Diploma Thesis

TIFI+: A Type Checker for Object Immutability with Flexible Initialization

Günther Noack

March 29, 2010

Department of Computer Science
Software Technology Group
University of Kaiserslautern

Advisors: Prof. Dr. Arnd Poetzsch-Heffter
M. Sc. Yannick Welsch

Erklärung

Ich versichere hiermit, dass ich die vorliegende Diplomarbeit mit dem Thema

> TIFI+: A Type Checker for Object Immutability with Flexible Initialization

selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

(Ort, Datum) (Unterschrift)

Acknowledgements

First of all, I would like to thank Prof. Dr. Arnd Poetzsch-Heffter for making this thesis possible and for his advice. Many thanks also go to my supervisor Yannick Welsch for the many insightful discussions, which were a great help.

I would also like to thank Christian Haack for his valuable support in understanding the "Type-Based Object Immutability with Flexible Initialization" paper, which greatly helped me getting up to speed in the early phases of this thesis.

My special thanks go to Mathias Scheffe, Reiner Hüchting and Sabine Proll, who proof-read large parts of this thesis and spotted many errors.

Further, many thanks go to the Software Technology Group for the enjoyable atmosphere and the many interesting conversations, which were a great learning opportunity.

Contents

1 Introduction

In programming languages with mutable state, changes to aliased objects or data structures can be hard to coordinate and can lead to unexpected and incorrect program behavior. Especially in multi-threaded scenarios, where lock-based mechanisms are often employed for this, the coordination can become very challenging.

Immutability is an easy way to ensure referential transparency to objects and avoid the whole problem of aliasing up-front. Using current object oriented programming languages, immutability is most often ensured at the class level, for example using the rules outlined in [5]. In fact, many object oriented languages already use immutable classes for elementary data structures like strings (Java) or numbers (Smalltalk, Ruby).

Nevertheless, in practice one is often faced with systems not designed with immutability in mind. At the same time however, as software gets more and more complex, and optimization for multi-core architectures becomes more and more important, not having to care about mutable state is an increasingly valuable advantage.

In order to circumvent aliasing of the ubiquitous mutable objects in contemporary programming languages, implementers often use defensive programming techniques like defensive copying [5] or giving out wrapper objects which prohibit modifications, e.g. using Java's `Collections.unmodifiableSet()` and related helper methods.

Although this approach allows to avoid shared mutable state, it is also clear that its dynamic nature can have an impact on program performance and memory usage. Making immutability an explicitly supported concept in programming languages can help to avoid these kinds of helper techniques, as well as faciliate reasoning about immutability for implementers. In fact, the dynamically typed programming languages Ruby [20] and ECMAScript 5 [12] already provide support for *freezing* object state, disallowing further modifications to individual objects.

Another way to provide this language support are *static type systems for immutability*, which support implementers in statically reasoning about which objects are immutable, and which help proving program behavior with respect to the immutability aspect.

Furthermore, unlike dynamic language support for immutability, static type systems allow to completely avoid runtime overhead and can be built to require no modifications to existing runtime environments, which is a significant advantage in deployment.

This thesis describes a formalization and a prototype implementation of the TIFI+ type system for object immutability.

Using this type system, it is possible for users of objects to statically know not only whether they are allowed to modify the object, but also whether the object can be modified by others over alias references.

In previous type systems for object immutability, like Immutability Generic Java (IGJ), it has turned out to be a challenge to safely guarantee the construction of immutable objects and their associated object graphs. TIFI+ improves on previous approaches by implementing the *Flexible Initialization* mechanism proposed in [10]. Flexible Initialization combines static tracking of object's initialization phases with static polymorphism over immutability parameters, which turns out to be a powerful combination for object and object graph construction.

The rest of this thesis is organized as follows: Chapter 2 gives an informal introduction to the TIFI+ immutability and programming model. Chapter 3 shows motivational code examples to illustrate some more sophisticated application scenarios. Chapter 4 formalizes the programming model by giving an operational semantics for a restricted Java-like language. Chapter 5 presents TIFI+'s type inference / type checking algorithm. The section concludes with an informal soundness proof to show that type correct programs cannot mutate immutable objects. Chapter 6 highlights conceptual differences between the TIFI+ formalization language and Java, and points out strategies to map the type system's concepts and algorithms to Java. Chapter 7 gives an overview over the actual TIFI+ implementation and its integration into the Java compiler. Chapter 8 compares TIFI+ on a conceptual level with similar immutability type systems available as extensions to Java. Finally, Chapter 9 concludes with a retrospective look on the type system, design decisions in the programming model and the implementation, and gives an outlook on possible ways for improvement and integration with similar models.

2 TIFI+

In [10] and [11], Haack and Poll describe an immutability type system which can be used to provide guarantees for class, object and reference immutability. This type system's major difference over similar type systems like *Immutability Generic Java* (IGJ) [24] is its explicit support for object construction, *"Flexible Initialization"*, which is realized by introducing special types for uninitialized objects and qualifier polymorphism.

In order to distinguish between the type system as proposed in [10] and our own extension, we will use different names for them in the following chapters. The name "TIFI" (Type system for Immutability with Flexible Initialization) will be used for the original version from Haack and Poll's paper, "TIFI+" for our extension and adaption to the Java programming language.

2.1 Overview

The TIFI+ type system is a static type system to ensure object immutability in Java-like languages.

It is crucially important to understand that the TIFI+ type system does not replace Java's usual type system, but rather augments it with a set of *orthogonal immutability types* (access qualifiers). Each object reference does thus not only have its usual type, but also an additional immutability type (access qualifier) describing the immutability of the referenced object.

TIFI+ is designed to have the following properties:

Static Programs using the TIFI+ type system require no additional runtime type information to ensure immutability.

Supports initialization phase Unlike many other type systems for immutability, TIFI+ explicitly supports management of an object's immutability life cycle. In TIFI+, each object's life cycle starts with an initialization phase, within which it may be modified, even if it is immutable later.

Modular type checking The TIFI+ type checker can be run independently on each method. It is up to the programmer to annotate a method's requirements and behavior with respect to immutability properties into each method's signature.

Object immutability TIFI+ is able to ensure that objects can never be changed again. Note that this is different to ensuring that objects can not be changed over a specific object reference. The exact meaning of this is discussed in Section 2.3.

2.2 Notion of Immutability

Observational immutability

The intuitive definition of immutability is that an immutable object cannot change its state. In [17, Chapter 4], under the assumption that equality is reasonably defined, it is explained as follows:

> An object X is called (observationally) immutable if after [its initialization phase] any two invocations
>
> $$X.m(p_1, \ldots, p_n) \qquad \text{and} \qquad X.m(q_1, \ldots, q_n)$$
>
> with p_i equals q_i $(1 \leq i \leq n)$ either
>
> - yield equal results or
> - throw equal exceptions or
> - both do not terminate.

In other words, two equal method invocations on the object will always result in equal behavior, independent of all interactions that can happen between those invocations.

Note that this definition still allows objects to change their inner state, as long as this change is not exposed to the outside. A typical use case for this is result caching, as it is for example done in Java's `String` class for the hash code calculation. Note that the above definition of equal method behavior does not require an equal method execution time.

TIFI+'s notion of immutability

The TIFI and TIFI+ notion of immutability is that the field values of an immutable object X may not be changed. Additionally, if transitively reachable objects belong to X's abstract state, their modification can be prohibited as well (Transitive immutability guarantees). Transitively reachable objects can also be excluded from X's abstract state, e.g. for caching.

It is up to the type system user to ensure that his classes' exposed interface with respect to immutability matches the definition of observational immutability above. Although in principle, TIFI+ can be misused so that immutable-typed objects actually expose mutable state, it can be assumed that in practice, only observationally immutable objects will be considered as immutable by the type system.

2.3 Kinds of Immutability

The TIFI type system [10] distinguishes three kinds of immutability:

Class immutability This form of immutability ensures that all instances of a class flagged as immutable are immutable.

Reference immutability Reference immutability prohibits the holder of a reference to mutate the referenced object over that reference.

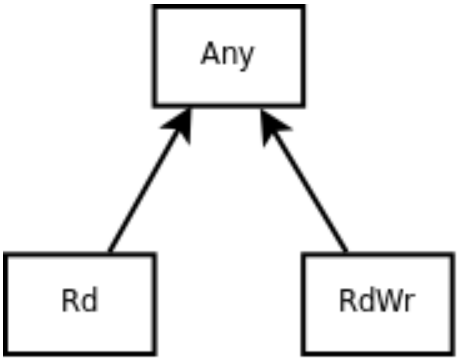

Figure 1: Type hierarchy for constructed objects

Object immutability Object immutability prohibits the holder of a reference to mutate the referenced object, but it also guarantees that the referenced object cannot be mutated by others as well.

It is important to note the difference between reference and object immutability. Using a reference immutable variable, mutating the referenced object is forbidden. However, the referenced object may still be mutable, so it is possible that other objects or methods hold non-immutable-typed references to the same object.

Object immutability, in contrast, is a much stronger guarantee: When holding a reference typed as "object immutable", this is a stronger guarantee about the referenced object: Not only can the reference holder not mutate the object any more, but it is even guaranteed that no one else can hold a reference to it allowing him to mutate the object, so the referenced object can never change its abstract state again.

2.4 Qualifier Hierarchy for Initialized Objects

Access qualifiers

We use the notion of *access qualifiers* for immutability types, which is also used in [10]. First, let's consider the qualifier hierarchy for constructed objects (Figure 1). The full qualifier hierarchy, which also considers object initialization issues, will be presented in Section 2.6.

In this type hierarchy, *RdWr* is the type used for mutable objects. This is the default qualifier, so that existing Java code keeps on working. The qualifier *Any* provides reference immutability. When a reference variable is typed as *Any*, the referenced object cannot be mutated using this reference.

Object immutability can be ensured using the *Rd* type. If a reference is typed as *Rd*, it cannot be used to mutate the referenced object. In contrast to *Any*, though, there is also a *guarantee to the holder* of that reference that no one else holds a writeable reference to the object.

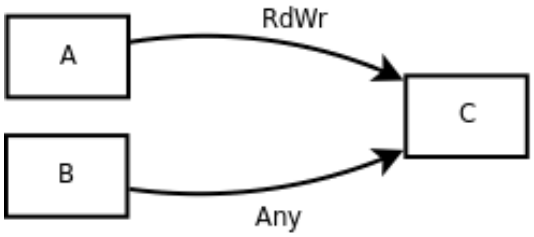

Figure 2: Three connected objects

Example 1 (Reference immutability)**.** Figure 2 depicts the three objects A, B, C, where A and B hold a reference to C. Let's view the situation from the viewpoint of B and consider the reference from B to C to be typed as reference-immutable (*Any*). For B, it cannot be statically ensured that the object behind the *Any* reference is writeable, so modifying C over this reference is obviously forbidden. However, A holds a *RdWr* reference to C, so A may still modify it.

□

An important implication of object immutability is that it is not sufficient if a type system is able to know one immutability type per class. Instead, classes can have both mutable and immutable instances, so a type system will need to track access qualifiers for individual objects. A simple way to think about this is to imagine how this would be solved in a dynamically typed language: In such a language, each object could be tagged with an access qualifier, which is for instance stored along with the object on the heap. In fact, Ruby's *Freezing* [20] and ECMAScript 5's *Sealing / Freezing* [12] provide such mechanisms.

Tagged objects as model for object immutability

It is often helpful to recall this model of *tagged objects* for reasoning about the TIFI+ type checker. It is interesting to note that there is no need for objects tagged as *Any* in the tagged object model, as the *Rd* qualifier already prohibits modifications.

2.5 Field Annotations

In TIFI+, all fields need to be annotated with an immutability annotation. The annotation may be one of `@Rd`, `@RdWr`, `@Any` and `@MyAccess`. If no annotation is given, the default annotation `@RdWr` is assumed.

The annotations `@Rd`, `@RdWr` and `@Any` behave as expected. Fields annotated with them can only reference objects of compatible (statically known) access qualifiers.

Deep immutability

The annotation `@MyAccess` is used for realizing deep immutability. An object referenced over a `@MyAccess`-annotated field needs to be tagged with the same qualifier as the referring object.

```
abstract class Tree { }
final class Leaf { }
final class Branch {
  @MyAccess Tree left;
  @MyAccess Tree right;
}
```

Listing 1: A tree class using `@MyAccess`

```
class Person {
  String name;
  Person(String name) {
    this.name = name;
  }
}
```

Listing 2: A simple constructor.

Example 2 (Deep immutability)**.** A tree data structure may be defined as seen in Listing 1.

Instances of the `Branch` class may now only have subtrees with the same access qualifier as the instance itself. If for example a `Branch` instance is immutable, all its subtrees will also be immutable.

□

2.6 Qualifier Hierarchy

A critical point in immutability type systems for Java is object construction: Even when a programmer intends an object to be immutable throughout its whole object life cycle, there is always a short phase of object construction during which the object needs to allow modifying its fields in order to set initial values other than `null`.

Example 3. Listing 2 shows a simplified `Person` class. We might want to make all instances immutable, but during the execution of the constructor, it still needs to be allowed to write the `name` field.

□

Object life cycle phases

We divide the object life cycle into the *object construction phase* and the *fully initialized phase* (Figure 3).

In order to keep track of objects in their construction phase, we introduce the access qualifiers *Fresh*(n). Note that by parameterizing *Fresh* over the *Fresh token* n, we can create an infinite amount of distinguishable Fresh qualifiers.

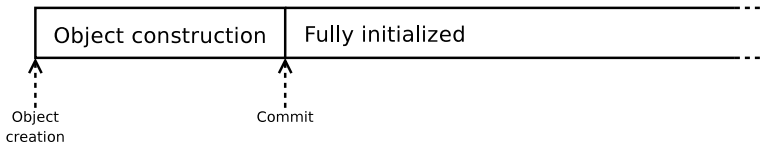

Figure 3: Object life cycle

Figure 4: Full TIFI qualifier hierarchy

Adding these *Fresh*(...) qualifiers results in the qualifier hierarchy shown in Figure 4. The Fresh qualifiers are kept completely separate from the qualifiers *Rd*, *RdWr* and *Any* for fully initialized objects.

In their construction phase, references to Fresh objects can thus never be used in places where fully initialized Java objects are expected. Especially, this prevents programmers from passing out references to objects in their construction phase to normal, unannotated Java methods, which by default only allow *RdWr*-tagged objects.

2.7 Constraints on Heap-Internal References

Having completed our type hierarchy in the previous section, we can have a closer look at the `@MyAccess` constraint's implications for references within the heap.

When an object o_1 has a field storing a reference to o_2, we say that o_1 references o_2.

- *Rd* and *RdWr* objects can be referenced by objects of all access qualifiers. This can easily be done by annotating these fields using the `Rd`, `RdWr` and `Any` annotations.
- *Fresh*(n) objects can only be referenced by other *Fresh*(n) objects. (Note that the Fresh token needs to be the same.)

