_2$ and $\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2)$.

B.2 Proofs

B.2.1 Proof Preservation

Proof (Preservation) by induction on $\mu|\delta|\Psi|\mathcal{G} \vdash e \mapsto e' \dashv (\mu_{\delta_\mathcal{R}}, \mu_{\delta_\mathcal{W}})|\Psi'|\mathcal{G}'$

Case E-FIELD-READ :
$\therefore e = o_r.f_i$
$\therefore e' = o_v$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|o_v)$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash o_v : T\ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq$ `race`
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists$ `inatomic` $\langle o.gn\rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \ldots \in e$
(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists$ `inatomic` $\langle o'.gn'\rangle \ldots \in e))$
by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|o_r.f_i)$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash o_r.f : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$T_i\ f_i \in fields(D)$

$o_r : D\langle\overline{gr}\rangle$

$\mathcal{G} = \{gr_0\}$

$gr_o : gp \in \Delta$ with $gp \in \{exclusive, protected\}$

WLOG: let $o'.gn' = gr_0$

by E-FIELD-READ:

$o'.gn' \in \delta$

$\Psi' = \Psi$

$\mathcal{G}' = \bullet$

$\mu_\delta = \langle o_r \mapsto D[\overline{f = v_f}]\rangle$

WLOG: let $\Sigma' = \Sigma$

by STORE-TYPING: $o_v : T_v \in \Sigma$ with $T_v <: T_i$

by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash o_v : T_v\ |\bullet$ (TS1.1)

by CONSTRUCTION: $\mu_\delta \neq \texttt{race} \implies \mu' = [\mu_\delta]\mu \neq \texttt{race}$ (TS1.2)

by CONSTRUCTION: $\mu' = [\mu_\delta]\mu = \mu$ (TS1.3)

by E-FIELD-READ: δ, Δ do not change (TS1.4)

by E-FIELD-READ,AS5,AS6: $\Psi = \Psi'$ (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|o_v)$ (TS1)

by E-FIELD-READ: $dom(\mu) \cap dom(\mu_\delta) = \{o_r\}$

by INVERSION, E-FIELD-READ:

$\Sigma(o_r) = D\langle o'.gn' \ldots\rangle$

$o'.gn' \in \delta$

$o'.gn' : gp \in \Delta \wedge gp \in \{exclusive, protected\}$

$\therefore o \in dom(\mu) \cap dom(\mu_\delta) \implies$

$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \Longrightarrow \exists\, \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$ (TS2)

Case E-Let-12 :

$\therefore e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$
$\therefore e' = \texttt{let}\ x = e_1'\ \texttt{in}\ e_2'$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2')$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : T\ |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$

$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\ \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2)$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$\Delta = \Delta_1, \Delta_R$

$\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2$

$\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1\ |\mathcal{G}_1$ for some T_1

$\Gamma, x{:}T_1|\Sigma|\Delta_1, \Delta_R \vdash e_2 : T\ |\mathcal{G}_2$

by E-LET-12:

let $\delta_1 = \delta \cap requiredPerms(\mathcal{G}_1)$ be the (sub-)set of permissions that are required by e_1

$\therefore o.gn \in \delta_1 \implies o.gn : gp \in \Delta_1$

$\mu \neq \texttt{race}$ and is well typed with respect to Σ

let $\Psi = \Psi_1, \Psi_2$ with $requiredTokens(e_1) \subseteq \Psi_1$ and $requiredTokens(e_2) \subseteq \Psi_2$

$\therefore o.gn@L \in \Psi_1 \implies \exists$ exactly on $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_1$

$\therefore (o.gn@U \in \Psi_1 \vee o.gn@\_ \notin \Psi_1) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_1$

$\therefore o.gn@L \in \Psi_2 \implies \exists$ exactly on $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_2$

$\therefore (o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e_2$

$\Gamma|\Sigma|\Delta \vdash e_1 : T_1\ |\mathcal{G}_1$ is well typed

$\therefore\ \Gamma|\Sigma|\Delta_1 \vdash_{wf} (\mu|\delta_1|\Psi_1|\mathcal{G}_1|e_1)$

by IH: on $\Gamma|\Sigma_1|\Delta_1 \vdash_{wf} (\mu_1|\delta_1|\Psi_1|\mathcal{G}_1|e_1)$ with $\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1\ |\mathcal{G}_1$ where $\mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto$

$e_1' \dashv \mu_{\delta_1}|\Psi_1'|\mathcal{G}_1'$

for some $\mathcal{G}_1', \Sigma \subseteq \Sigma_1$

(IS1.1-1) $\Gamma|\Sigma_1|\Delta_1 \vdash e_1' : T_1 \; |\mathcal{G}_1'$

(IS1.2-1) $\mu_1 \neq \texttt{race}$

(IS1.3-1) μ_1 is well typed with respect to Σ'

(IS1.4-1) $o.gn \in \delta_1 \implies o.gn : gp \in \Delta_1$

(IS1.5-1) $(o.gn@U \in \Psi_1 \vee o.gn@\_ \notin \Psi_1) \implies \nexists\, \texttt{inatomic}\, \langle o.gn\rangle \ldots \in e_1'$

(IS1.6-1) $o.gn@L \in \Psi_1 \implies \exists$ exactly one $\texttt{inatomic}\, \langle o.gn\rangle \ldots \in e_1'$

(IS2-1) $o \in dom(\mu) \cap dom(\mu_{\delta_1}) \implies$

$(\Sigma_1(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta_1 \wedge o'.gn' : gp \in \Delta_1 \wedge (gp = shared \implies \exists\, \texttt{inatomic}\, \langle o'.gn'\rangle \ldots \in e_1))$

by E-LET-12:

let $\delta_2 = (\delta - \delta_1)$ be the sub-set of permissions that did not go to e_1

$\therefore o.gn \in \delta_2 \implies o.gn : gp \in \Delta_2$

$\mu \neq \texttt{race}$ and is well typed with respect to Σ

let $\Psi = \Psi_1, \Psi_2$ with $requiredTokens(e_1) \subseteq \Psi_1$ and $requiredTokens(e_2) \subseteq \Psi_2$

$\therefore o.gn@L \in \Psi_1 \implies \exists$ exactly on $\texttt{inatomic}\, \langle o.gn\rangle \ldots \in e_1$

$\therefore (o.gn@U \in \Psi_1 \vee o.gn@\_ \notin \Psi_1) \implies \nexists\, \texttt{inatomic}\, \langle o.gn\rangle \ldots \in e_1$