These two insights are visualized in Figure 5. In the depiction, the heap is divided into the group of constructed objects (white), and two groups of Fresh objects (blue and purple). Arrows between these groups indicate the possibility of one object of the source group referencing an object of the

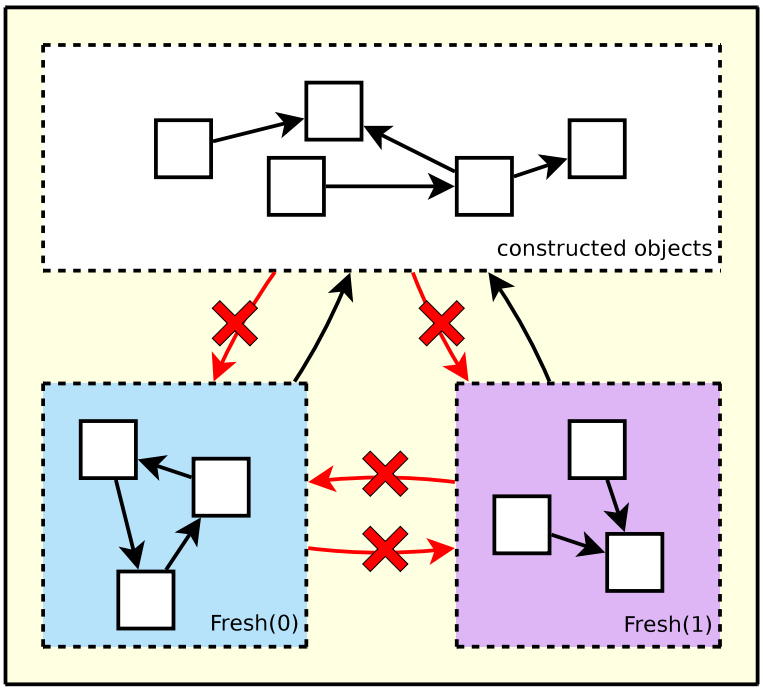

Figure 5: Allowed and disallowed references within the heap

target group. As we can see, every object may point to objects in the group of constructed objects. Within these objects groups, references between all objects are possible.

Fresh reachability from outside the heap

It is interesting to note that *Fresh*(n) objects can only be reachable via other *Fresh*(n) objects. An implication of this is that when a method instance has no reference to an object tagged with the *Fresh*(n) qualifier, it will never be able to reach one. The set of Fresh qualifiers whose objects are visible to a method is limited to the set of Fresh qualifiers directly passed as arguments to the method.

2.8 Transition from Construction Phase to Fully Initialized

Commit

The point in an object's life cycle at which the object changes from the *construction phase* to being *fully initialized* is called *Commit*. In a system with runtime information about objects' access qualifiers, the commit operation replaces a *Fresh*(n) qualifier tagged to objects with another qualifier.

The commit operation can be seen as a special additional Java statement with a source and a target qualifier as arguments. The source qualifier is the Fresh qualifier to be replaced, the target qualifier is the qualifier to be used instead of the old one. Note that a commit does not just change one object, but actually changes the tagged qualifiers of all objects tagged with the source qualifier.

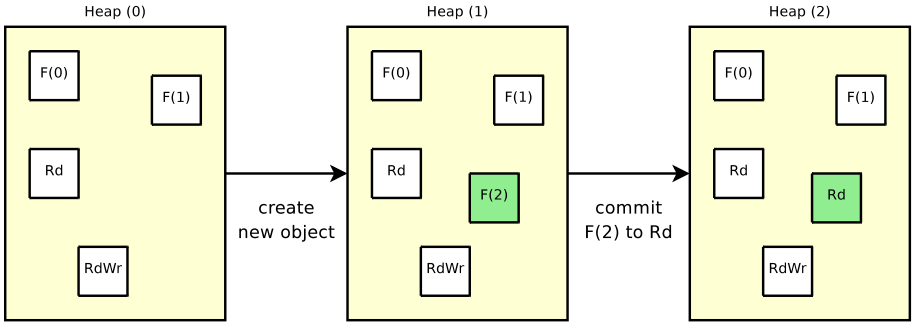

Figure 6: Updating the heap in a simple commit operation.

```
class Person {
  @MyAccess Person roommate;
  // ...
}
```

Listing 3: A person class with a roommate field

Example 4 (Simple case)**.** In the most simple case, the commit operation changes only one object. Figure 6 depicts the heap configurations between execution steps. At the start of the program, a few objects of different types populate the heap (Heap 0). Creating a new object updates the heap to contain a new object, which is initially tagged with a new Fresh type (here: *Fresh*(2)). The new object is not used by any other object yet (Heap 1). Before using the object in a context where a fully initialized object is expected, the program commits the object to *Rd* (Heap 2).

□

Example 5 (Committing Fresh qualifiers to Fresh qualifiers)**.** It is not strictly necessary to commit a Fresh qualifier to *Rd*, *RdWr* or *Any*. In fact, Fresh qualifiers are often also committed to other Fresh qualifiers, thereby merging two Fresh groups into one.

Consider a `Person` class with a `roommate` field annotated as `@MyAccess`, as shown in Listing 3.

So let's assume we created two Fresh `Person` instances `ernie`, tagged as *Fresh*(e), and `bert`, tagged as *Fresh*(b), which are ultimately supposed to be committed as immutable. But before doing that, we would like to set their `roommate` fields to each other, as shown in Listing 4.

Recall that in `@MyAccess` fields, only references to objects with the same tagged access qualifier as the referring object can be stored. So before being able to set `ernie`'s `roommate` field, we need to ensure `ernie` and `bert` have the same access qualifier.

```
ernie.roommate = bert;
bert.roommate = ernie;
```

Listing 4: Assigning `ernie` and `bert`'s roommate fields

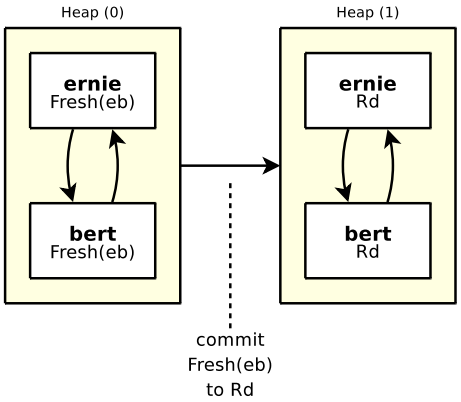

Figure 7: Making both `ernie` and `bert` immutable at once

This can be done without already making them both immutable by committing *Fresh*(e) to *Fresh*(b) before setting `ernie`'s field. To see why this is needed, consider the alternatives when committing both *Fresh*(e) and *Fresh*(b) to other qualifiers:

Commit to RdWr: This allows us to create the circular object graph, but prevents us from making the two objects immutable later.

Commit to Rd: Now both objects are directly immutable, which they are ultimately supposed to be. However, if they're immutable, we cannot set their `roommate` fields any more.

□

Example 6 (Committing multiple objects at once)**.** Recall the circular object graph created in Example 5: The two objects `ernie` and `bert` point to each other over `@MyAccess`-annotated `roommate` fields. In order not to violate the `@MyAccess` constraint, they both have the same Fresh qualifier, *Fresh*(eb).

Consider how to make both of these objects immutable: The naive approach, making them immutable one after another, is clearly infeasible: No matter which of the two objects would be re-tagged as *Rd* first, the resulting object graph would directly break the `@MyAccess` constraint.

In order to properly commit the objects in an object graph connected using `@MyAccess`-annotated fields, we can thus commit all objects at once

```
Rd Person makeErnieAndBert() {
  Person ernie = new Person();
  Person bert = new Person();
  // commit Fresh(e) to Fresh(b)
  ernie.roommate = bert;
  bert.roommate = ernie;
  // commit Fresh(b) to Rd
  return ernie;
}
```

Listing 5: The `makeErnieAndBert()` factory method with comments indicating commits

by committing *Fresh*(*eb*) to *Rd*. Figure 7 depicts the heap before and after the commit. Note that because of the atomicity of this step, no intermediate inconsistent heap state with respect to the `@MyAccess` constraint can occur.

□

2.9 Flexible Initialization

A major advantage of TIFI and TIFI+ is its explicit support for object construction. Unlike many other type systems, TIFI has an explicit qualifier (*Fresh*(...)) for objects in their construction phase, which is used to statically track the state of object construction. This tracking is what finally allows us to extends an object's initialization phase even beyond the execution of a constructor.

This is especially helpful when constructing circular object graphs like the graph in Example 5 and Example 6.

Example 7 (Factory method)**.** Putting Example 5 and Example 6 together, we can write a factory method `makeErnieAndBert()` creating the two connected objects (Listing 5).

Note that we assume the two constructor calls to return Fresh objects.

□

2.9.1 Modular Type Checking

It is generally a desirable property of Java type systems to allow modular type checking. In our case, we would like to be able to apply the type checker to each of a program's methods individually and independent of all other methods. It is reached by annotating each type occurring in a method signature with a suitable access qualifier, including the type of the implicit parameter `this`.

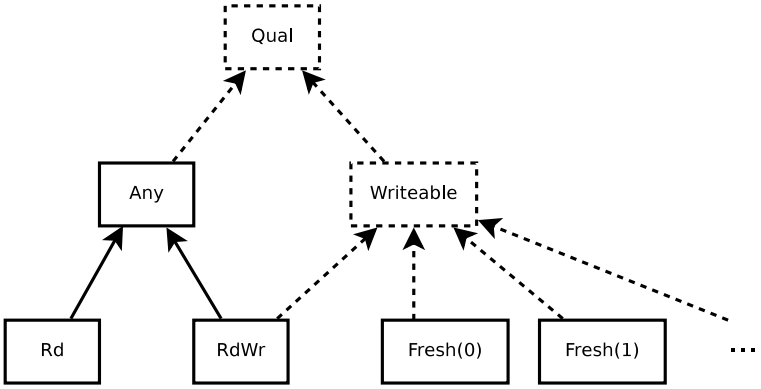

Figure 8: Qualifier hierarchy with upper bounds

Example 8. For example, a method could have the signature

```
Rd Bill repairCar(RdWr Car car) Any;
```

The method takes a mutable car object and returns an immutable bill object, so that the caller cannot modify the bill afterwards. The *Any* annotation after the parameter list is the receiver object's expected immutability qualifier. When calling the method, the receiver's type is checked in the same way as that of the other parameters.

□

In TIFI and TIFI+, a type inference algorithm is used to infer the local variables' qualifiers, so these variables require no manual annotation. The algorithm will be described in Chapter 5.

2.9.2 Qualifier Polymorphism

It is clear that beyond constructed objects, we also want to pass Fresh objects as method parameters. Unfortunately, considering that there is an arbitrarily high – and not statically predictable – number of such $Fresh(n)$ qualifiers, they cannot be directly used in method signatures. In addition to that, it does not matter for their behavior or allowed operations whether a group of objects is tagged as $Fresh(n)$ or $Fresh(m)$. When allowing to pass Fresh objects as method parameters, we thus would like to allow passing any kind of Fresh object.

Qualifier Polymorphism

As a solution, [10] introduces qualifier polymorphism as a means to pass Fresh objects between methods.

The paper's qualifier polymorphism mechanism is semantically similar to Java generics : Each method has a list of *formal qualifier parameters*, which need to be statically replaced with *actual qualifier parameters* when

```
q Person <q extends Qual> getRoommate() q {
  return this.roommate;
}
void <q extends Writeable> setRoommate(
    q Person other) q {
  this.roommate = other;
}
```

Listing 6: Getters and Setters for the `Person` classes' `roommate` field.

invoking it. Unlike usual fresh qualifiers, it is not allowed to commit fresh qualifiers passed in as actual qualifier parameters.

To restrict the set of qualifiers allowed to be passed as actual qualifier parameters, [10] introduces the upper qualifier bounds Qual and Writeable (Figure 8). It is important to note that these bounds are not part of the actual qualifier hierarchy, but just used to restrict the allowed parameters in method signatures.

Example 9 (Getter and Setter)**.** As an example of how qualifier polymorphism may be used in practice, let's have a look at the implementation of the `Person` classes' `setRoommate()` and `getRoommate()` getter and setter methods (Listing 6).

These getter and setter methods can now be called while constructing the `ernie` and `bert` objects, even without prematurely committing these objects' Fresh qualifiers, by giving this Fresh qualifier as actual parameter: `ernie.setRoommate<Fresh(eb)>(bert);`

□

It is important to note that a method invocation's actual qualifier parameters are always inferred in practice, so the programmer never needs to write out a specific Fresh qualifier in his code[1].

Example 10. Note that qualifier polymorphism also enables us to write factory methods returning Fresh objects (Listing 7).

In practice, through automatic inference of actual qualifier parameters, calling such a method is simple. Nevertheless, it is interesting to note that the fresh qualifier of the object returned by `makeSamson()` was actually introduced by the calling method. It gets obvious by explicitly writing out the `commit` and `newtoken` statements (Listing 8).

□

[1]The reason for this design decision is that Java provides no syntax for qualifier polymorphism. Always inferring these parameters allows to provide qualifier polymorphism without needing to extend the syntax.

```
q Person <q extends Qual> makeSamson() {
  Person samson = new Person();
  // samson: Fresh(samson)
  commit Fresh(samson) to q.  // (implicit)
  // samson: q
  return samson;
}
```

Listing 7: A factory method potentially returning a fresh object

```
newtoken n;
x = <Fresh(n)>makeSamson();
// x: Fresh(n)
```

Listing 8: Calling a factory (Commits made explicit)

2.9.3 Method-Level Constraints for Handling Fresh Qualifiers

Until now, we have looked at how to pass objects of different immutability types to methods. However, Fresh qualifiers can be committed, so when doing modular type checking, we need to be aware of all changes to the set of "active" Fresh qualifiers which happen by calling a method. This information thus also needs to be encoded in method signatures.

Methods never modify caller-visible Fresh qualifiers

[10] resolves this issue by unifying all methods' commit behavior: We thus impose the constraint that the set of active Fresh qualifiers before a method invocation is the same as after a method invocation.

With this rule, there is no need for an additional method signature annotation.

Figure 9 depicts the control flow (arrows) throughout the execution of four method instances (white boxes). Each of the red bars can be assigned a set of Fresh qualifiers which needs to equal the set of active Fresh qualifiers every time the control flow crosses the bar.

In this graphic, it is clear to see that each time a new Fresh qualifier is created by a method instance, the exact same method instance is also responsible for ensuring is does not count into the set of active Fresh qualifiers when the method instance returns. This can either be reached by committing the Fresh qualifier or by dropping all references to object of that Fresh group. Especially, it is not possible for a method instance to delegate the job of committing to other methods, as these other methods in turn also need to fulfill the same constraint of not modifying the set of Fresh qualifiers. If we explicitly use `newtoken` and `commit` statements, a `commit` statement thus always occurs in the same method body as the corresponding `newtoken` statement.

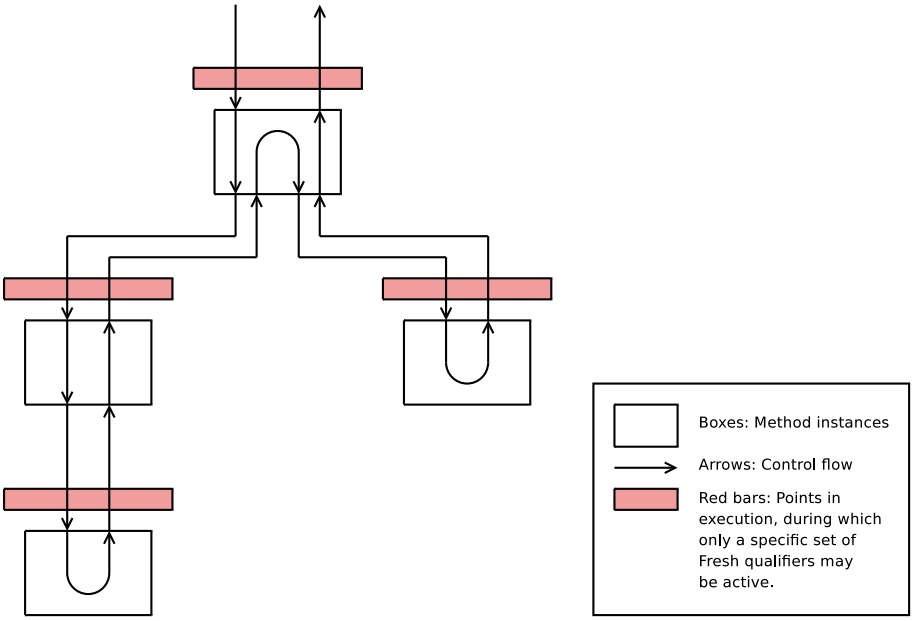

Figure 9: Typical control flow

3 Motivational Examples

This chapter gives a number of motivational examples for some of TIFI+'s possible usage scenarios.

All of the presented examples can be run using the TIFI+ type checker implementation and are included in the tests.

Unfortunately, the TIFI+ implementation currently lacks support for Java 1.5's type parameters ([9, Section 8.1.2]). In some cases, the examples thus use specially crafted non-generic classes where the use of Java's generics mechanism would have been more appropriate.

Note the different notation between the actual implementation and the notation used in the TIFI+ introduction and formalization: In the implementation, immutability qualifiers are type annotations and thus start with the "At" sign (`@`).

To syntactically support qualifier polymorphism, the implementation uses the annotations `@PolyQual`, `@PolyQual1`, `@PolyQual2`, `@PolyWriteable`, `@PolyWriteable1`, `@PolyWriteable2`, `@PolyAny`, `@PolyAny1` and `@PolyAny2`. Conceptually, each of these different annotations occurring in a method signature corresponds to one formal qualifier parameter, with the upper bound referred to in the annotation name.

For example, a method with the signature

```
void m(@PolyWriteable C a, @PolyWriteable1 C b,
       @PolyWriteable C c, @PolyAny1 C d);
```

corresponds to

```
void <q1 extends Writeable,
      q2 extends Writeable,
      q3 extends Any>
     m(q1 C a, q2 C b, q1 C c, q3 C d);
```

3.1 Flexible Initialization

This example (Listing 9) includes the features discussed in the previous section.

The created objects `ernie` and `bert` are both created to be immutable. During the execution of the `makeErnieAndBert()` method, these objects first have individual different Fresh types, which are then merged when `setRoommate()` is called and finally committed to `@Rd` when they are returned.

It is possible to create both mutable and immutable instances of the `Person` class. While `ernie` and `bert` will always be roommates, there may be other persons who can change their abstract state in that respect.

```
1  import checkers.immutability.quals.*;
2  import static checkers.immutability.api.API.*;
3
4  class ErnieAndBert {
5    static class Person {
6      @MyAccess Person roommate;
7      @PolyQual Person getRoommate() @PolyQual {
8        return roommate;
9      }
10     void setRoommate(@PolyWriteable Person pal) @PolyWriteable {
11       this.roommate = pal;
12     }
13   }
14
15   // Uncomment magic() to let the type inference dump the
16   // type environment to stdout.
17   @Rd Person makeErnieAndBert() {
18     Person ernie = new Person();
19     Person bert = new Person();
20     // magic();
21     ernie.setRoommate(bert);
22     bert.setRoommate(ernie);
23     // magic();
24     return ernie;
25   }
26 }
```

Listing 9: Creating circular immutable object graphs.

3.2 Modeling Abstract State

There is often a mismatch between an object's abstract, observable state and the value of its instance variables. For example, an object may use an instance variable to cache the results of a calculation, without changing its abstract state when assigning it.

In this example (Listing 10), the `CachingCalculator` class explicitly excludes the cache from its abstract state. At the same time, it also specifies using `@MyAccess` that the `Calculator` instance referenced by it has the same immutability type. In this way, the referenced object becomes part of the outer object's abstract state: If the inner object is mutable (`@RdWr`), the outer object is mutable, too. If the inner object is immutable (`@Rd`), the outer object is immutable, too.

Note that it is up to the programmer to make sure that the abstract state, which can be enforced by the type system, corresponds to the observable state. For example, when annotating all of a classes' instance variables as `@RdWr` instead of `@MyAccess`, it is easily possible to construct a class whose instances have mutable observe able state independent of their type in the TIFI+ type system.

3.3 Accumulating Results

Flexible initialization allows to keep objects in their initialization phase for longer than just the execution of their constructor. This example shows how a partially initialized (Fresh) object can be passed between methods to accumulate a result, where each method adds a part of the result to it. Finally, after returning from these methods, the object finishes its initialization phase and is made immutable.

Listing 11 is an implementation of a simple binary tree data structure, in which each leaf node contains an instance of the `Item` class. In such a tree, invoking the method `findItems()` with an `ItemPredicate` argument yields an `ItemSet` containing all items in the tree for which the predicate is fulfilled.

Instead of calling `findItems()` recursively and joining the resulting sets when returning from each branch node, this algorithm uses an initially empty `ItemSet` instance as accumulator object. The method `actualFindItems()`, implemented in the leaf and the branch class, takes a predicate and the accumulator set as parameters. The branch class passes them both down to the children, the leaf class adds its contained item to the set, if it fulfills the predicate.

When `actualFindItems()` is called to execute this algorithm, the set is empty and the variable referencing is statically of a Fresh type. When the algorithm runs, the Fresh object is passed down using qualifier polymorphism. When every tree node has been visited, the set is committed to `@Rd`

```
1  import checkers.immutability.quals.*;
2  import static checkers.immutability.api.API.*;
3
4  abstract class CachingExample {
5    interface Input {}
6    interface Output {}
7    interface Parameters {}
8
9    /**
10    * Calculates an output from an input, using the
11    * current calculation parameters.
12    */
13   interface Calculator {
14     @Rd Output calculate(@Rd Input input) @PolyQual;
15     void setParameters(@Rd Parameters params) @PolyWriteable;
16   }
17
18   interface ResultMap {
19     void put(@Rd Input key, @Rd Output value) @PolyWriteable;
20     @Rd Output get(@Rd Input key) @PolyQual;
21     void clear() @PolyWriteable;
22   }
23
24   /** A calculator which caches its results. */
25   final class CachingCalculator implements Calculator {
26     @MyAccess Calculator actualCalculator;
27     @RdWr ResultMap cachedResults;
28
29     // Constructor omitted.
30
31     /** Caches calculation results. */
32     @Override
33     public @Rd Output calculate(@Rd Input input) @PolyQual {
34       Output result = cachedResults.get(input);
35       if (result == null) {
36         result = actualCalculator.calculate(input);
37         cachedResults.put(input, result);
38       }
39       return result;
40     }
41
42     /** Sets parameters and clears cache. */
43     @Override
44     public void setParameters(
45         @Rd Parameters params) @PolyWriteable {
46       actualCalculator.setParameters(params);
47       cachedResults.clear();
48     }
49   }
50 }
```

Listing 10: An implementation of result-caching.

and is then immutable.

It is interesting to note that this example is motivated by a similar method called `inferLoopInner()`[2] in the TIFI+ implementation, where a set of inference results is accumulated.

[2]This method is part of the `ImmutabilityInferenceVisitor` class.

```
1  import checkers.immutability.quals.*;
2
3  abstract class TreeTraversal {
4    interface Item {}
5    interface ItemSet {
6      void addItemInPlace(@Rd Item item) @PolyWriteable;
7    }
8    interface ItemPredicate {
9      @Rd boolean isFulfilled(@Any Item it) @Rd;
10   }
11   /** A tree is either a leaf containing an item
12       or a branch containing multiple trees. */
13   abstract static class ItemTree {
14     abstract @PolyWriteable ItemSet makeEmptyItemSet() @Any;
15
16     @Rd ItemSet findItems(@Rd ItemPredicate pred) @Any {
17       ItemSet unfinishedSet = makeEmptyItemSet();
18       actualFindItems(pred, unfinishedSet);
19       return unfinishedSet;
20     }
21
22     abstract void actualFindItems(
23         @Rd ItemPredicate pred,
24         @PolyWriteable ItemSet unfinishedSet) @Any;
25   }
26
27   abstract static class ItemTreeLeaf extends ItemTree {
28     @Rd Item item;
29
30     @Override void actualFindItems(
31         @Rd ItemPredicate pred,
32         @PolyWriteable ItemSet unfinishedSet) @Any {
33       if (pred.isFulfilled(this.item)) {
34         unfinishedSet.addItemInPlace(this.item);
35       }
36     }
37   }
38
39   abstract static class ItemTreeBranch extends ItemTree {
40     @Rd ItemTree leftTree;
41     @Rd ItemTree rightTree;
42
43     @Override void actualFindItems(
44         @Rd ItemPredicate pred,
45         @PolyWriteable ItemSet unfinishedSet) @Any {
46       this.leftTree.actualFindItems(pred, unfinishedSet);
47       this.rightTree.actualFindItems(pred, unfinishedSet);
48     }
49   }
50 }
```

Listing 11: Traversing a tree

4 Formal Model

This section presents an operational semantics which formalizes the TIFI+ programming model.

The formalization is done in preparation for the next chapter, where it will be useful for explaining the TIFI+ inference algorithm and reasoning about it.

In order to facilitate this reasoning, the language has only a limited set of features. In particular, only the access qualifiers for immutability are statically typed.

4.1 Mathematical Notation

The mathematical notation we use here is similar as in [10].

$X \to Y$ denotes the set of functions from X to Y, $X \rightharpoonup Y$ is the set of partial functions. $x \mapsto y$ $(x \in X, y \in Y)$ is defined as a partial function $f \in X \rightharpoonup Y$, where $f(x) = y$ and $f(x')$ is undefined for any other $x' \in X \setminus \{x\}$. A partial function f's domain is the set $\mathrm{dom}(f) \subseteq X$, for which it is defined. Its range is defined as $\mathrm{rng}(f) := \{f(x) \mid x \in \mathrm{dom}(f)\}$. We can combine two partial functions $f, g \in X \rightharpoonup Y$ using the combination operator $f[g]$. The resulting function $f' := f[g] \in X \rightharpoonup Y$ is defined as

$$f'(x) := \begin{cases} g(x) & \text{iff } x \in \mathrm{dom}(g) \\ f(x) & \text{iff } x \notin \mathrm{dom}(g) \text{ and } x \in \mathrm{dom}(f) \\ \mathit{undef} & \text{otherwise} \end{cases}$$

We write $f, x \mapsto y$ for $f[x \mapsto y]$ to indicate that $x \notin \mathrm{dom}(f)$.

Partial functions can be seen as sets of tuples containing key-value pairs. Therefore, $\emptyset$ also denotes a partial function with an empty domain. For example, $\emptyset, 1 \mapsto 2, 2 \mapsto 3$ is a partial function in $\mathbb{N} \rightharpoonup \mathbb{N}$ mapping 1 and 2 to their successors. We also sometimes write the tuples (x, y) as $x \mapsto y$, so that $\{(1, 2), (2, 3)\}$ is written as $\{1 \mapsto 2, 2 \mapsto 3\}$.

On a partial function $f \in X \rightharpoonup Y$, the replacement operator $f[y_1/y_2]$ for $y_1, y_2 \in X$ is defined as

$$f[y_1/y_2] := \begin{cases} (x, y_2) & \mathit{if}\ (x, y_1) \in f \\ (x, y) & \mathit{if}\ (x, y) \in f, y \neq y_1 \end{cases}$$

Replacement of elements $x_1, x_2 \in X$ is defined likewise.

The restriction of a function $f \in X \to Y$ to a subset $X' \subseteq X$ is written as $f|X' := f \cap (X' \times Y)$.

Lists are written in the usual functional style, using a construction operator ::, where *nil* denotes the empty list. We abbreviate lists of elements $x \in X$ using an overbar notation, $\bar{x}$. The function $\mathit{ListOf}(X)$ denotes the

set of all lists $x_0 :: x_1 :: \cdots :: nil$ containing items $x_i \in X$ ($i \in \mathbb{N}$). The notation $x \in \bar{x}$ denotes that the element x is part of the list $\bar{x}$.

Similar to the list notation, we will use $x^\star$ to denote subsets of X ($x \in X$). $\mathrm{Pot}(X)$ denotes a set's power set.

Relations over sets X are modeled as set $R \subseteq X \times X$. For $x_1, x_2 \in X$, $x_1 \, R \, x_2$ is an abbreviation for $(x_1, x_2) \in R$.

In addition to partial functions, replacement is also defined on lists, tuples and triples, where it is applied recursively and componentwise (Example 11).

Example 11 (Recursive, componentwise replacement)**.** These examples demonstrate how replacement is defined on compound data structures like tuples and lists.

$$(x_1 :: x_2 :: x_3 :: nil)[x_2/x_4] = x_1 :: x_4 :: x_3 :: nil$$
$$(x_1, x_2, x_3)[x_2/x_4] = (x_1, x_4, x_3)$$
$$(x_1, x_2, (x_1 :: x_2 :: x_3 :: nil))[x_2/x_4] = (x_1, x_4, (x_1 :: x_4 :: x_3 :: nil))$$

□

4.2 Basic Data Structures

Class, field, method, variable and object identifiers are the basic data structures underlying these operational semantics. The variable names *cid*, *fid*, *mid*, *x*, *oid* and many others defined below are conventions used in the formalism. Usages of these names need to be understood with an implicit preceding "Let $cid \in ClassID$" or similar.

$$cid \in ClassID \qquad fid \in FieldID$$
$$mid \in MethodID \qquad x \in VarID$$
$$oid \in ObjID$$

Furthermore, $null \in VarID$ denotes a special variable identifier, whereas $x, x_1, x_2, \ldots$ denote arbitrary variable identifiers in $VarID$. We also introduce a special value $nullID$ and a special type $NullT$, which is used as the type of variables whose value is $nullID$. $NullT$ is a subtype of all other static types, so that `null` can be used wherever an object is expected. Just like in Java, $NullT$'s existence is mostly hidden from the programmer – it cannot be used in method signatures or variable declarations.

The variable identifiers $arg_0, arg_1, \ldots \in VarID$ are used for method arguments. $this \in VarID$ is used for the implicit receiver argument.

Dynamic types / static types

The operational semantics track two kinds of immutability type information: First, there is a dynamically known type for each object, stored on

the heap, called the *dynamic type*, and often denoted q_d or p_d. Second, we also dynamically keep track of each method's local variables' types. The type tracking we do there is as close as possible to the type checker / type inference algorithm which we will describe later. Therefore, these types are called *static types* (q_s, p_s).