$\therefore o.gn@L \in \Psi_2 \implies \exists$ exactly on $\texttt{inatomic}\, \langle o.gn\rangle \ldots \in e_2$

$\therefore (o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\, \texttt{inatomic}\, \langle o.gn\rangle \ldots \in e_2$

$\Gamma|\Sigma|\Delta \vdash e_2 : T_2 \; |\mathcal{G}_2$ is well typed

$\therefore \; \Gamma|\Sigma|\Delta_2 \vdash_{wf} (\mu|\delta_2|\Psi_2|\mathcal{G}_2|e_2)$

by IH: on $\Gamma|\Sigma_2|\Delta_2 \vdash_{wf} (\mu_2|\delta_2|\Psi_2|\mathcal{G}_2|e_2)$ with $\Gamma|\Sigma|\Delta_1 \vdash e_2 : T_1 \; |\mathcal{G}_1$ where $\mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e_2' \dashv \mu_{\delta_2}|\Psi_2'|\mathcal{G}_2'$

for some $\mathcal{G}_2', \Sigma \subseteq \Sigma_1$

(IS1.1-2) $\Gamma|\Sigma_2|\Delta_2 \vdash e_2' : T \; |\mathcal{G}_2'$

(IS1.2-2) $\mu_2 \neq \texttt{race}$

(IS1.3-2) μ_2 is well typed with respect to Σ'

(IS1.4-2) $o.gn \in \delta_2 \implies o.gn : gp \in \Delta_2$

(IS1.5-2) $(o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\, \texttt{inatomic}\, \langle o.gn\rangle \in e_2'$

(IS1.6-2) $o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\, \langle o.gn\rangle \in e_2'$

(IS2-1) $o \in dom(\mu) \cap dom(\mu_{\delta_1}) \implies$

$(\Sigma_2(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta_1 \wedge o'.gn' : gp \in \Delta_1 \wedge (gp = shared \implies \exists\, \texttt{inatomic}\, \langle o'.gn'\rangle \ldots \in e_1))$

by UNIQUE ALLOCATE:

$\Sigma_1 = \Sigma \cup \Sigma_1'$ and $\Sigma_1 = \Sigma \cup \Sigma_2'$ with $dom(\Sigma_1') \cap dom(\Sigma_2') = \bullet$

$\therefore$ let $\Sigma' = \Sigma \cup \Sigma_1 \cup \Sigma_2$

$dom(\Psi_1') \cap dom(\Psi_2') = \bullet$

by E-LET-12: $\Psi' = \Psi'_1, \Psi'_2$ with $dom(\Psi'_1) \cap dom(\Psi_2)'$

by IS1.5-1, IS1.5-2:
$(o.gn@U \in \Psi' \vee o.gn@\_ \notin \Psi') \implies \nexists\, \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e'$ (TS1.5)

by IS1.6-1, IS1.6-2:
$o.gn@L \in \Psi' \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e'$ (TS1.6)

by E-LET-12: $\delta = \delta_1, \delta_2$
$\therefore\ \nexists o_1, o_2 \ :\ o_1 \in dom(\mu_{\delta_1}) \wedge o_2 \in dom(\mu_{\delta_2}) \wedge \Sigma'(o_1) = C\langle o'.gn' \ldots \rangle \wedge \Sigma'(o_2) = D\langle o'.gn' \ldots \rangle$
$\therefore dom(\mu_{\delta_1}) \cap dom(\mu_{\delta_2}) = \bullet$
$\therefore \mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2} \neq \texttt{race}$
$\therefore \mu' = [\mu'_\delta]\mu \neq \texttt{race}$ (TS1.3)

by IS2-1, IS2-2, E-LET-12:
$o \in dom(\mu) \cap dom(\mu_\delta) \implies$ (TS2)
$(\Sigma'(o) = C\langle o'.gn' \ldots \rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\, \texttt{inatomic}\ \langle o'.gn' \rangle \ldots \in e')$

by IS2-1,IS2-2:
μ_1 is well typed with respect to Σ_1
μ_2 is well typed with respect to Σ_2
$\therefore \mu'$ well typed with respect to Σ' (TS1.2)

by E-LET-12: δ, Δ does not change
$o.gn \in \delta \implies o.gn : gp \in \Delta$ (TS1.4)

by T-LET-12: $\Gamma|\Sigma'|\Delta \vdash \texttt{let}\ x = e'_1\ \texttt{in}\ e'_2 : T\ |\mathcal{G}'$ (TS1.1)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: (TS1)

Case E-LET-1 :
Proof is a sub-case of case E-LET-12, without the e_2 sub-expression step.

Case E-LET-2 :
Proof is a sub-case of case E-LET-12, without the e_1 sub-expression step.

Case E-LET-VALUE :
$\therefore e = \texttt{let}\ x = v\ \texttt{in}\ e_2$
$\therefore e' = [^v/_x]e_2$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|[^v/_x]e_2)$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash [^v/_x]e_2 : T\ |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$

$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\, \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{let}\ x = v\ \texttt{in}\ e_2)$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{let}\ x = v\ \texttt{in}\ e_2 : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$\Delta = \Delta_1, \Delta_R$

$\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2$

$\Gamma|\Sigma|\Delta_1 \vdash v : T_1\ |\mathcal{G}_1$ for some T_1

$\Gamma|\Sigma|\Delta_1, \Delta_R \vdash e_2 : T_2\ |\mathcal{G}_2$

by rule E-LET-VALUE:

$\mathcal{G}' = [^v/_x]\mathcal{G}_2$

$\Psi' = \Psi$

$\mu_\delta = \bullet$

by SUBSTITUTION:

$\Gamma, x : T_1, \Gamma'|\Sigma|\Delta_1, \Delta_R \vdash e_2 : T_2\ |\mathcal{G}_2 \implies \Gamma, [^v/_x]\Gamma'|\Sigma|[^v/_x]\Delta_1, \Delta_R \vdash [^v/_x](e_2 : T_2\ |\mathcal{G}_2)$ (TS1.1)

by E-LET-VALUE: $\mu_\delta = \bullet$

$\therefore \mu' = [\mu_\delta]\mu = \mu \neq \texttt{race}$ (TS1.3)

$\therefore dom(\mu) \cap dom(\mu_\delta) = \bullet$ (TS2)