For representing type information, we introduce the access qualifier symbols *Rd*, *RdWr*, *Any*, as well as F_d and F_s for Fresh qualifiers. As with q_d and q_s, F_d denotes a dynamic Fresh type, whereas F_s denotes a static Fresh type. As programs can at runtime create an arbitrarily large number of different Fresh qualifiers, we parameterize F_d over the set of object IDs belonging to that (dynamic) Fresh type. Likewise, F_s is parameterized over a set of local variables pointing to objects having the same Fresh type (see Example 12).

$$
\begin{aligned}
\mathit{ObjFreshTypes} &:= \{F_d(S) \mid S \subseteq \mathit{ObjID}\} \\
\mathit{VarFreshTypes} &:= \{F_s(S) \mid S \subseteq \mathit{VarID}\} \\
\mathit{VarWriteableTypes} &:= \mathit{VarFreshTypes} \cup \{\mathit{RdWr}\} \\
p_d, q_d \in \mathit{ObjImmType} &:= \{\mathit{Rd}, \mathit{RdWr}\} \cup \mathit{ObjFreshTypes} \\
p_s, q_s \in \mathit{VarImmType} &:= \{\mathit{Rd}, \mathit{RdWr}, \mathit{Any}\} \cup \mathit{VarFreshTypes} \cup \{\mathit{NullT}\} \\
\mathit{fanno} \in \mathit{FieldAnno} &:= \{\mathit{Rd}, \mathit{RdWr}, \mathit{Any}, \mathit{MyAccess}\}
\end{aligned}
$$

Fresh clouds

It is crucial to note that this definition of *ObjFreshTypes* and *VarFreshTypes* deviates from the idea of giving out tokens described in the last chapter and used in the original TIFI paper [10]. Instead of identifying a Fresh type over its token, we now identify it over the objects (or variables) which belong to that type, called the *Fresh group* or *Fresh cloud*.

The set *VarFreshTypes* is used for tracking the types directly reachable over a currently executing method's local variables. This tracking is built to be very similar to the tracking the type checker will later do statically. The correspondence between *ObjFreshTypes* and *VarFreshTypes* is explained in Example 14.

The subtype relations $<:_d$ and $<:_d$ over *ObjImmType* and *VarImmType* are the sets

$$
\begin{aligned}
<:_d &:= \{(\mathit{Rd}, \mathit{Any}), (\mathit{RdWr}, \mathit{Any})\} \cup \{(q_d, q_d) \mid q_d \in \mathit{ObjImmType}\} \\
<:_s &:= \{(\mathit{Rd}, \mathit{Any}), (\mathit{RdWr}, \mathit{Any})\} \cup \{(q_s, q_s) \mid q_s \in \mathit{VarImmType}\}
\end{aligned}
$$

We will usually write $<:$ instead of $<:_s$ and $<:_d$.

Example 12 (Fresh types)**.** $F_s(\{x_1, x_2, x_3\})$ is a *static* Fresh type ranging over the *local variables* x_1, x_2 and x_3. If the variables x_1, x_2 and x_3 are typed with this static Fresh type, their referenced objects are all in their

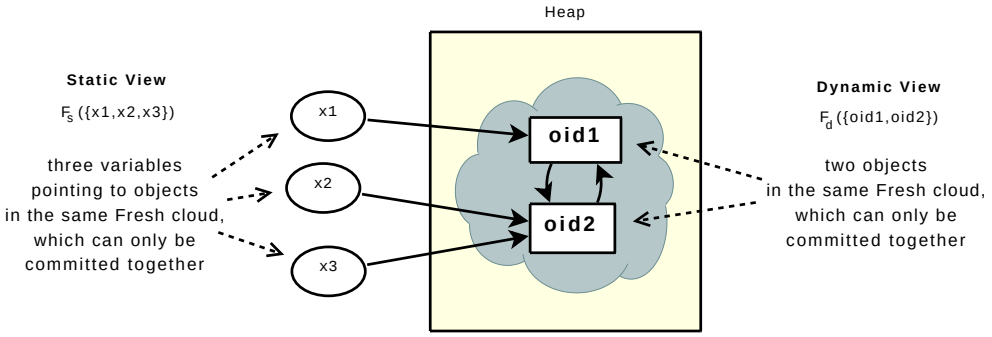

Figure 10: Correspondence between static and dynamic types

initialization phase (Fresh), and they can only be committed together – their Fresh types are already merged.

$F_d(\{oid_1, oid_2\})$ is a *dynamic* Fresh type ranging over the *objects* oid_1 and oid_2. Dynamic Fresh types only make sense in the context of an actual program execution and its heap configuration. If the objects oid_1 and oid_2 are "tagged" with this dynamic Fresh type on the heap, they can only be commited together – their Fresh types are already merged.

A static Fresh type like $F_s(\{x_1, x_2, x_3\})$ and a dynamic Fresh type like $F_d(\{oid_1, oid_2\})$ can correspond to each other by describing the same Fresh cloud. For example, if the variable x_1 references the object oid_1 and both variables x_2 and x_3 reference the object oid_2, the Fresh type $F_s(\{x_1, x_2, x_3\})$ can be considered a static representation of the dynamic Fresh type $F_d(\{oid_1, oid_2\})$ (Figure 10).

□

4.3 Method Signatures

In order to support qualifier polymorphism, we distinguish method signatures with formal qualifier parameters and those without.

$$
\begin{aligned}
\alpha \in \mathit{FormalTypeParams} &:= \{\alpha_i \mid i \in \mathbb{N}\} \\
\mathit{mst} \in \mathit{MethodSigTypes} &:= (\mathit{VarImmType} \cup \mathit{FormalTypeParams}) \setminus \{\mathit{NullT}\} \\
\mathit{ms} \in \mathit{MethodSig} &::= (\overline{\mathit{mst}_{args}}, \mathit{mst}_{recv}) \\
B \in \mathit{Bounds} &:= \{\mathit{Qual}, \mathit{Any}, \mathit{Writeable}\} \\
\mathit{qpms} \in \mathit{QPMethodSig} &:= \{(\underbrace{\overline{(\alpha, B)}}_{\text{List of formal qualifier parameters and their corresponding bounds}}, \mathit{ms}) \mid \alpha \text{ in } \mathit{ms} \text{ implies } \alpha \text{ in } \overline{(\alpha, B)}\}
\end{aligned}
$$

FormalTypeParams is the set of formal qualifier parameters used for qualifier polymorphism. In an actual qualifier polymorphic method signature, each of these occurring formal qualifier parameters α is restricted by an upper bound, which can be one of *Qual*, *Any* or *Writeable*.

Qualifier polymorphic method signatures are instantiated by replacing their formal qualifier parameters α_i with actual types $q_{si} \in \mathit{VarImmType}$, according to the restrictions imposed by the bounds. In TIFI+, there is no need for explicitly annotating these replacements: The instantiation of qualifier polymorphic methods is done automatically by the (RESOLVE) rule (Section 4.10.2). This is partly because Java has no syntax for instantiating qualifier polymorphism, partly because it would be hard to write out otherwise, especially when Fresh objects are passed.

4.4 Abstract Syntax

The language's abstract syntax is given in Figure 11. Program definitions consist of classes, class definitions of a class ID, annotated fields and methods. Method definitions are composed of a method ID, a method signature and a list of statements. Finally, we define a number of statements allowed in the language.

For method signatures, the definition uses the helper set *MethodSigDecl*, which is a subset of *QPMethodSig* containing all method signatures which may be used in a method declaration. We cannot use *QPMethodSig* directly here, as TIFI+ allows to pass Fresh objects as parameters only using qualifier polymorphism, so Fresh types cannot be directly used in method declarations.

$$\mathit{msdecl} \in \mathit{MethodSigDecl} := \{(\ldots, \mathit{ms}) \in \mathit{QPMethodSig} \mid \mathit{ms} \text{ contains no Fresh types}\}$$

Note that there is no explicit way to define constructors. We can emulate the behavior of constructors by creating objects using the `new` statement (which initializes all fields to null and returns a fresh object), then calling a qualifier polymorphic method to initialize the object's contents.

Furthermore, methods have no return value, which simplifies the formalization. Later, in Section 6.3, we will show how methods with return values can still be supported. As our formalization language focuses on immutability types, local variable declarations are written as `var` x instead of *cid* x. Each field identifier *fid* $\in$ *FieldID* may occur in at most one class definition.

In examples, we will often use a more Java-like syntax for programs.

$$
\begin{aligned}
\mathit{pdef} \in \mathit{ProgramDef} &::= \overline{\mathit{cdef}} \\
\mathit{cdef} \in \mathit{ClassDef} &::= \texttt{class}\ \mathit{cid}\ (\overline{(\mathit{fanno}, \mathit{fid})}, \overline{\mathit{mdef}}) \\
\mathit{mdef} \in \mathit{MethodDef} &::= \mathit{mid}\ \mathit{msdecl}\ \overline{s}
\end{aligned}
$$

$$
\begin{aligned}
s \in \mathit{Statement} ::= {} & x_1 = x_2 && \text{(Local Assignment)} \\
\mid {} & x_1 = x_2.\mathit{fid} && \text{(Field Get)} \\
\mid {} & x_1.\mathit{fid} = x_2 && \text{(Field Assignment)} \\
\mid {} & \texttt{if}\ x_1\ \overline{s_1}\ \overline{s_2} && \text{(If-Then-Else)} \\
\mid {} & \texttt{var}\ x && \text{(Var Decl)} \\
\mid {} & x = \texttt{new}\ \mathit{cid}() && \text{(New)} \\
\mid {} & x_r.\mathit{mid}(\bar{x}) && \text{(Call)}
\end{aligned}
$$

Figure 11: Abstract syntax

4.5 Runtime Configuration

We devise the following sets and functions to model the program state.

$$
\begin{aligned}
\mathit{Value} &:= \mathit{ObjID} \cup \{\mathit{nullID}\} \\
o \in \mathit{Object} &:= \mathit{ClassID} \times \mathit{ObjImmType} \times (\mathit{FieldID} \rightharpoonup \mathit{Value}) \\
h \in \mathit{Heap} &:= \mathit{ObjID} \rightharpoonup \mathit{Object} \\
l \in \mathit{LVarValues} &:= \mathit{VarID} \rightharpoonup \mathit{Value} \\
\Phi \in \mathit{LVarTypes} &:= \mathit{VarID} \rightharpoonup \mathit{VarImmType}
\end{aligned}
$$

l, Φ and h represent a runtime configuration: l and Φ describe the current stack frame by mapping local variables to values and (static) immutability types. h models the heap by mapping object IDs to objects.

Note that Φ contains quasi-static type information, which is tracked by the operational semantics. After describing the inference algorithm in Section 5, we will later be able to relate Φ to the type information inferred by the inference algorithm in a purely static way.

Example 13. For $x \in \mathit{VarID}$, $h(l(x))$ is the object referenced by the local variable x. If $l(x) = \mathit{nullID}$, $h(l(x))$ is undefined.

□

For an object $o := (\mathit{cid}, q_d, \mathit{fields}) \in \mathit{Object}$, where $\mathit{fields} \in \mathit{FieldID} \rightarrow \mathit{Value}$, $\mathit{fid} \in \mathit{FieldID}$, $v \in \mathit{Value}$, we devise the helper functions

$$
\begin{aligned}
I(o) &:= q_d \\
class(o) &:= cid \\
o[\mathit{fid} \mapsto v] &:= (cid, q_d, \mathit{fields}[\mathit{fid} \mapsto v]) \\
o.\mathit{fid} &:= \mathit{fields}(fid)
\end{aligned}
$$

Example 14 (Correspondence between $I(h(l(\ldots)))$ and $\Phi(\ldots)$)**.** For every heap h and mapping of local variables l, a corresponding $\Phi_\$ \in \mathit{LVarTypes}$ with $\mathrm{dom}(\Phi_\$) = \mathrm{dom}(l)$ can be determined using the following definition. For conciseness, we write $\mathbb{H}(x)$ for $I(h(l(x)))$.

$$
\Phi_\$(x) := \begin{cases} \mathit{NullT} & \text{if } l(x) = \mathit{nullID} \\ F_s(\{x' \in \mathrm{dom}(l) \mid \mathbb{H}(x') = q_d\}) & \text{if } q_d := \mathbb{H}(x) \in \mathit{ObjFreshTypes} \\ q_d & \text{if } q_d := \mathbb{H}(x) \in \{\mathit{Rd}, \mathit{RdWr}\} \end{cases}
$$

Note that $\Phi_\$$ is not necessarily the same as the type environment Φ calculated in the operational semantics below. In fact, Φ is usually less precise than $\Phi_\$$. For example, it is often the case that there is a $x \in \mathrm{dom}(\Phi)$ so that $\Phi(x) = \mathit{Any}$ whereas $\Phi_\$(x) = \mathit{Rd}$.

□

4.6 Object Model Helper Functions

All of these helper functions implicitly use the program's abstract syntax tree *pdef*, which can be seen as an implicit parameter. It is simple to extend the operational semantics to pass *pdef* along with each function or rule application, but it improves readability if we don't.

Helper functions are partial

Some of these helper functions are partial functions. When used in an undefined way from within the operational semantics, we assume the operation gets stuck.

methodBody, methodSig

The functions $\mathit{methodSig}(\mathit{mid}, \mathit{cid})$ and $\mathit{methodBody}(\mathit{mid}, \mathit{cid})$ look up a method's body $\bar{s} \in \mathit{ListOf}(\mathit{Statement})$ and (qualifier polymorphic) signature declaration $\mathit{msdecl} \in \mathit{MethodSigDecl}$.

hasField

The function $\mathit{hasField}(\mathit{cid}, \mathit{fid})$ returns *true* or *false*, indicating whether instances of the class *cid* have the field *fid*.

fieldAnno

$\mathit{fieldAnno}(\mathit{fid})$ returns the field annotation (*Rd*, *RdWr*, *Any* or *MyAccess*) for a given field ID. Note that there is only one annotation per field ID. Thus, like in Java, using a field identifier in a statement always refers to a field in a statically unambiguously defined class.

resolveFieldType

resolveFieldType resolves the static type of field variables. Because of the MyAccess constraint, a field's static type can depend on the static type q_s of the object holding that field.

$$resolveFieldType(q_s, fid) := \begin{cases} q_s & \text{if } fieldAnno(fid) = MyAccess \\ fieldAnno(fid) & \text{if } fieldAnno(fid) \in \{Rd, RdWr, Any\} \end{cases}$$

Example 15 (Resolving field types)**.** Let $f, g \in FieldID, cid \in ClassID$. The program *pdef* contains the class definition `class cid { MyAccess f; Rd g; }`

If an instance of *cid* is tagged with the $Fresh(n)$ access qualifier, we can now find out the types of its fields f and g by using the *resolveFieldType* helper function:

$$resolveFieldType(Fresh(n), f) = Fresh(n)$$
$$resolveFieldType(Fresh(n), g) = Rd$$

For another instance of *cid*, tagged with the $RdWr$ qualifier, the results are:

$$resolveFieldType(RdWr, f) = RdWr$$
$$resolveFieldType(RdWr, g) = Rd$$

□

Core rules	Interpreting program statements
Type compatibility rules	Ensuring type compatibility
Commit rules	Committing types

Figure 12: The operational semantics' layered design

$$\frac{\begin{array}{c} h \vdash \mathit{mergeFreshes}\ p_d, q_d \Downarrow h' \\ \Phi, X \vdash \mathit{mergeFreshes}\ F_s(S_1), F_s(S_2) \Downarrow \Phi', X' \\ p_d = \mathit{dQual}(F_s(S_1), h, l) \qquad q_d = \mathit{dQual}(F_s(S_2), h, l) \end{array}}{l, \Phi, h, X \vdash \mathit{commit}\ F_s(S_1)\ \mathit{to}\ F_s(S_2) \Downarrow \Phi', h', X'}\ \textsc{Fresh-to-Fresh}$$

$$\frac{\begin{array}{c} h \vdash \mathit{commit}\ p_d\ \mathit{to}\ q_d \Downarrow h' \qquad \Phi, X \vdash \mathit{commit}\ p_s\ \mathit{to}\ q_s \Downarrow \Phi', X' \\ p_s = F_s(\{x, \ldots\}) \qquad q_s \in \{Rd, RdWr\} \\ p_d = \mathit{dQual}(p_s, h, l) \qquad q_d = \mathit{dQual}(q_s, h, l) \end{array}}{l, \Phi, h, X \vdash \mathit{commit}\ p_s\ \mathit{to}\ q_s \Downarrow \Phi', h', X'}\ \textsc{Fresh-to-Rd-or-RdWr}$$

Figure 13: Commit layer rules

4.7 Operational Semantics

Like the implementation, the operational semantics use a layered design, depicted in Figure 12. The rules in each layer are only allowed to use rules from the same layer or the layer below it.