WLOG: let $\Sigma' = \Sigma$

by AS2: $\mu' = \mu$ is well typed with respect $\Sigma' = \Sigma$ (TS1.2) by E-LET-VALUE: neither $\delta\Delta$ changes (TS1.4)

by AS5,AS6: $\Psi' = \Psi$ (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.5: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-FIELD-ASSIGN :
$\therefore e = v_r.f_i := o_v$
$\therefore e' = o_v$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|o_v)$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash o_v : T\ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\ \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|v_r.f_i := o_v)$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash v_r.f_i := o_v : T\ |\mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:
$T_i\ f_i \in fields(D)$
$o_r : D\langle\overline{gr}\rangle$
$\mathcal{G} = \{gr_0\}$
$gr_o : gp \in \Delta$ with $gp \in \{exclusive, protected\}$
$o_v : T_v$ with $T_v : T_i$
WLOG: let $o'.gn' = gr_0$

by E-FIELD-ASSIGN:

$o'.gn' \in \delta$
$o'.gn'@\_ \in \Psi$
$\Psi' = \Psi$
$\mathcal{G}' = \bullet$
$\mu_\delta = \langle o_r \mapsto D[\overline{f = v_f}]\rangle$

WLOG: let $\Sigma' = \Sigma$

by STORE-TYPING: $o_v : T_v \in \Sigma$ with $T_v <: T_i$
by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash o_v : T_v\ |\bullet$ (TS1.1)
by CONSTRUCTION: $\mu_\delta \neq \texttt{race} \implies \mu' = [\mu_\delta]\mu \neq \texttt{race}$ (TS1.2)
by CONSTRUCTION: $\mu' = [\mu_\delta]\mu = \mu$ (TS1.3)
by E-FIELD-ASSIGN: δ, Δ do not change (TS1.4)
by E-FIELD-ASSIGN,AS5,AS6: $\Psi = \Psi'$ (TS1.5, TS1.6)
by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

by CONSTRUCTION: $dom(\mu) \cap dom(\mu_\delta) = \{o_r\}$
by INVERSION, E-FIELD-ASSIGN:
$\Sigma(o_r) = D\langle o'.gn' \ldots\rangle$
$o'.gn' \in \delta$
$o'.gn' : gp \in \Delta \wedge gp \in \{exclusive, protected\}$
$\therefore o \in dom(\mu) \cap dom(\mu_\delta) \implies \Sigma(o) = C\langle o'.gn' \ldots\rangle \implies o'.gn' \in \delta \implies o'.gn' : gp$
$\wedge\ gp = shared \implies \exists\ \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e'$ (TS2)

Case E-NEW :
$\therefore e = \texttt{ new } C\langle\overline{v_g.gn_g}\rangle$
$\therefore e' = o_{new}$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|o_{new})$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash o_{new} : T\ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\ \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\texttt{ new } C\langle\overline{v_g.gn_g}\rangle)$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash \texttt{new}\ C\langle\overline{v_g.gn_g}\rangle : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$T = [^{\overline{o_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}]C\langle\overline{\alpha},\overline{b}\rangle$

$\mathcal{G} = \bullet$

by E-NEW:

$o'.gn' \in \delta$

$o'.gn'@\_ \in \Psi$

$groupDecls(C) = \overline{gn_c} \qquad \Psi' = \Psi, \overline{o_{new}.gn_c@U}$

$\mathcal{G}' = \bullet$

$\mu_\delta = \langle o_{new} \mapsto C[\overline{f = \texttt{null}}]\rangle$

by E-NEW:

$\mu_\delta \neq \texttt{race} \implies \mu' = [\mu_\delta]\mu$ (TS1.3)

δ, Δ do not change (TS1.4)

WLOG: let $\Sigma' = \Sigma, o_{new} : T$

by E-TRANS-N: $\mu' = [\mu_\delta]\mu$ is well typed with respect to Σ' (TS1.2)

by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash o_{new} : T\ |\mathcal{G}'$ (TS1.1)

by E-NEW,AS5,AS6: $\Psi' = \Psi, \{o_{new}.gn_c@U\}$

newly added access tokens are in unlocked state

$\therefore$ $\texttt{atomic} \rightarrow \texttt{inatomic}$ transmission could have happened so far (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

by CONSTRUCTION: $dom(\mu) \cap dom(\mu_\delta) = \{o_r\}$

by INVERSION, E-NEW:

$\Sigma(o_r) = D\langle o'.gn' \ldots\rangle$

$o'.gn' \in \delta$

$o'.gn' : gp \in \Delta \wedge gp \in \{exclusive, protected\}$

$\therefore o \in dom(\mu) \cap dom(\mu_\delta) \implies \Sigma(o) = C\langle o'.gn' \ldots\rangle \implies o'.gn' \in \delta \implies o'.gn' : gp$
$\wedge\ gp = shared \implies \exists\ \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e'$ (TS2)

Case E-CALL :
$\therefore e = v_r.m\langle\overline{v_g.gn_g}\rangle(\overline{v_p})$
$\therefore e' = [{}^{\overline{v_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}][{}^{\overline{v_p}}/_{\overline{x}}][{}^{v_r}/_{this}]e$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|[{}^{\overline{v_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}][{}^{\overline{v_p}}/_{\overline{x}}][{}^{v_r}/_{this}]e_b)$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash [{}^{\overline{v_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}][{}^{\overline{v_p}}/_{\overline{x}}][{}^{v_r}/_{this}]e_b : T\ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\, \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|v_r.m\langle\overline{v_g.gn_g}\rangle(\overline{v_p}))$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash v_r.m\langle\overline{v_g.gn_g}\rangle(\overline{v_p}) : T\ |\mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

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

by E-CALL:
$\overline{vg_g.gn_g} \in \delta$
$mbody(C,m) = \overline{\alpha}.\overline{x}.e_b \times \mathcal{G}_e$
$\mathcal{G}' = [{}^{\overline{v_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}][{}^{\overline{v_p}}/_{\overline{x}}][{}^{v_r}/_{this}]\mathcal{G}_e$
$\Psi' = \Psi$
$\mu_\delta = \bullet$
WLOG: let $\Sigma' = \Sigma$

by SUBSTITUTION: $\Gamma', \overline{x : T_x}, \texttt{this} : T_r, \overline{gr} : \mathbb{G}, \Gamma'|\Sigma|\Delta \vdash e : T_e \,|\mathcal{G}_e$
$\Rightarrow \Gamma, [^{\overline{v_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}][^{\overline{v_p}}/_{\overline{x}}][^{v_r}/_{this}]\Gamma'|\Sigma|[^{\overline{v_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}][^{\overline{v_p}}/_{\overline{x}}][^{v_r}/_{this}]\Delta \vdash [^{\overline{v_g.gn_g}}/_{\overline{\alpha},\overline{\beta}}][^{\overline{v_p}}/_{\overline{x}}][^{v_r}/_{this}](e : T_{result} \,|\mathcal{G}_e)$
(TS1.1)

by E-CALL: $\mu_\delta = \bullet \implies \mu' = [\mu_\delta]\mu = \mu$
μ' is well typed with respect to Σ' (TS1.2)
$\mu' \neq Race$ (TS1.3)
$dom(\mu) \cap dom(\mu_\delta) = \bullet$ (TS2)

by E-CALL: δ, Δ do not change (TS1.4)

by E-CALL, AS5, AS6: $\texttt{inatomic} \notin e_b$ because it is a runtime only construct (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-UNPACKGROUPSOF-REPLACE :
$\therefore e = \texttt{unpackGroupsOf } v_r \texttt{ in } e_{sub}$
$\therefore e' = \texttt{unpackGroupsOf } v_r \texttt{ in } e'_{sub}$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|\texttt{ unpackGroupsOf } v_r \texttt{ in } e'_{sub})$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash \texttt{ unpackGroupsOf } v_r \texttt{ in } e'_{sub} : T \,|\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists \texttt{ inatomic } \langle o.gn \rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic } \langle o.gn \rangle \ldots \in e$
(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
$(\Sigma(o) = C\langle o'.gn' \ldots \rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists \texttt{ inatomic } \langle o'.gn' \rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\texttt{ unpackGroupsOf } v_r \texttt{ in } e_{sub})$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash \texttt{ unpackGroupsOf } v_r \texttt{ in } e_{sub} : T \,|\mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists \texttt{ inatomic } \langle o.gn \rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic } \langle o.gn \rangle \ldots \in e$

by INVERSION:

$\Gamma|\Sigma \vdash v_r : D\langle\overline{gr}\rangle$
$\Delta = \Delta', (gr_0 : qp)$
$groupDecls(D) = \overline{gn}$
$\Delta'' = \Delta', (\overline{v_r.gn : qp'})$
$\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash e : T \mid \mathcal{G}_e$
$\mathcal{G} = (\{gr_o, \overline{r.gn}\} \oplus \mathcal{G}_e)$

by E-UNPACKGROUPSOF-REPLACE:
$\mathcal{G} = (\{v'.gn, \overline{v_r.gn}\} \oplus \mathcal{G}_e)$
$\delta = \delta', v'.gn$
$\delta'' = \delta', \overline{v_r.gn}$

Sub-Case T-UNPACKGROUPSOF-SHARED :
$qp \in \{shared, protected\} \Rightarrow qp' = shared$

Sub-Case T-UNPACKGROUPSOF-EXCLUSIVE :
If $qp \in \{exclusive\} \Rightarrow qp' = exclusive$

by IH: on $\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash_{wf} (\mu|\delta''|\Psi|\mathcal{G}_e|e_{sub})$ with $\Gamma|\Sigma|\Delta'' \vdash e_{sub} : T\ |\mathcal{G}_e$ where $\mu|\delta''|\Psi|\mathcal{G}_e \vdash e_{sub} \mapsto e'_{sub} \dashv \mu_\delta|\Psi'|\mathcal{G}'_e$
for some $\mathcal{G}'_2, \Sigma \subseteq \Sigma'$
(IS1.1) $\Gamma|\Sigma'|\Delta'' \vdash e'_{sub} : T\ |\mathcal{G}'_e$
(IS1.2) $\mu' \neq \texttt{race}$
(IS1.3) μ' is well typed with respect to Σ'
(IS1.4) $o.gn \in \delta'' \implies o.gn : gp \in \Delta''$
(IS1.5-2) $(o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \in e'_{sub}$
(IS1.6-2) $o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \in e'_{sub}$
(IS2-1) $o \in dom(\mu) \cap dom(\mu_{\delta_1}) \implies$
$(\Sigma_2(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta_1 \wedge o'.gn' : gp \in \Delta_1 \wedge (gp = shared \implies \exists\ \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e_{sub}))$

by IS1.2, IS1.3, IS1.5, IS1.6,IS2: (IS1.2, IS1.3, IS1.5, IS1.6,TS2)

by T-UNPACKGROUPSOF-EXCLUSIVE, T-UNPACKGROUPSOF-SHARED, E-UNPACKGROUPSOF-REPLACE:
$\Gamma|\Sigma'|\Delta \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e'_{sub} : T\ |\mathcal{G}'$ (IS1.1)

by IS1.4, E-UNPACKGROUPSOF-REPLACE:
$o.gn \in \delta \implies o.gn : gp \in \Delta$ (IS1.1)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-UNPACKGROUPSOF-NONE :
$\therefore e =$ `unpackGroupsOf` v_r `in` e_{sub}
$\therefore e' =$ `unpackGroupsOf` v_r `in` e'_{sub}

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|$ `unpackGroupsOf` v_r `in` $e'_{sub})$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash$ `unpackGroupsOf` v_r `in` $e'_{sub} : T\,|\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq$ `race`
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists$ `inatomic` $\langle o.gn\rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \ldots \in e$
(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists$ `inatomic` $\langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|$ `unpackGroupsOf` v_r `in` $e_{sub})$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash$ `unpackGroupsOf` v_r `in` $e_{sub} : T\,|\mathcal{G}$
(AS2) $\mu \neq$ `race`
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists$ `inatomic` $\langle o.gn\rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one `inatomic` $\langle o.gn\rangle \ldots \in e$

by INVERSION:
$\Gamma|\Sigma \vdash v_r : C\langle\overline{gr}\rangle$
$\Delta = \Delta', (gr_0 : qp)$
$groupDecls(C) = \overline{gn}$
$\Delta'' = \Delta', (\overline{v_r.gn : qp'})$
$\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash e : T \mid \mathcal{G}_e$
$G = (\{gr_o, \overline{r.gn}\} \oplus \mathcal{G}_e)$
WLOG: let $v'.gn = gr_0$

by E-UNPACKGROUPSOF-NONE:
$gr_0 \notin \delta$
$\mathcal{G} = (\{v'.gn\} \oplus \mathcal{G}_e)$
$\mathcal{G}' = (\{v'.gn\} \oplus \mathcal{G}'_e)$

by IH: on $\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash_{wf} (\mu|\delta''|\Psi|\mathcal{G}_e|e_{sub})$ with $\Gamma|\Sigma|\Delta'' \vdash e_{sub} : T\,|\mathcal{G}_e$ where $\mu|\delta''|\Psi|\mathcal{G}_e \vdash$