The *core rules* are responsible for interpreting the different kinds of program statements (and sequences thereof). Where a core rule needs to enforce a subtype relation between a variable's type and an expected type, it invokes a rule in the *type compatibility layer*. In this rule, when the required type compatibility is not already given, the rule may choose to establish it by invoking a *commit layer* rule.

For example, if the statement "`x=rd(x);`" is executed in a configuration where the object referenced by x is of a Fresh type, and the method expects it to be Rd, the core rule for method calls will invoke a type compatibility rule, which will in turn decide to commit x's Fresh type to Rd.

This chapter describes the operational semantics in order of this layered design, starting at the bottom.

4.8 Commit Layer

The commit layer contains rules of the form

$$\underbrace{l, \Phi, h,}_{\text{runtime configuration}} \overbrace{X}^{\text{any data structure containing static types}} \vdash \mathit{commit}\ p_s\ \mathit{to}\ q_s \Downarrow \underbrace{\Phi', h', X'}_{\text{updated data structures}}$$

which are given in Figure 13. Helper rules are shown in Figure 14.

Templating

The rules given in this section need to be understood as templates. The

$$\frac{p_d = F_d(\ldots) \qquad q_d \in \{Rd, RdWr\} \qquad h' = h[p_d/q_d]}{h \vdash \mathit{commit}\ p_d\ \mathit{to}\ q_d \Downarrow h'}\ \textsc{D-Commit}$$

$$\frac{p_s = F_s(\ldots) \qquad q_s \in \{Rd, RdWr, Any\} \qquad \Phi' = \Phi[p_s/q_s] \qquad X' = X[p_s/q_s]}{\Phi, X \vdash \mathit{commit}\ p_s\ \mathit{to}\ q_s \Downarrow \Phi', X'}\ \textsc{S-Commit}$$

$$\frac{h' = h[F_d(S_1)/F_d(S_1 \cup S_2)][F_d(S_2)/F_d(S_1 \cup S_2)]}{h \vdash \mathit{mergeFreshes}\ F_d(S_1), F_d(S_2) \Downarrow h'}\ \textsc{Merge-Dynamic-Fresh-types}$$

$$\frac{\begin{array}{l}\Phi' = \Phi[F_s(S_1)/F_s(S_1 \cup S_2)][F_s(S_2)/F_s(S_1 \cup S_2)] \\ X' = X[F_s(S_1)/F_s(S_1 \cup S_2)][F_s(S_2)/F_s(S_1 \cup S_2)]\end{array}}{\Phi, X \vdash \mathit{mergeFreshes}\ F_s(S_1), F_s(S_2) \Downarrow \Phi', X'}\ \textsc{Merge-Static-Fresh-types}$$

Figure 14: Commit layer helpers

templates are instantiated by replacing X with a data structure containing static types, for instance a method signature or a list of static types. This will allow the user of a rule to track the committing of static types in different data structures without losing consistency with Φ. In some cases, like in Example 16, we will omit giving the X parameter when invoking a commit layer rule, which is just a syntactic convenience for passing an empty set as X and not using the resulting X'.

When instantiating these templates, replacement of static types ($X[p_s/q_s]$) must be well-defined on X. It is guaranteed that on X, the same replacements are done as on Φ.

Idea

Invoking a rule of this form for a (static) Fresh type p_s commits the type in Φ and X to q_s, yielding Φ' and X'. It also commits the appropriate types on the heap h.

Changes to the static type environment Φ and the heap h always occur in parallel: Whenever one of them is updated, the other is updated as well, resulting in Φ' and h'. This is always done in a way so that their correspondence (Example 14) is retained.

Mapping between static and dynamic types

To commit the types on the heap, the static types p_s and q_s must be mapped to their corresponding dynamic types p_d and q_d, which is done by the *dQual* helper function[3].

$$dQual(p_s, h, l) := \begin{cases} F_d(\emptyset) & if\ p_s = F_s(\emptyset) \\ I(h(l(x))) & if\ p_s = F_s(\{x, \ldots\}) \\ Rd & if\ p_s = Rd \\ RdWr & if\ p_s = RdWr \\ undef & \text{otherwise} \end{cases}$$

The function *dQual* is only defined for heaps h, which belong to the same

[3]Note that *dQual* is never invoked with the static type *Any* as argument: As *Any* has no well-defined dynamic counterpart, such cases cannot be handled using *dQual*.

runtime configuration as Φ. These heaps have the property that if $p_s = F_s(\{x, \ldots\})$, $I(h(l(x)))$ is well-defined and the same for all such x. In other words, if two variables x and y have the same static Fresh type in Φ, the objects they point to will also have the same dynamic Fresh type.

Rules

The simplest form of committing is using the (Fresh-to-Rd-or-RdWr) rule, which simply replaces all occurrences of a Fresh type with *Rd* or *RdWr*. Apart from that, the commit layer also merges Fresh types when committing, when using the rule (Fresh-to-Fresh). Committing $F_s(S_1)$ to $F_s(S_2)$ in a type environment Φ yields a type environment Φ', where both these types are replaced by $F_s(S_1 \cup S_2)$.

Helper rules

The helper rules are given in both a variant for dynamic and for static types. It is to be noted that the (S-Commit) rule also allows to commit to *Any*, which is used in one rule of the type compatibility layer. Committing to *Any* is semantically the same as committing to *Rd* or *RdWr*, then upcasting all objects whose type changed to *Any*.

Example 16 (Committing a Fresh type to Rd)**.** For the runtime configuration l, Φ, h, the commit from $F_s(\{x\})$ to *Rd* yields the updated static type environment and heap Φ', h' using the rule (Fresh to Rd or RdWr):

$$\frac{\begin{array}{c} h \vdash \mathit{commit}\ p_d\ \mathit{to}\ q_d \Downarrow h' \qquad \Phi \vdash \mathit{commit}\ F_s(\{x\})\ \mathit{to}\ \mathit{Rd} \Downarrow \Phi' \\ p_d = \mathit{dQual}(F_s(\{x\}), h, l) = I(h(l(x))) = F_d(\{\mathit{oid}_x, \mathit{oid}_y\}) \\ q_d = \mathit{dQual}(\mathit{Rd}, h, l) = \mathit{Rd} \end{array}}{l, \Phi, h \vdash \mathit{commit}\ F_s(\{x\})\ \mathit{to}\ \mathit{Rd} \Downarrow \Phi', h'}$$

$$\begin{array}{ll} l := \{x \mapsto \mathit{oid}_x\} & \\ \Phi := \{x \mapsto F_s(\{x\})\} & \Phi' := \{x \mapsto \mathit{Rd}\} \\ h := \{\mathit{oid}_x \mapsto (\mathit{cid}, F_d(\{\mathit{oid}_x, \mathit{oid}_y\}), \mathit{fields}_x), & h' := \{\mathit{oid}_x \mapsto (\mathit{cid}, \mathit{Rd}, \mathit{fields}_x), \\ \quad \mathit{oid}_y \mapsto (\mathit{cid}, F_d(\{\mathit{oid}_x, \mathit{oid}_y\}), \mathit{fields}_y)\} & \quad \mathit{oid}_y \mapsto (\mathit{cid}, \mathit{Rd}, \mathit{fields}_y)\} \end{array}$$

Note that by using *dQual*, the rule finds out which actual (dynamic) Fresh types need to be updated. Even though the dynamic Fresh type $F_d(\{\mathit{oid}_x, \mathit{oid}_y\})$ is different to $F_s(\{x\})$, these are still corresponding fresh types in Φ and h and updated appropriately.

□

4.9 Type Compatibility Layer

The general form of a type compatibility rule is

$$l, \Phi, h, X \vdash x!q_s \Downarrow \Phi', h', X'$$

$$\frac{true}{l, \Phi, h, X \vdash nil!nil \Downarrow \Phi, h, X}\ \textsc{Type Compat List Empty}$$

$$\frac{\begin{array}{c} l, \Phi, h, (\bar{q_s}, X) \vdash x!q_s \Downarrow \Phi', h', (\bar{q_s}', X') \\ l, \Phi', h', X' \vdash \bar{x}!\bar{q_s}' \Downarrow \Phi'', h'', X'' \end{array}}{l, \Phi, h, X \vdash x :: \bar{x}!q_s :: \bar{q_s} \Downarrow \Phi'', h'', X''}\ \textsc{Type Compat Cons List}$$

Figure 15: Type compatibility layer – List rules

for "weak" enforcing of type compatibility or

$$l, \Phi, h, X \vdash x!!q_s \Downarrow \Phi', h', X'$$

for "strong" enforcing of type compatibility.

Invoking such a rule with a runtime configuration l, Φ, h and a data structure X commits the type of x or q_s, so that x's type is a subtype of q_s. Mathematically spoken, this means $\forall y \in \text{dom}(\Phi)$ where $\Phi(y) = q_s$ follows that $\Phi'(x) <: \Phi'(y)$. The same holds for dynamic types on the heap h.

Like the commit layer rules (Section 4.8), the rules given here are templates, which are instantiated by replacing X with a data structure containing static types. On X, the same replacements take place as on Φ.

The difference between the weak and the strong form is that in order to establish type compatibility, they perform different commits. While the strong rules can do a superset of the commits that the weak rules do, the weak rules are more conservative in leaving the type q_s untouched, which is also needed in some situations.

4.9.1 Weak Type Compatibility Rules

List Rules

The (weak) list rules for type compatibility (Figure 15) allow to establish multiple type compatibilities at once, which is used when the type compatibility of a method invocation's actual arguments is checked.

Main Rules

The type compatibility layer's main rules are given in Figure 16. Note that when requiring a variable x's type to be a subtype of *Any*, the result can be ambiguous: We can either commit to *Rd* (Below Any (1)) or to *RdWr* (Below Any (2)).

4.9.2 Strong Type Compatibility Rules

The type compatibility layer also provides strong rules for type compatibility, of the general form

$$l, \Phi, h, X \vdash x!!q_s \Downarrow \Phi', h', X'$$

$$\frac{\Phi(x) = q_s}{l, \Phi, h, X \vdash x!q_s \Downarrow \Phi, h, X} \textsc{Simple 1} \qquad \frac{q_s = Any \qquad \Phi(x) \in \{Rd, RdWr\}}{l, \Phi, h, X \vdash x!q_s \Downarrow \Phi, h, X} \textsc{Simple 2}$$

$$\frac{\Phi(x) = NullT}{l, \Phi, h, X \vdash x!q_s \Downarrow \Phi, h, X} \textsc{Simple 3}$$

$$\frac{\Phi(x) = F_s(\ldots) \qquad q_s \in \{Rd, RdWr\} \qquad l, \Phi, h, X \vdash commit\ \Phi(x)\ to\ q_s \Downarrow \Phi', h', X'}{l, \Phi, h, X \vdash x!q_s \Downarrow \Phi', h', X'} \textsc{Fresh below Rd or RdWr}$$

$$\frac{l, \Phi, h, X \vdash x!Rd \Downarrow \Phi', h', X'}{l, \Phi, h, X \vdash x!Any \Downarrow \Phi', h', X'} \textsc{Below Any (1)} \qquad \frac{l, \Phi, h, X \vdash x!RdWr \Downarrow \Phi', h', X'}{l, \Phi, h, X \vdash x!Any \Downarrow \Phi', h', X'} \textsc{Below Any (2)}$$

$$\frac{\Phi(x) = F_s(\ldots) \qquad q_s = F_s(\ldots) \qquad l, \Phi, h, X \vdash commit\ \Phi(x)\ to\ q_s \Downarrow \Phi', h', X'}{l, \Phi, h, X \vdash x!q_s \Downarrow \Phi', h', X'} \textsc{Fresh below Fresh}$$

Figure 16: Type compatibility Layer – Weak Rules

Using $x!!q_s$, we allow even q_s to be committed (to a target type q'_s) in order to fulfil $\Phi'(x) <: q'_s$. Compare: If a weak type compatibility rule does a commit, $\Phi(x)$ always appears in the left hand side of a commit rule application (*commit* $\Phi(x)$ *to* ...). In effect, this means that only $\Phi(x)$ may be committed to *Rd*, *RdWr* or *Any* if it is a Fresh type. The type q_s, if it is a Fresh type, may only be merged with other Fresh types, but not end up being committed to *Rd*, *RdWr* or *Any*.

Why do we make a difference between these two kinds of type compatibility rules? There are two major places from where type compatibility rules are used:

- The first place is the core rule for field assignment (page 39): If $x_1.fid = x_2$ is executed, x_2's type must be compatible to the field's annotated type. If the (resolved) annotated type is a Fresh type, then *fid* is annotated with *MyAccess* and x_1 has the same Fresh type. In this case, we can allow to commit this Fresh type to ensure the type compatibility, thus we use $x_2!!\Phi(x_2)$.

- The second place is the core rule for method invocation (page 40). Here, the actual arguments' types need to be subtypes of the declared formal arguments' types, but we cannot simply commit the expected types in the formal arguments. After all, we later want to build a modular type checker, where modules are methods, so we can only assume that the called method has been checked with the parameter types given in the method signature, not with those they can be commited to[4]. Consequently, we can only use $x_{arg}!q_{s\,arg}$ here.