$e_{sub} \mapsto e'_{sub} \dashv \mu_\delta|\Psi'|\mathcal{G}'_e$

for some $\mathcal{G}'_2, \Sigma \subseteq \Sigma'$

(IS1.1) $\Gamma|\Sigma'|\Delta'' \vdash e'_{sub} : T\ |\mathcal{G}'_e$

(IS1.2) $\mu' \neq \texttt{race}$

(IS1.3) μ' is well typed with respect to Σ'

(IS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta'$

(IS1.5-2) $(o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \in e'_{sub}$

(IS1.6-2) $o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \in e'_{sub}$

(IS2-1) $o \in dom(\mu) \cap dom(\mu_{\delta_1}) \implies$

$(\Sigma_2(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta_1 \wedge o'.gn' : gp \in \Delta_1 \wedge (gp = shared \implies \exists\, \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e_{sub}))$

by IS1.2, IS1.3, IS1.5, IS1.6,IS2: (IS1.2, IS1.3, IS1.5, IS1.6,TS2)

by T-UNPACKGROUPSOF-EXCLUSIVE, T-UNPACKGROUPSOF-SHARED, E-UNPACKGROUPSOF-NONE:

$\Gamma|\Sigma'|\Delta \vdash \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e'_{sub} : T\ |\mathcal{G}'$ (IS1.1)

by IS1.4, E-UNPACKGROUPSOF-NONE:

$o.gn \in \delta \implies o.gn : gp \in \Delta$ (IS1.1)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-UNPACKGROUPSOF-VALUE :

$\therefore e = \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ v$

$\therefore e' = v$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|v)$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash v : T\ |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$

$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\, \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e_{sub})$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash$ $\texttt{unpackGroupsOf}\ v_r\ \texttt{in}\ e_{sub} : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$\Gamma|\Sigma \vdash v_r : C\langle\overline{gr}\rangle$

$\Delta = \Delta', (gr_0 : qp)$

$groupDecls(C) = \overline{gn}$

$\Delta'' = \Delta', (\overline{v_r.gn : qp'})$

$\Gamma, (\overline{v_r.gn : \mathbb{G}})|\Sigma|\Delta'' \vdash e : T \mid \mathcal{G}_e$

$G = (\{gr_o, \overline{r.gn}\} \oplus \mathcal{G}_e)$

by E-UNPACKGROUPSOF-VALUE:

$\mathcal{G}' = \bullet$

$\mu'_\delta = \bullet$

$\Psi' = \Psi$

WLOG: let $\Sigma' = \Sigma$

by STORE-TYPING: $v : T \in \Sigma$

by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash v : T\ |\bullet$ (TS1.1)

by E-UNPACKGROUPSOF-VALUE:

$\mu_\delta = \bullet \implies \mu' = [\mu_\delta]\mu \neq \texttt{race}$ (TS1.3)

$\mu' = \mu$ is well typed with respect to $\Sigma' = \Sigma$ (TS1.2)

$dom(\mu) \cap dom(\mu_\delta) = \bullet$ (TS2)

by AS1.4, AS1.5, AS1.6,E-UNPACKGROUPSOF-VALUE: d, Δ, Ψ do not change (TS1.4, TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-ATOMIC-STEP1 :
Follows the reasoning as the *E-UnpackGroupsOf-None* case, by allowing the sub-expression to execute code that does not depend on the aotmic permission.

Case E-ATOMIC-STEP2 :
Analog to case *E-Atomic-Step1*. Despite the fact that we have the necessary permission the data

group access token indicate the already another atomic block is executing. Therefore only allow sub-expression only to execute code that does not depend on the atomic permission.

Case E-ATOMIC-INATOMIC :
$\therefore e =$ `atomic` $\langle gr \rangle\ e_{sub}$
$\therefore e' =$ `inatomic` $\langle gr \rangle\ e'_{sub}$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|\ \texttt{inatomic}\ \langle gr \rangle\ e_{sub})$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash\ \texttt{inatomic}\ \langle gr \rangle\ e_{sub} : T\ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$
(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
$(\Sigma(o) = C\langle o'.gn' \ldots \rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\ \texttt{inatomic}\ \langle o'.gn' \rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{atomic}\ \langle gr \rangle\ e_{sub})$
by DEFINITION:
(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{atomic}\ \langle gr \rangle\ e_{sub} : T\ |\mathcal{G}$
(AS2) $\mu \neq \texttt{race}$
(AS3) μ is well typed with respect to Σ
(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$
(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$

by INVERSION:
$\mathcal{G} = (\{gr\} \oplus \mathcal{G}_e)$
$\Delta = \Delta', (gr : shared)$
$\Gamma|\Sigma \vdash gr : \mathbb{G}$
$\Gamma|\Sigma|(\Delta', gr : protected) \vdash_C e_{sub} : T \mid \mathcal{G}$

by E-ATOMIC-INATOMIC:
$\Psi = \Psi'', gr@U$
$\Psi' = \Psi'', gr@L$
$\mathcal{G}' = \mathcal{G}$
$\mu_\delta = \bullet$
WLOG: let $\Sigma' = \Sigma$

by AS2, E-ATOMIC-INATOMIC: $\mu_\delta = \bullet \implies \mu' = [\mu_\delta]\mu$

$\mu' = \mu \neq \texttt{race}$ (TS1.3)
$\mu' = \mu$ is well typed with respect to $\Sigma' = \Sigma$ (TS1.2)
$dom(\mu) \cap dom(\mu_\delta) = \bullet$ (TS2)

by T-INATOMIC:
$\Gamma|\Sigma'|\Delta \vdash \texttt{inatomic}\ \langle gr\rangle\ e_{sub} : T\ |\mathcal{G}'$ (TS1.1)

by E-ATOMIC-INATOMIC: δ, Δ do not change (TS1.4)

by AS1,AS5,AS6:
$gr@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle gr\rangle \ldots \in e$
$gr@\_ \notin \Psi'' \implies \nexists\, \texttt{inatomic}\ \langle gr\rangle \ldots \in e_{sub}$
$gr@L \in \Psi' \implies \nexists\, \texttt{inatomic}\ \langle gr\rangle \ldots \in e'$ (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-INATOMIC-STEP1 :
Follows a similar logic as *E-Atomic-Step2*. In this case the all permissions are passed to the sub-expression let the sub-expression take a step.

Case E-INATOMIC-VALUE :
$\therefore e = \texttt{inatomic}\ \langle gr\rangle\ v$
$\therefore e' = v$

TO SHOW:
(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|v)$ i.e.
(TS1.1) $\Gamma|\Sigma'|\Delta \vdash v : T\ |\mathcal{G}'$
(TS1.2) μ is well typed with respect to Σ'
(TS1.3) $\mu' \neq \texttt{race}$
(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$
(TS1.5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$
(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$
$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\, \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{inatomic}\ \langle gr\rangle\ e_{sub})$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash$ $\texttt{inatomic}\ \langle gr \rangle\ e_{sub} : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@\_ \notin \Psi) \implies \nexists\ \texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn \rangle \ldots \in e$

by INVERSION:

$\mathcal{G} = (\{gr\} \oplus \mathcal{G}_e)$

$\Delta = \Delta', (gr : shared)$

$\Gamma|\Sigma \vdash gr : \mathbb{G}$

$\Gamma|\Sigma|(\Delta', gr : protected) \vdash_C v : T \mid \mathcal{G}$

by E-INATOMIC-VALUE:

$gr \in \delta$

$\Psi = \Psi'', gr@L$

$\Psi' = \Psi''', gr@U$

$\mu_\delta = 0$

$\mathcal{G}' = \bullet$

WLOG: let $\Sigma' = \Sigma$

by STORE-TYPING: $v : T \in \Sigma$

by T-REFERENCE: $\Gamma|\Sigma'|\Delta \vdash v : T\ |\bullet$ (TS1.1)

by E-INATOMIC-VALUE:

$\mu_\delta = \bullet \implies \mu' = [\mu_\delta]\mu \neq \texttt{race}$ (TS1.3)

$\mu' = \mu$ is well typed with respect to $\Sigma' = \Sigma$ (TS1.2)

$dom(\mu) \cap dom(\mu_\delta) = \bullet$ (TS2)

by AS1.4 E-UNPACKGROUPSOF-VALUE: d, Δ, Ψ do not change (TS1.4)

by AS1,AS5,AS6:

$gr@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle gr \rangle \ldots \in e$

$gr@U \in \Psi' \implies \nexists\ \texttt{inatomic}\ \langle gr \rangle \ldots \in e'$ (TS1.5, TS1.6)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|e')$ (TS1)

Case E-SPLIT-12 :

$\therefore e = \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e_1 \parallel e_2$

$\therefore e' = \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e'_1 \parallel e'_2$

TO SHOW:

(TS1) $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu'|\delta|\Psi'|\mathcal{G}'|\ \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e'_1 \parallel e'_2)$ i.e.

(TS1.1) $\Gamma|\Sigma'|\Delta \vdash\ \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e'_1 \parallel e'_2 : T\ |\mathcal{G}'$

(TS1.2) μ is well typed with respect to Σ'

(TS1.3) $\mu' \neq \texttt{race}$

(TS1.4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(TS1.5) $(o.gn@U \in \Psi \vee o.gn@_- \notin \Psi) \implies \nexists\,\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS1.6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(TS2) $o \in dom(\mu) \cap dom(\mu_\delta) \implies$

$(\Sigma(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \implies \exists\,\texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e))$

by ASSUMPTION: $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|\ \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e_1 \parallel e_2)$

by DEFINITION:

(AS1) $\Gamma|\Sigma|\Delta \vdash\ \texttt{share}\ \langle\overline{gr}\rangle\ \texttt{between}\ e_1 \parallel e_2 : T\ |\mathcal{G}$

(AS2) $\mu \neq \texttt{race}$

(AS3) μ is well typed with respect to Σ

(AS4) $o.gn \in \delta \implies o.gn : gp \in \Delta$

(AS5) $(o.gn@U \in \Psi \vee o.gn@_- \notin \Psi) \implies \nexists\,\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

(AS6) $o.gn@L \in \Psi \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e$

by INVERSION:

$\{\overline{gp}\} \subseteq \{exclusive, shared\}$

$\Delta = \Delta_1, \Delta_2, \Delta_r, (\overline{gr : gp})$

$\Gamma|\Sigma|(\Delta_1, \overline{gr : shared}) \vdash_C e_1 : T_1\ |\mathcal{G}_1$

$\Gamma|\Sigma|(\Delta_2, \overline{gr : shared}) \vdash_C e_2 : T_2\ |\mathcal{G}_2$

$\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2)$

by E-SPLIT-12:

$\mathcal{G} = (\mathcal{G}_1 \parallel \mathcal{G}_2)$

$\delta_1 = \delta \cap required(\mathcal{G}_1)$ the sub-set of permission that are required by e_1

$\delta_2 = \delta \cap required(\mathcal{G}_2)$ the sub-set of permission that are required by e_2

$\Psi = \Psi_1, \Psi_2$ with $requiredTokens(e_1) \subseteq \Psi_1$ and $requiredTokens(e_2) \subseteq \Psi_2$

$\Psi' = \Psi'_1, \Psi'_2$

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

by ASSUMPTION:

$\Gamma|\Sigma|\Delta_1 \vdash_{wf} (\mu|\delta_1|\Psi_1|\mathcal{G}_1|e_1)$

$\Gamma|\Sigma|\Delta_2 \vdash_{wf} (\mu|\delta_2|\Psi_2|\mathcal{G}_2|e_2)$

by IH: on $\Gamma|\Sigma_1|\Delta_1 \vdash_{wf} (\mu_1|\delta_1|\Psi_1|\mathcal{G}_1|e_1)$ with $\Gamma|\Sigma|\Delta_1 \vdash e_1 : T_1 \;|\mathcal{G}_1$ where $\mu|\delta_1|\Psi_1|\mathcal{G}_1 \vdash e_1 \mapsto e'_1 \dashv \mu_{\delta_1}|\Psi'_1|\mathcal{G}'_1$

for some $\mathcal{G}'_1, \Sigma \subseteq \Sigma_1$

(IS1.1-1) $\Gamma|\Sigma_1|\Delta_1 \vdash e'_1 : T_1 \;|\mathcal{G}'_1$

(IS1.2-1) $\mu_1 \neq \texttt{race}$

(IS1.3-1) μ_1 is well typed with respect to Σ'

(IS1.4-1) $o.gn \in \delta_1 \implies o.gn : gp \in \Delta_1$

(IS1.5-1) $(o.gn@U \in \Psi_1 \vee o.gn@\_ \notin \Psi_1) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e'_1$

(IS1.6-1) $o.gn@L \in \Psi_1 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e'_1$