[4]It is actually possible that a Fresh type occurring in the signature will be merged with

$$\frac{\Phi(x) \in \{Rd, RdWr\} \quad l, \Phi, h, X \vdash \mathit{commit}\, F_s(S)\ \mathit{to}\, \Phi(x) \Downarrow \Phi', h', X'}{l, \Phi, h, X \vdash x!!F_s(S) \Downarrow \Phi', h', X'}\ \textsc{Rd or RdWr below Fresh}$$

$$\frac{\begin{array}{c}\Phi, X \vdash \mathit{commit}\, q_s\ \mathit{to}\, p_s \Downarrow \Phi', X' \\ h \vdash \mathit{commit}\, q_d\ \mathit{to}\, p_d \Downarrow h' \qquad p_s = \Phi(x) = \mathit{Any} \\ q_s = F_s(\{x_q, \ldots\}) \qquad p_d = I(h(l(x))) \qquad q_d = I(h(l(x_q)))\end{array}}{l, \Phi, h, X \vdash x!!q_s \Downarrow \Phi', h', X'}\ \textsc{Any below Fresh}$$

$$\frac{l, \Phi, h, X \vdash x!q_s \Downarrow \Phi', h', X'}{l, \Phi, h, X \vdash x!!q_s \Downarrow \Phi', h', X'}\ \textsc{Reduce}$$

Figure 17: Type compatibility layer – Strong rules

The strong rules for type compatibility are given in Figure 17. Note that if the type compatibility can be established using the weak rules ($x!q_s$), this solution will also be found using the strong rules ($x!!q_s$), via the (Reduce) rule.

Example 17. Using $x!q_s$, we can guarantee that q_s does not appear in the left hand side of a commit rule application. Nonetheless, if q_s is a Fresh type, is may still have been changed in the resulting type environment.

Let $\Phi := \{x \mapsto F_s(\{x\}), y \mapsto F_s(\{y\})\}$. Invoking the rule $\ldots, \Phi, \ldots \vdash x!F_s(\{y\}) \Downarrow \ldots$ yields a type environment Φ' where $\Phi'(x) = \Phi'(y) = F_s(\{x, y\})$. In this case, *commit* $F_s(\{x\})$ *to* $F_s(\{y\})$ has been applied.

□

4.10 Core Rule Layer

This layer defines rules to execute statements and statement sequences, of the form

$$l, \Phi, h \vdash \bar{s} \to l', \Phi', h'$$

l, Φ and h together represent the runtime configuration before executing the program statements $\bar{s}$. The configuration after executing $\bar{s}$ is represented by l', Φ' and h'.

4.10.1 Helpers for Type Environments

The set of types *contained in a type environment* Φ is denoted as $\text{types}(\Phi) := \text{rng}(\Phi) \cup \{F_s(\emptyset)\}$. Even though there can never be a variable of type $F_s(\emptyset)$, it is still useful to consider $F_s(\emptyset)$ to be contained in every Φ (see Example 19). types(Φ)

the Fresh type of an argument expression. However, this does not change the fact that it is still a Fresh type, which is distinct from other Fresh types in the signature.

For easier type environment updating, we define the helper functions $\Phi!\langle x \mapsto \ldots\rangle$, $\Phi\langle \mathit{delete}\ x\rangle$, and $\Phi\langle x \mapsto \ldots\rangle$.

Overview

$\Phi!\langle x \mapsto \ldots\rangle$ introduces a *new variable* to a type environment, $\Phi\langle x \mapsto \ldots\rangle$ updates the type of an *existing variable*. Finally, $\Phi\langle \mathit{delete}\ x\rangle$ deletes an existing variable from a type environment. Unlike the usual update operator $\Phi[x \mapsto \ldots]$, these operators also take care about updating the Fresh types contained in Φ.

Note that updating an existing variable $\Phi\langle x \mapsto \ldots\rangle$ turns out to be more complicated than adding a new one. The update operation is thus defined in terms of the other two operations $\Phi!\langle x \mapsto \ldots\rangle$ and $\Phi\langle \mathit{delete}\ x\rangle$.

$\Phi!\langle x \mapsto q_s\rangle$

$\Phi!\langle x \mapsto q_s\rangle$ introduces a new variable x into a type environment. It is thus only defined if $x \notin \mathrm{dom}(\Phi)$ and the type q_s is contained in Φ, that is, $q_s \in \mathrm{types}(\Phi)$. Note that the empty Fresh type is contained in every type environment.

The result of $\Phi!\langle x \mapsto q_s\rangle$ is a type environment $\Phi' := \Phi!\langle x \mapsto q_s\rangle$ so that

$$\Phi'(x') := \begin{cases} \Phi(x') & \text{if } x' \neq x \\ q_s & \text{if } x' = x \text{ and } q_s \in \{Rd, RdWr, Any, NullT\} \\ F_s(S \cup \{x\}) & \text{if } x' = x \text{ and } q_s = F_s(S) \end{cases}$$

Example 18 (Introducing an existing Fresh variable)**.** Let $w, x, y, z \in \mathit{VarID}$, $\Phi := \{(x, F_s(\{x, y\})), (y, F_s(\{x, y\})), (z, F_s(\{z\}))\}$.

Using our helper function, we can calculate $\Phi' := \Phi!\langle w \mapsto F(\{x, y\})\rangle$, which can be written as

$$\Phi' = \{(w, F_s(\{w, x, y\})), (x, F_s(\{w, x, y\})), \\ (y, F_s(\{w, x, y\})), (z, F_s(\{z\}))\}$$

It is important to note that the parameter q_s passed to the helper function is always a type existing in the type environment Φ the function is applied to. In the above example, $\Phi!\langle w \mapsto F(\{w, x, y\})\rangle$ is undefined.

□

Example 19 (Introducing a new variable with a new Fresh type)**.** Let Φ be a type environment that does not contain x, $\Phi(x)$ is undefined. We can build a similar type environment, where x is defined to be of a new Fresh type, using $\Phi' := \Phi!\langle x \mapsto F(\emptyset)\rangle$. We get $\Phi'(x) = F_s(\{x\})$.

Invoking this function with $F(\emptyset)$ effectively introduces a new Fresh type. When automatically inferring `newtoken` and `commit` from the original TIFI language, this is the equivalent to `newtoken`.

□

In order to delete variables from type environments, we define the function $\Phi\langle delete\ x\rangle$, whose result is a type environment $\Phi' := \Phi\langle delete\ x\rangle$ so that

$\Phi\langle delete\ x\rangle$

$$\Phi'(x') := \begin{cases} undef & if\ x = x' \\ \Phi(x) & if\ x \neq x' \text{ and } \Phi(x) \in \{Rd, RdWr, Any, NullT\} \\ F_s(S \setminus \{x\}) & if\ x \neq x' \text{ and } \Phi(x) = F_s(S) \end{cases}$$

The function $\Phi\langle delete\ x\rangle$ is only defined if $\Phi(x)$ is also defined; we can only delete a variable if it already exists in the type environment Φ.

Using the two functions above, we can devise[5] the reassignment operation for an existing variable's type $\Phi\langle x \mapsto q_s\rangle$ as

$\Phi\langle x \mapsto q_s\rangle$

$$\Phi\langle x \mapsto q_s\rangle := \begin{cases} \Phi & if\ \Phi(x) = q_s \\ (\Phi\langle delete\ x\rangle)!\langle x \mapsto q_s\rangle & otherwise \end{cases}$$

Again, the function is only defined if Φ contains the type q_s

4.10.2 Actual Rules

To interpret statement sequences, we divide them into their individual statements and interpret those in sequence, passing through the resulting runtime configurations.

$$\frac{}{l, \Phi, h \vdash nil \to l, \Phi, h}\ \textsc{Seq-Empty}$$

$$\frac{l, \Phi, h \vdash s \to l', \Phi', h' \qquad l', \Phi', h' \vdash \bar{s} \to l'', \Phi'', h''}{l, \Phi, h \vdash s :: \bar{s} \to l'', \Phi'', h''}\ \textsc{Seq-Nonempty}$$

We start by defining the rules for variable declaration and object creation.

Variable declaration

$$\frac{l' = l[x \mapsto nullID] \qquad \Phi' = \Phi!\langle x \mapsto NullT\rangle}{l, \Phi, h \vdash \texttt{var}\ x \to l', \Phi', h}\ \textsc{Var-Decl}$$

Local variable assignment, both in the (New) and the (Local-Assign) rule, is done by first checking that the assigned variable x is not *null*, then changing the l and Φ mappings to map the assigned variable to a new value and access qualifier. Note that this uses $\Phi\langle x \mapsto \ldots\rangle$, which gets stuck if $x \notin \text{dom}(\Phi)$.

Local variable assignment

[5] It is interesting to note that the same definition without the case $\Phi(x) = q_n$ would be undefined for $q_s \in VarFreshTypes$

$$\frac{\begin{array}{c} x \neq \mathit{null} \qquad \mathit{oid} \notin \mathrm{dom}(h) \qquad l' = l[x \mapsto \mathit{oid}] \\ \Phi' = \Phi\langle x \mapsto F_s(\emptyset)\rangle \qquad h' = h[\mathit{oid} \mapsto (\mathit{cid}, F_d(\{\mathit{oid}\}), \emptyset)] \end{array}}{l, \Phi, h \vdash x = \texttt{new}\ \mathit{cid}() \rightarrow l', \Phi', h'}\ \textsc{New}$$

$$\frac{\begin{array}{c} x_1 \neq \mathit{null} \\ l' = l[x_1 \mapsto l[x_2]] \qquad \Phi' = \Phi\langle x_1 \mapsto \Phi(x_2)\rangle \end{array}}{l, \Phi, h \vdash x_1 = x_2 \rightarrow l', \Phi', h}\ \textsc{Local-Assign}$$

The rule (Field-Get) shows how a field's type is looked up using the *resolveFieldType* helper function.

Reading field values

$$\frac{\begin{array}{c} x_1 \neq \mathit{null} \qquad l(x_2) \neq \mathit{nullID} \\ \mathit{hasField}(\mathit{class}(h(l(x_2))), \mathit{fid}) \qquad l' = l[x_1 \mapsto h(l(x_2)).\mathit{fid}] \\ \Phi' = \Phi\langle x_1 \mapsto \mathit{resolveFieldType}(\Phi(x_2), \mathit{fid})\rangle \end{array}}{l, \Phi, h \vdash x_1 = x_2.\mathit{fid} \rightarrow l', \Phi', h}\ \textsc{Field-Get}$$

The field assignment rule is similar to (Field-Get) in that it accesses both a local and a field variable. It differs, however, in that it modifies an object, and thus needs to check whether this is allowed according to its current access qualifier.

Field assignment

$$\frac{\begin{array}{c} \mathit{hasField}(\mathit{class}(h(l(x_1))), \mathit{fid}) \\ \Phi(x_1) \in \mathit{VarWriteableTypes} \\ l, \Phi, h \vdash x_2!!\mathit{resolveFieldType}(\Phi(x_1), \mathit{fid}) \Downarrow \Phi', h' \\ \mathit{ID}_1 = l(x_1) \qquad \mathit{ID}_2 = l(x_2) \\ \mathit{obj}_1 = h'(\mathit{ID}_1) \qquad h'' = h'[l(x_1) \mapsto \mathit{obj}_1[\mathit{fid} \mapsto \mathit{ID}_2]] \end{array}}{l, \Phi, h \vdash x_1.\mathit{fid} = x_2 \rightarrow l, \Phi', h''}\ \textsc{Field-Assign}$$

In (Field-Assign), note that it is never needed to commit a qualifier to make it writeable. If it is still Fresh, it is already writeable. If it is one of *Rd*, *RdWr* and *Any*, it is either writeable or it is too late to make it so.

In the (Call) rule, the following two definitions are used to facilitate the construction of l_{bin} and Φ_{bin}, which are part of the input configuration for the called method's body. l_{base} and Φ_{base} are the parts of l_{bin} and Φ_{bin}, which are the same for every method invocation.

Method Invocation (Call)

$$l_{base} := \emptyset, \mathit{null} \mapsto \mathit{nullID}$$
$$\Phi_{base} := \emptyset, \mathit{null} \mapsto \mathit{NullT}$$

$$\frac{\begin{array}{c} msdecl = methodSig(mid, class(h(l(x_r)))) \\ \Phi, msdecl \vdash Resolve\ \{\Phi(x) \mid x \in \bar{x}\} \Downarrow ms \\ \Phi, h, ms \vdash x_r :: \bar{x}!recvtype(ms) :: fpTypes(ms) \Downarrow \Phi', h', ms' \\ fpVarIDs = arg_1 :: \cdots :: arg_{|\bar{x}|} :: nil \\ l_{bin} = l_{base}[this \mapsto l(x_r), fpVarIDs \mapsto l(\bar{x})] \\ \Phi_{bin} = \Phi_{base}\langle this \mapsto recvtype(ms'), fpVarIDs \mapsto fpTypes(ms')\rangle \\ h_{bin} = h' \\ l_{bin}, \Phi_{bin}, h_{bin} : methodBody(mid, class(h(l(x_r)))) \rightarrow l_{bout}, \Phi_{bout}, h_{bout} \\ checkNoCommits(h_{bin}, h_{bout}) \end{array}}{l, \Phi, h \vdash x_r.mid(\bar{x}) \rightarrow l, \Phi', h_{bout}}\ \textsc{Call}$$

The individual steps of the method invocation rule are:

- Find the qualifier polymorphic method signature in the AST. (This works only if $l(x_r) \neq nullID$.) Replace formal type parameters by actual type parameters, obtaining the instantiated method signature ms. This is done using the (Resolve) rules below. Note that this step is ambiguous.
- Ensure the compatibility of the actual argument types to the required types imposed by the signature.

  Note that this makes use of the type compatibility rules' ability to replace types in an additional data structure X, in this case the method signature ms.
- Calculate the method body input configuration l_{bin}, Φ_{bin}, h_{bin}. The variables l_{bin} and Φ_{bin} are set to map the method arguments ($this$, $arg_1, \ldots, arg_{|\bar{x}|}$) to values and types. The heap h_{bin} stays the same as before (h').
- Interpret the method body, obtaining l_{bout}, Φ_{bout}, h_{bout}.
- Check that the called method did not commit any of the passed objects' Fresh qualifiers. $checkNoCommits(h_{bin}, h_{bout})$ ensures that the directly reachable objects after the method call still have the same tagged qualifiers as before. It is defined as $true$ iff for all $oid \in \mathrm{dom}(h_{bin})$ either $h_{bin}(oid) = h_{bout}(oid)$ or $h_{bin}(oid) = F_d(S_1), h_{bout}(oid) = F_d(S_2)$ so that $S_1 \subseteq S_2$ and $(S_2 \setminus S_1) \cap \mathrm{dom}(h_{bin}) = \emptyset$. This formula ensures that
  - No Fresh type was committed to a non-Fresh type.
  - If a Fresh type "contains" more objects in h' than in h, the additional objects did not exist in h. (Consequently, their Fresh type did not exist in h, which means that a called method cannot merge two Fresh types which existed before calling, but just Fresh types it created itself.)

We now know that the static Fresh types visible in Φ' also do not need to be merged with or committed to any other type in $\mathrm{rng}(\Phi')$. Therefore, Φ' is still a corresponding type environment to h_{bout}.

- The program configuration after the method call consists of l, Φ', h_{bout}. The local variable mappings stay the same, as these local variables did not change. In addition to that, the previous check ensured that the type environment does not change when invoking a method. The changes to the heap are of course visible to the calling method.

Qualifier polymorphism

Qualifier polymorphic method signatures are automatically instantiated. The (Resolve) rule replaces each formal type parameter α (ambiguously) with one of the method's actual parameters' types $q_s^\star$ or one of their supertypes in the current type environment Φ.