(IS2-1) $o \in dom(\mu) \cap dom(\mu_{\delta_1}) \implies$

$(\Sigma_1(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta_1 \wedge o'.gn' : gp \in \Delta_1 \wedge (gp = shared \implies \exists\, \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e_1))$

by IH: on $\Gamma|\Sigma_2|\Delta_2 \vdash_{wf} (\mu_2|\delta_2|\Psi_2|\mathcal{G}_2|e_2)$ with $\Gamma|\Sigma|\Delta_1 \vdash e_2 : T_1 \;|\mathcal{G}_1$ where $\mu|\delta_2|\Psi_2|\mathcal{G}_2 \vdash e_2 \mapsto e'_2 \dashv \mu_{\delta_2}|\Psi'_2|\mathcal{G}'_2$

for some $\mathcal{G}'_2, \Sigma \subseteq \Sigma_1$

(IS1.1-2) $\Gamma|\Sigma_2|\Delta_2 \vdash e'_2 : T \;|\mathcal{G}'_2$

(IS1.2-2) $\mu_2 \neq \texttt{race}$

(IS1.3-2) μ_2 is well typed with respect to Σ'

(IS1.4-2) $o.gn \in \delta_2 \implies o.gn : gp \in \Delta_2$

(IS1.5-2) $(o.gn@U \in \Psi_2 \vee o.gn@\_ \notin \Psi_2) \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \in e'_2$

(IS1.6-2) $o.gn@L \in \Psi_2 \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \in e'_2$

(IS2-1) $o \in dom(\mu) \cap dom(\mu_{\delta_1}) \implies$

$(\Sigma_2(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta_1 \wedge o'.gn' : gp \in \Delta_1 \wedge (gp = shared \implies \exists\, \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e_1))$

by UNIQUE ALLOCATE:

$\Sigma_1 = \Sigma \cup \Sigma'_1$ and $\Sigma_1 = \Sigma \cup \Sigma'_2$ with $dom(\Sigma'_1) \cap dom(\Sigma'_2) = \bullet$

$\therefore$ let $\Sigma' = \Sigma \cup \Sigma_1 \cup \Sigma_2$

$dom(\Psi'_1) \cap dom(\Psi'_2) = \bullet$

by E-SPLIT-12: $\Psi' = \Psi'_1, \Psi'_2$

by IS1.5-1, IS1.5-2:

$(o.gn@U \in \Psi' \vee o.gn@\_ \notin \Psi') \implies \nexists\, \texttt{inatomic}\ \langle o.gn\rangle \ldots \in e'$ (TS1.5)

by IS1.6-1, IS1.6-2:

$o.gn@L \in \Psi' \implies \exists$ exactly one $\texttt{inatomic}\ \langle o.gn\rangle \ldots \in e'$ (TS1.6)

by E-SPLIT-12: $\delta = \delta_1, \delta_2$

$\therefore\ \nexists o_1, o_2 : o_1 \in dom(\mu_{\delta_1}) \wedge o_2 \in dom(\mu_{\delta_2}) \wedge \Sigma'(o_1) = C\langle o'.gn' \ldots\rangle \wedge \Sigma'(o_2) =$

$D\langle o'.gn' \ldots\rangle$

$\therefore dom(\mu_{\delta_1}) \cap dom(\mu_{\delta_2}) = \bullet$

$\therefore \mu_\delta = \mu_{\delta_1} \uplus \mu_{\delta_2} \neq \texttt{race}$

$\therefore \mu' = [\mu'_\delta]\mu \neq \texttt{race}$ (TS1.3)

by IS2-1, IS2-2, E-LET-12:

$o \in dom(\mu) \cap dom(\mu_\delta) \implies$ (TS2)

$(\Sigma'(o) = C\langle o'.gn' \ldots\rangle \wedge o'.gn' \in \delta \wedge o'.gn' : gp \in \Delta \wedge (gp = shared \Longrightarrow \exists\ \texttt{inatomic}\ \langle o'.gn'\rangle \ldots \in e')$

by IS2-1,IS2-2:

μ_1 is well typed with respect to Σ_1

μ_2 is well typed with respect to Σ_2

$\therefore \mu'$ well typed with respect to Σ' (TS1.2)

by E-SPLIT-12: δ, Δ does not change

$o.gn \in \delta \implies o.gn : gp \in \Delta$ (TS1.4)

by T-SPLIT-12: $\Gamma|\Sigma'|\Delta \vdash e' : T\ |\mathcal{G}'$ (TS1.1)

by TS1.1, TS1.2, TS1.3, TS1.4, TS1.5, TS1.6: (TS1)

Case E-SPLIT-1 :
Follows the same approach as case *E-Split-12* with the the difference that the evaluation of e_2 is not considered.

Case E-SPLIT-2 :
Follows the same approach as case *E-Split-12* with the the difference that the evaluation of e_1 is not considered.

Case E-SPLIT-VALUE :
Follows the same approach as case *E-UnpackGroupsOf-Value*.

■

B.2.2 Progress Proof

Proof (Progress) by induction on $\Gamma|\Sigma|\Delta \vdash_{wf} (\mu|\delta|\Psi|\mathcal{G}|e)$ with $\Gamma|\Sigma|\Delta \vdash_C e : T\ |\mathcal{G}$.

Case T-UNPACKGROUPSOF-EXCLUSIVE :
$e = \texttt{unpackGroupsOf}\ r\ \texttt{in}\ e_1$
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Case e_1 is value :
by E-UNPACKGROUPSOF-VALUE: $\mu|\delta|\Psi|\mathcal{G} \vdash$ $\texttt{unpackGroupsOf}\ r\ \texttt{in}\ v_1 \mapsto v_1 \dashv \bullet|\Psi|\bullet$
$\therefore e \mapsto e'$ takes a step

Sub-Case $e_1 \mapsto e_1'$ takes a step :
by E-UNPACKGROUPSOF-REPLACE:
$\mu|\delta|\Psi|\mathcal{G} \vdash$ $\texttt{unpackGroupsOf}\ r\ \texttt{in}\ e_1 \mapsto$ $\texttt{unpackGroupsOf}\ r\ \texttt{in}\ e_1 \dashv \mu_\delta|\Psi'|G'$
$\therefore e \mapsto e'$ takes a step

Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-UNPACKGROUPSOF-SHARED :
Symmetric to the T-UNPACKGROUPSOF-EXCLUSIVE case.

Case T-SPLIT :
$e = \texttt{share}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2$
by CANONICAL-FORM: $r_i = \texttt{null}$ or $r_i = o$

Sub-Case $\exists i : r_i = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $\forall i : r_i \neq \texttt{null}$:
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Sub-Case e_1 is value :
by IH: e_2 is value | e_2 takes a step | e_2 stops with null-dereference | e_2 waits for resources

Sub-Sub-Sub-Case e_2 is value :
by E-SPLIT-VALUE:
$\mu|\delta|\Psi|\mathcal{G} \vdash$ $\texttt{share}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ v_1 \parallel v_2 \mapsto \texttt{null} \dashv \bullet|\Psi|\bullet$ $\therefore e \mapsto e'$ takes a step

Sub-Sub-Sub-Case $e_2 \mapsto e_2'$ takes a step :
by E-SPLIT-2:
$\mu|\delta|\Psi|\mathcal{G} \vdash$ $\texttt{share}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto$ $\texttt{share}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2' \dashv \mu_\delta|\mathcal{G}';|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Sub-Case e_2 stops with null-dereference :
Then e stops with null-dereference.

Sub-Sub-Sub-Case e_2 waits for resources :
Then e waits for resources.

Sub-Sub-Case $e_1 \mapsto e_1'$ takes a step :
by E-SPLIT-1:
$\mu|\delta|\Psi|\mathcal{G} \vdash$ $\texttt{share}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1 \parallel e_2 \mapsto \texttt{share}\ \langle\overline{r.gn}\rangle\ \texttt{between}\ e_1' \parallel e_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-ATOMIC :
$e = \texttt{atomic}\ \langle r.gn\rangle\ e_1$
by CANONICAL-FORM: $r_i = \texttt{null}$ or $r_i = o$

Sub-Case $\exists i : r_i = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $\forall i : r_i \neq \texttt{null}$:
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Sub-Case e_1 is value :
by E-ATOMIC-INATOMIC:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{atomic}\ \langle r.gn\rangle\ v_1 \mapsto \texttt{inatomic}\ \langle r.gn\rangle v_1 \dashv \bullet|\Psi'|\bullet$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-INATOMIC :
$e = \texttt{inatomic}\ \langle r.gn\rangle\ e_1$
by CANONICAL-FORM: $r_i = \texttt{null}$ or $r_i = o$

Sub-Case $\exists i : r_i = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $\forall i : r_i \neq \texttt{null}$:
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Sub-Case e_1 is value :
by E-INATOMIC-VALUE:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{inatomic}\ \langle r.gn\rangle\ v_1 \mapsto v_1 \dashv \bullet|\Psi'|\bullet$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-LET :
$e = \texttt{let } x = e_1 \texttt{ in } e_2$
by IH: e_1 is value | e_1 takes a step | e_1 stops with null-dereference | e_1 waits for resources

Sub-Case e_1 is value :
by E-LET-VALUE:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{let } x = v_1 \texttt{ in } e_2 \mapsto [^{v_1}\_e_2] \dashv \bullet|\Psi|\bullet$
$\therefore e \mapsto e'$ takes a step

Sub-Case $e_1 \mapsto e_1'$ takes a step :
by E-LET-1:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \mapsto \texttt{let } x = e_1 \texttt{ in } e_2 \dashv \mu_\delta|\Psi'|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Case e_1 stops with null-dereference :
Then e stops with null-dereference.

Sub-Case e_1 waits for resources :
Then e waits for resources.

Case T-REFERENCE :
$e = r$
by CANONICAL-FORM: $r = \texttt{null}$ or $r = o$

Sub-Case $r = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $r \neq \texttt{null}$:
The e is value.

Case T-FIELD-READ :
$\therefore e = r.f_i$
by CANONICAL-FORM: $r = \texttt{null}$ or $r = o$

Sub-Case $r = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $r \neq \texttt{null}$:

Sub-Sub-Case $r.gn \in \delta$:
by E-FIELD-READ:
$\mu|\delta|\Psi|\mathcal{G} \vdash r.f_i \mapsto v_i \dashv \mu_\delta|\Psi|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case $r.gn \notin \delta$:
Then e is waiting for resources.

Case T-FIELD-ASSIGN :
$\therefore e = r.f_i := r_v$
by CANONICAL-FORM: $r = \texttt{null}$ or $r = o$

Sub-Case $r = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $r \neq \texttt{null}$:

Sub-Sub-Case $r.gn \in \delta$:
by E-FIELD-READ:
$\mu|\delta|\Psi|\mathcal{G} \vdash r.f_i := r_v \mapsto v_i \dashv \mu_\delta|\Psi|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Sub-Sub-Case $r.gn \notin \delta$:
Then e is waiting for resources.

Case T-NEW :
$e = \texttt{new}\ C\langle\overline{r.gn}\rangle()$
by CANONICAL-FORM: $r_i = \texttt{null}$ or $r_i = o$

Sub-Case $\exists i : r_i = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $\forall i : r_i \neq \texttt{null}$:
by E-NEW:
$\mu|\delta|\Psi|\mathcal{G} \vdash \texttt{new}\ C\langle\overline{r.gn}\rangle() \mapsto o \dashv \mu_\delta|\Psi'|\mathcal{G}'$
$\therefore e \mapsto e'$ takes a step

Case T-CALL :
$e = r_r.m\langle r_g.gn\rangle(r_p)$
by CANONICAL-FORM: $r_i \in \{r_r, \overline{r_g}\} \implies r_i = \texttt{null}$ or $r_i = o$

Sub-Case $\exists i : r_i = \texttt{null}$:
Then e stops with null-dereference.

Sub-Case $\forall i : r_i \neq \texttt{null}$:

Sub-Sub-Case $\exists r_g \notin \delta$:
Then e waits for resources.

Sub-Sub-Case $\forall r_g \in \delta$:
by E-CALL:

$$\mu|\delta|\Psi|\mathcal{G} \vdash r_r.m\langle r_g.gn\rangle(r_p) \mapsto [\overline{r_g.gn}/\overline{\alpha}][\overline{r_p}/\overline{x}][{r_r}/{this}]e_b \dashv \bullet|\Psi|\mathcal{G}_b$$

$\therefore e \mapsto e'$ takes a step

■

References

N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2008.

K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of nesl. In *International Conference on Functional Programming*, May 1996.

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

J. Boyland. Checking Interference with Fractional Permissions. In *Static Analysis: 10th International Symposium*, 2003.

D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

K. R. M. Leino. Data groups: specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Principles of Programming Languages*, 2008.

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

S. Stork, P. Marques, and J. Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Onward!*, 2009.