For bounds checking, we introduce the relation $\lhd \in \mathit{VarImmType} \times \mathit{Bounds}$:

$$q_s \lhd \mathit{Any} \text{ for all } q_s \in \{\mathit{Rd}, \mathit{RdWr}, \mathit{Any}\}$$
$$q_s \lhd \mathit{Qual} \text{ for all } q_s \in \mathit{VarImmType} \setminus \{\mathit{NullT}\}$$
$$q_s \lhd \mathit{Writeable} \text{ for all } q_s \in \mathit{VarWriteableTypes}$$

$$\frac{}{\Phi, \underbrace{(\mathit{nil}, ms)}_{qpms} \vdash \mathit{Resolve} \ldots \Downarrow ms} \textsc{Resolve-Empty}$$

$$\frac{\begin{array}{c} q_s \in q_s^\star \qquad p_s \in \mathrm{rng}(\Phi) \qquad q_s <:_s p_s \qquad p_s \lhd B \\ ms' = ms[\alpha/p_s] \qquad \Phi, (\overline{(\alpha, B)}, ms') \vdash \mathit{Resolve}\, q_s^\star \Downarrow ms'' \end{array}}{\Phi, \underbrace{((\alpha, B) :: \overline{(\alpha, B)}, ms)}_{qpms} \vdash \mathit{Resolve}\, q_s^\star \Downarrow ms''} \textsc{Resolve-Cons}$$

Control flow statements

For the control flow statement rules (Figure 18), we introduce a function *restrict* to restrict the set of variables in Φ to the set $S \subseteq \mathit{VarID}$ when leaving a scope. The function is only defined for $S \subseteq \mathrm{dom}(\Phi)$.

$$\mathit{restrict}(\Phi, S) := \begin{cases} \Phi & \text{if } \mathrm{dom}(\Phi) = S \\ \mathit{restrict}(\Phi\langle \mathit{delete}\ q_s \rangle, S) & \text{if } q_s \in \mathrm{dom}(\Phi) \setminus S \end{cases}$$

Unlike $\Phi|S$, this operation also cares about adjusting the Fresh types appropriately. For restricting l, the usual restriction operator $l|S$ is used.

Example 20. When applying the Fresh-aware restriction function, the visible variables in the Fresh clouds are restricted as well.

$$\mathit{restrict}(\{x \mapsto F_s(\{x, y\}), y \mapsto F_s(\{x, y\})\}, \{x\}) = \{x \mapsto F_s(\{x\})\}$$

□

$$\frac{l(x) \neq \mathit{nullID} \qquad l, \Phi, h \vdash \mathit{thenBlock} \to l', \Phi', h' \qquad l'' = l'|\,\mathrm{dom}(l) \qquad \Phi'' = \mathit{restrict}(\Phi', \mathrm{dom}(\Phi))}{l, \Phi, h \vdash \texttt{if}\ x\ \mathit{thenBlock}\ \mathit{elseBlock} \to l'', \Phi'', h'}\ \textsc{If-True}$$

$$\frac{l(x) = \mathit{nullID} \qquad l, \Phi, h \vdash \mathit{elseBlock} \to l', \Phi', h' \qquad l'' = l'|\,\mathrm{dom}(l) \qquad \Phi'' = \mathit{restrict}(\Phi', \mathrm{dom}(\Phi))}{l, \Phi, h \vdash \texttt{if}\ x\ \mathit{thenBlock}\ \mathit{elseBlock} \to l'', \Phi'', h'}\ \textsc{If-False}$$

$$\frac{l(x) \neq \mathit{nullID} \qquad l, \Phi, h \vdash \bar{s} \to l', \Phi', h' \qquad l'' = l'|\,\mathrm{dom}(l) \qquad \Phi'' = \mathit{restrict}(\Phi', \mathrm{dom}(\Phi))}{l, \Phi, h \vdash \texttt{while}\ x\ \bar{s} \to l'', \Phi'', h'}\ \textsc{While-True}$$

$$\frac{l(x) = \mathit{nullID}}{l, \Phi, h \vdash \texttt{while}\ x\ \bar{s} \to l, \Phi, h}\ \textsc{While-False}$$

Figure 18: Control flow statement rules

4.11 How to Execute Programs

How to execute programs

A TIFI+ program is an abstract syntax tree *pdef* and a list of statements $\bar{s}$. To execute it using the operational semantics, we find a tree of rule applications for $\{\mathit{null} \mapsto \mathit{nullID}\}, \{\mathit{null} \mapsto \mathit{NullT}\}, \emptyset \vdash \bar{s} \to l', \Phi', h'$.

Because of the commit rules' ambiguity, the whole program execution is ambiguous as well. Luckily, as this is only a property of the immutability checks, we can later make the program execution unambiguous again by statically checking the immutability constraints and backtracking through all possibilities in the type checker.

5 Type Inference Algorithm

As the name already suggests, TIFI+'s type inference / type checking algorithm does multiple things at the same time. The algorithm is executed for each occurrence of a method body in a program and answers the following question:

> Given a method body and method signature, can the method always be executed without breaking the immutability property?

Recall that method bodies do not contain any immutability annotations or explicit commit statements. Consequently, the algorithm needs to do the following three things:

- **Track immutability types** along the path of execution. Variable types always reflect the types of the referenced objects, so when local variables are assigned within a method, the type environment after this statement is changed as well.
- **Infer commit statements**. Commits are always done as late as possible. Just like in the operational semantics, they can be enforced before method invocations or field assignments, when object references are potentially exposed to other methods.
- **Check the soundness** of commands with respect to the inferred type environments in whose context they are executed.

The type inference / type checking algorithm's basic strategy is to start with a suitable type environment (mapping from local variables to immutability types), then applying inference rules for each of the method body's statements to it, modifying the type environment along the way.

Whenever we infer over a statement where type compatibility is enforced for variables, we do implicit commits if needed and check that the statement can be executed soundly with the resulting type environment. If we can reach the end of a method without breaking the soundness property, it is considered sound.

Example 21. The simplest kind of method for the inference algorithm is that without control flow statements and commits. Consider the method `m()` in Listing 12.

The inference over `m()`'s body is started by determining the type environment when entering the method. The only visible variables are the method argument x, the implicit argument *this*, and *nullID*, which is also modeled as variable in our formalization. The types for *this* and x are determined

```
void m(Rd x) RdWr {
  var v;
  v = x;
  m2(v);
}

void m2(Rd x) RdWr {
  // ...
}
```

Listing 12: A method `m()` without If-Then-Else and loop statements.

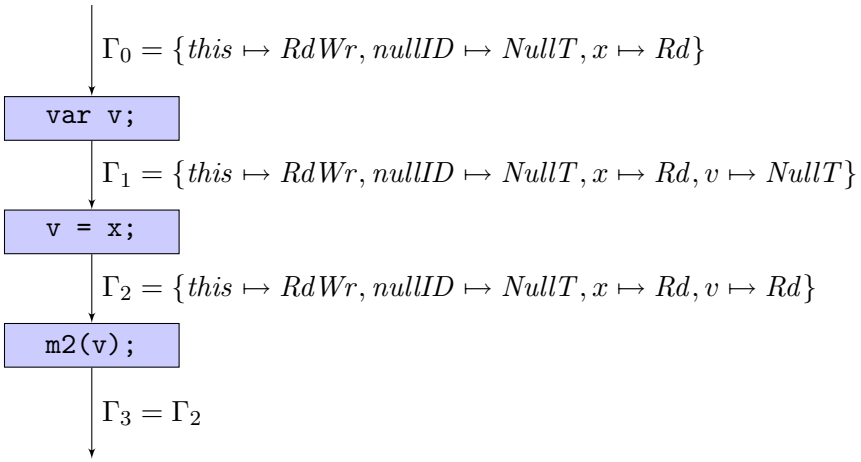

Figure 19: Schematic view of the inference over `m()` (Listing 12)

from the method signature, while the type for *nullID* is always *NullT*. The initial type environment is thus

$$\Gamma_0 = \{this \mapsto RdWr, nullID \mapsto NullT, x \mapsto Rd\}$$

We now successively infer an output type environment for an input type environment, always passing the current statement's output as input to the next (Figure 19).

The inference steps for the individual statements can now be executed step by step.

`var v;` This statement introduces a new variable v. Initially, the variable's type is set to $NullT$, so that Γ_1 is calculated by augmenting Γ_0 with v:

$$\begin{aligned}\Gamma_1 &:= \Gamma_0!\langle v \mapsto NullT\rangle \\ &= \{this \mapsto RdWr, nullID \mapsto NullT, x \mapsto Rd, v \mapsto NullT\}\end{aligned}$$

`v = x;` On local variable assignment, the immutability types of the variables are tracked by assigning x's type to v:

$$\begin{aligned}\Gamma_2 &:= \Gamma_1\langle v \mapsto \Gamma_1(x)\rangle \\ &= \Gamma_1\langle v \mapsto Rd\rangle \\ &= \{this \mapsto RdWr, nullID \mapsto NullT, x \mapsto Rd, v \mapsto Rd\}\end{aligned}$$

`m2(v);` For method invocations, it is ensured that the actual arguments' types are compatible to the corresponding formal parameters in the method signature. If this is not already the case, suitable commits may be implicitly placed before the method call, which result in a compatible type environment $\Gamma_\star$.
In this example, no commits are needed, so that $\Gamma_\star := \Gamma_2$. As all methods guarantee not to commit any of the caller-visible Fresh types passed to it, this allows us to set $\Gamma_3 := \Gamma_\star$.

□

5.1 Satisfying Immutability Properties

Some statements, like method calls and field assignments, require the variables used by them to have specific immutability types or relations between those. To satisfy these requirements, commits may be necessary beforehand.

Like in the TIFI paper, these commits are inferred using a lazy strategy: For example, when a Fresh typed variable is used in a context where *Rd*, *RdWr* or *Any* are expected, the Fresh type is committed appropriately directly by the same rule requiring this.

The mechanism used to infer commits is the same as in the operational semantics, where the decision what to commit to already depends solely on the static type environment Φ.

It is important to note that this step may introduce ambiguities (Example 22). In that case, there are two possible ways to satisfy the immutability constraint by committing, but they are incompatible to each other, i.e. we cannot know which of the two possible paths through the method will work later on. The inference algorithm resolves this by trying both possibilities.

```
void m() {
  var x = new C();
  any(x);
  rdwr(x);
}
```

Listing 13: Method `m()`

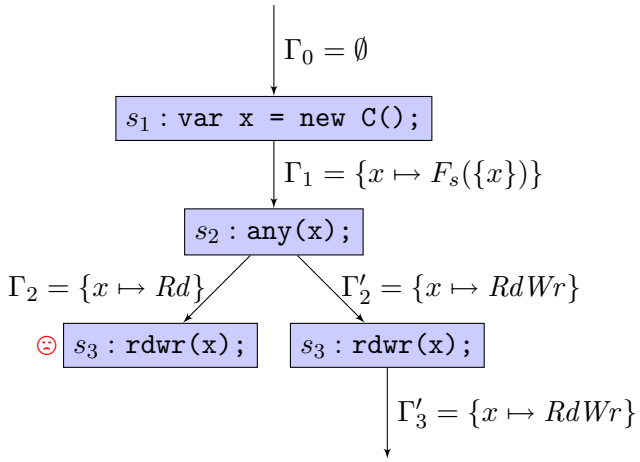

Figure 20: Schematic view of the inference over `m()` (Listing 13)

Example 22. Consider the method in Listing 13. The methods `any()` and `rdwr()` both expect arguments of the equally named types.

Inferring through this method begins with a type environment where only the immutability type of the implicit argument `this` is known. We will ignore this variable for this example, though, so we start with $\Gamma_0 = \emptyset$.

After interpreting the first statement[6], the resulting type environment Γ_1 contains the freshly introduced variable x, with the type $F_s(\{x\})$.

The method's second statement is harder to interpret: The method `any()` expects an argument of type Any. This constraint however, can be satisfied by two incompatible types of commits: On one hand, we can commit $F_s(\{x\})$ to Rd, on the other hand we can commit it to $RdWr$. At this point in the algorithm, we cannot already know which of the two possibilities leads to success further on, so we give both Γ_2 and Γ_2' as output and try both possibilities.

Finally, the third and last method statement reveals which of the two possibilities works: Trying to infer through s_3 using Γ_2, the algorithm notices that x's type is not compatible to the expected type $RdWr$. By giving an empty set of resulting type environments, it cuts off this path along the tree of ambiguities, as it is obviously not possible to reach the method's end in

[6] In the actual Operational Semantics, these are actually two separate statements.

a sound way along this path.

Luckily, the other input type environment Γ'_2 to the statement s_3 already has x's type set to $RdWr$, so we can still find a way to execute the method soundly, Γ'_3 being the output type environment.

□

5.2 Statements without Control Flow

Summing up, the type inference rules for the statements except If-Then-Else and While can be directly derived from their core rules in the operational semantics, stripping them from their interpretation aspect and renaming Φ to Γ.

The full set of inference rules for the non-control-flow statements is given in Figure 21.

Obviously, the inference rules build on the same layered design as the operational semantics do. As we did for the core rules, we strip the dynamic aspect (l and h) from the other layers as well.

Apart from being stripped from their interpretation aspect, differences to the operational semantics rules include:

- *hasField*() checks are removed. Instead, we just assume these fields exists. In Java, this check is already done before the immutability type checker is run, so it is not a problem.

- In (Call), we cannot retrieve the method signature properly, as the class information is statically not available. Again, this is not a problem in Java.

Note that the (Call) rule became a lot simpler after stripping out the interpretation aspect. It is sufficient now to check that the method usage adheres to the declared type, but we do not need to infer through the method body any more. Recall that in the operational semantics, the only thing used from the method body's evaluation result was the heap h_{bout}. This is obviously not needed for the type inference, so everything related to inferring through the method body can be omitted.

Apart from that, we also left out checking that the method does unexpected commits. In our modular type checker, all methods are checked independently at compile time, so we can just assume that all used methods have been checked to keep caller-visible (static) Fresh types untouched.

5.3 Control Flow Statements

Another problem we ignored until now is how to handle control flow statements like loops and If-Then-Else.

Figure 21: Type inference rules for non-control flow statements

$$\frac{\Gamma' = \Gamma!\langle x \mapsto \mathit{NullT}\rangle}{\Gamma \vdash \texttt{var}\ x \Downarrow \Gamma'}\ \textsc{Var-Decl} \qquad \frac{x \neq \mathit{null} \qquad \Gamma' = \Gamma\langle x \mapsto F_s(\emptyset)\rangle}{\Gamma \vdash x = \texttt{new}\ \mathit{cid}() \Downarrow \Gamma'}\ \textsc{New}$$

$$\frac{x_1 \neq \mathit{null} \qquad \Gamma' = \Gamma\langle x_1 \mapsto \Gamma(x_2)\rangle}{\Gamma \vdash x_1 = x_2 \Downarrow \Gamma'}\ \textsc{Local-Assign}$$

$$\frac{x_1 \neq \mathit{null} \qquad \Gamma' = \Gamma\langle x_1 \mapsto \mathit{resolveFieldType}(\Gamma(x_2), \mathit{fid})\rangle}{\Gamma \vdash x_1 = x_2.\mathit{fid} \Downarrow \Gamma'}\ \textsc{Field-Get}$$

$$\frac{\begin{array}{c}\Gamma(x_1) \in \mathit{VarWriteableTypes} \\ \Gamma \vdash x_2!!\mathit{resolveFieldType}(\Gamma(x_1), \mathit{fid}) \Downarrow \Gamma'\end{array}}{\Gamma \vdash x_1.\mathit{fid} = x_2 \Downarrow \Gamma'}\ \textsc{Field-Assign}$$

$$\frac{\begin{array}{c}\mathit{msdecl} = \mathit{methodSig}(\mathit{mid}, \ldots) \\ \Gamma, \mathit{msdecl} \vdash \mathit{Resolve}\ \{\Gamma(x) \mid x \in \bar{x}\} \Downarrow \mathit{ms} \\ \Gamma, \mathit{ms} \vdash x_r :: \bar{x}!\mathit{recvtype}(\mathit{ms}) :: \mathit{fpTypes}(\mathit{ms}) \Downarrow \Gamma', \mathit{ms}'\end{array}}{\Gamma \vdash x_r.\mathit{mid}(\bar{x}) \Downarrow \Gamma'}\ \textsc{Call}$$

To handle those, we introduce a Merge-Operation $\sqcup$, which, when given two type environments Γ_1, Γ_2, calculates a "compatible" type environment $\Gamma_\star$, so that when the end of the method can be reached using $\Gamma_\star$, it can also be reached using both Γ_1 and Γ_2. This compatibility can be ensured independent from the remaining sequence of statements.

Definition 1 (Check-implication)**.** Let $\mathit{check} : \mathit{TypeEnv} \times \mathit{ListOf}(\mathit{Statement}) \to \{\mathit{True}, \mathit{False}\}$ be the function that returns *True* if there is a sound path through the sequence of statements $\bar{s}$, when using the given type environment. Then the statement

$$\forall \bar{s} \in \mathit{ListOf}(\mathit{Statement}) : \mathit{check}(\Gamma_\star, \bar{s}) \Rightarrow \mathit{check}(\Gamma_1, \bar{s})$$

is called "$\Gamma_\star$ check-implies Γ_1" and written as

$$\Gamma_\star \overset{c}{\Rightarrow} \Gamma_1$$

.

The check-implication is a partial relation on the set of type environments *TypeEnv*.

Example 23.

$$\Phi_1 = \{x \mapsto \mathit{Any}\} \overset{c}{\Rightarrow} \{x \mapsto \mathit{RdWr}\} = \Phi_2$$

Φ_1 check-implies Φ_2, because *RdWr* variables can be used in all places where *Any* is expected, while the converse is not true. The programs executable using Φ_2 are a superset of those executable using Φ_1.

□

Example 24.

$$\Phi_1 = \{x \mapsto RdWr\} \overset{c}{\Rightarrow} \{x \mapsto F_S(\{x\})\} = \Phi_2$$

Whenever a sequence of statements is executable using Φ_1, it is also executable using Φ_2: If a statement requires x to be of type *RdWr*, it can simply be committed to that.

□

Example 25.

$$\Phi_1 = \{x \mapsto F_s(\{x, y\}), y \mapsto F_s(\{x, y\})\} \overset{c}{\Rightarrow} \{x \mapsto F_s(\{x\}), y \mapsto F_s(\{y\})\} = \Phi_2$$

Programs executable using Φ_1 can be easily executed using Φ_2 by first merging the two Fresh types.

□

Definition 2 (Sound operations)**.** We can define a set of "sound operations" on type environments Φ. If during execution, we replace Φ by a type environment with one or more of these operations applied, we preserve soundness, but restrict the set of possible statement sequences which can still be executed. These operations are

Upcasting the type of a variable in Φ.

Committing a Fresh type in Φ to *Rd* or *RdWr*.

Merging two Fresh types in Φ.

Derivations

Building upon this, the set of a type environment Φ's *derivations* $\mathcal{D}(\Phi)$ is defined as the set of all type environments Φ' so that Φ' can be derived from Φ by applying a (possibly empty) sequence of sound operations.

For type environments $\Phi, \Phi' \in \mathit{TypeEnv}$,

$$\Phi' \in \mathcal{D}(\Phi) \text{ implies } \Phi' \overset{c}{\Rightarrow} \Phi$$

Proof: Let $\bar{s} \in \mathit{ListOf}(\mathit{Statement})$, so that $\mathit{check}(\Phi', \bar{s})$. An execution of $\bar{s}$ using Φ may choose to first do the operations needed to derive Φ' from Φ, then execute the statements. Obviously, $\mathit{check}(\Phi, \bar{s})$ holds as well.

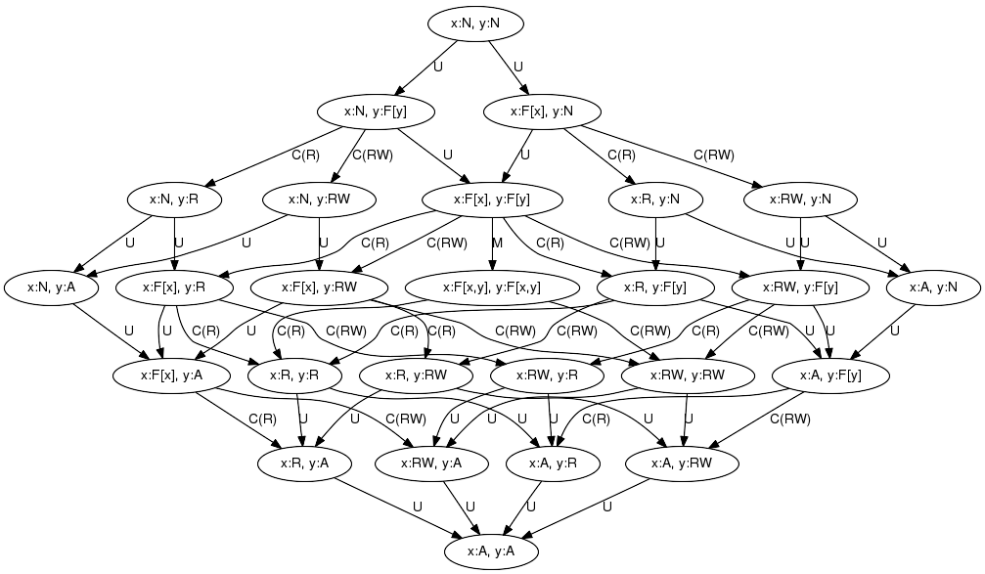

Figure 22: Derivation of type environments with two variables

Example 26. Figure 22 illustrates the relationship between type environments with two variables with respect to how they can be derived from each other using the three sound operations.

Each node in the graph represents a type environment. Arrows represent sound operations according to Definition 2, the pointed-to type environment is the operation's result. The arrows' labels designate the operation type according to the following table:

abbreviation	meaning
U	upcast
$C(R)$	commit to Rd
$C(RW)$	commit to $RdWr$
M	merging Fresh types

To save space, type names are abbreviated using their first letter, $RdWr$ is abbreviated RW.

□

5.3.1 Merge-Operation

Definition 3 (Merge operation)**.** Two type environments $\Gamma_1, \Gamma_2 \in \mathit{TypeEnv}$ can be *merged* using the merge operation

$$\sqcup : \mathit{TypeEnv} \times \mathit{TypeEnv} \to \mathrm{Pot}(\mathit{TypeEnv})$$

so that

$$\Gamma_1 \sqcup \Gamma_2 := \{\Gamma_\star \mid \Gamma_\star \text{ minimal so that } \Gamma_\star \overset{c}{\Rightarrow} \Gamma_1 \text{ and } \Gamma_\star \overset{c}{\Rightarrow} \Gamma_2\}$$

where the property of minimality for $\Gamma_\star$ means that there is no other type environment $\Gamma_\star{}'$ so that $\Gamma_\star{}'$ satisfies the properties $\Gamma_\star{}' \overset{c}{\Rightarrow} \Gamma_1$, $\Gamma_\star{}' \overset{c}{\Rightarrow} \Gamma_2$ and $\Gamma_\star{}' \overset{c}{\Rightarrow} \Gamma_\star$

More intuitively, for Γ_1 and Γ_2, $\sqcup$ searches all $\Gamma_\star$ so that $\Gamma_\star$ can be derived from both Γ_1 and Γ_2 with a minimal sequence of sound operations.

Note that again, merge turns out to be an ambiguous operation: There can be multiple type environments $\Gamma_\star$ check-implying both Γ_1 and Γ_2.

Example 27 (Merge is ambiguous)**.** For conciseness, this example uses colons instead of „↦" to denote type environments.

Consider merging the type environments Γ_1 and Γ_2:

$$\Gamma_1 := \{x : F_s(\{x, y\}), y : F_s(\{x, y\}), z : RdWr\}$$
$$\Gamma_2 := \{x : Rd, y : F_s(\{y, z\}), z : F_s(\{y, z\})\}$$

There are multiple minimal type environments check-implying both Γ_1 and Γ_2:

$$\begin{aligned}
&\text{In } \Gamma_1, \text{commit } F_s(\{x, y\}) \text{ to } Rd && \rightsquigarrow \{x : Rd, y : Rd, z : RdWr\}\\
&\quad \text{upcast } z \text{ to } Any && \rightsquigarrow \{x : Rd, y : Rd, z : Any\}\\
&\text{In } \Gamma_2, \text{commit } F_s(\{y, z\}) \text{ to } Rd && \rightsquigarrow \{x : Rd, y : Rd, z : Rd\}\\
&\quad \text{upcast } z \text{ to } Any && \rightsquigarrow \{x : Rd, y : Rd, z : Any\}
\end{aligned}$$

The resulting type environment $\Gamma_\star{}' := \{x : Rd, y : Rd, z : Any\}$ check-implies both Γ_1 and Γ_2.

$$\begin{aligned}
&\text{In } \Gamma_1, \text{commit } F_s(\{x, y\}) \text{ to } RdWr && \rightsquigarrow \{x : RdWr, y : RdWr, z : RdWr\}\\
&\quad \text{upcast } x \text{ to } Any && \rightsquigarrow \{x : Any, y : RdWr, z : RdWr\}\\
&\text{In } \Gamma_2, \text{commit } F_s(\{y, z\}) \text{ to } RdWr && \rightsquigarrow \{x : Rd, y : RdWr, z : RdWr\}\\
&\quad \text{upcast } x \text{ to } Any && \rightsquigarrow \{x : Any, y : RdWr, z : RdWr\}
\end{aligned}$$

The resulting type environment $\Gamma_\star{}'' := \{x : Any, y : RdWr, z : RdWr\}$ check-implies both Γ_1 and Γ_2.

$$\begin{aligned}
&\text{In } \Gamma_1, \text{commit } F_s(\{x, y\}) \text{ to } Rd && \rightsquigarrow \{x : Rd, y : Rd, z : RdWr\}\\
&\quad \text{upcast } y \text{ to } Any && \rightsquigarrow \{x : Rd, y : Any, z : RdWr\}\\
&\text{In } \Gamma_2, \text{commit } F_s(\{y, z\}) \text{ to } RdWr && \rightsquigarrow \{x : Rd, y : RdWr, z : RdWr\}\\
&\quad \text{upcast } y \text{ to } Any && \rightsquigarrow \{x : Rd, y : Any, z : RdWr\}
\end{aligned}$$

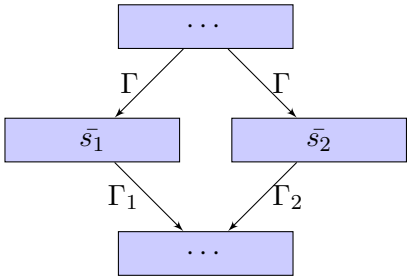

Figure 23: Typical control flow graph for an If-Then-Else statement. The statement sequences / subgraphs in the branches are collapsed into a single node.

The resulting type environment $\Gamma_\star''' := \{x : Rd, y : Any, z : RdWr\}$ check-implies both Γ_1 and Γ_2.

It is clear that $\Gamma_\star', \Gamma_\star''$ and $\Gamma_\star'''$ all check-imply both input type environments, as they are constructed from them using only the three sound operations (see Definition 2).

However, $\Gamma_\star', \Gamma_\star''$ and $\Gamma_\star'''$ do not check-imply each other. This is clear because for each of them, we can give a sequence of statements which is only executable with it, but not with the others:

- `rd(x);rd(y)` can only be executed using $\Gamma_\star'$
- `rdwr(y);rdwr(z)` can only be executed using $\Gamma_\star''$
- `rd(x);rdwr(z)` can only be executed using $\Gamma_\star'''$

As usual, the methods `rd()`, `rdwr()` and `any()` each expect one argument of the equally named type.

As $\Gamma_\star', \Gamma_\star''$ and $\Gamma_\star'''$ all check-imply both Γ_1 and Γ_2, but they do not check-imply each other, $\Gamma_\star', \Gamma_\star'', \Gamma_\star''' \in \Gamma_1 \sqcup \Gamma_2$.

□

5.3.2 If-Then-Else

If-Then-Else statements, when seen as a control flow graph have the form depicted in Figure 23. Using the merge operation $\sqcup$, we can now infer through If-Then-Else statements by inferring through both possible paths of execution, then merging their results.

The input type environment is used as input to both the sequence of statements in the then-case $\bar{s}_1$ and those in the else-case $\bar{s}_2$. The resulting

$$\frac{\Gamma \vdash \bar{s_1} \Downarrow \Gamma_1 \qquad \Gamma \vdash \bar{s_2} \Downarrow \Gamma_2 \qquad \Gamma_{out} = \Gamma_1 \sqcup \Gamma_2}{\Gamma \vdash \texttt{if}\ x\ \bar{s_1}\ \bar{s_2} \Downarrow \Gamma_{out}}\ \textsc{If-Then-Else}$$

Figure 24: If-Then-Else inference rule

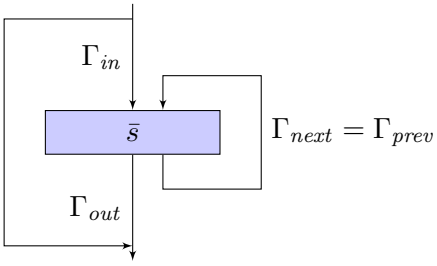

Figure 25: Control flow graph for While loops. The left hand side arrow represents the case where the body is not executed at all.

type environments Γ_1 and Γ_2 are then merged to form the type environment Γ_{out}, which is the output type environment of the If-Then-Else statement. The rule is given in Figure 5.3.2.

5.3.3 Loops

The control flow graph for a while loop is given in Figure 25.

We cannot statically tell how often a loop will be executed at runtime. Thus, a loop's output type environment needs to be valid for all possible numbers of loop iterations. (It needs to check-imply the output environment after all numbers of iterations.)

The problem is solved here by doing a fix point iteration, the fix point being the while statement's output environment. Each fix point candidate Γ_{next} is calculated from the previous fix point candidate Γ_{prev} by

1. merging Γ_{prev} with Γ_{in}
2. inferring through the loop body $\bar{s}$, using one of the previous step's result type environments as input.

These two steps are repeated until a fix point ($\Gamma_{prev} = \Gamma_{next}$) is found. The commit decisions of the last step in the fix point iteration are then a strategy which would give the same results for all subsequent loop body inferences. The complete algorithm is given in Figure 26.

It is important to note that the output type environment of the loop itself, Γ_{out}, is not the same as the last calculated Γ_{next}, as it would be the case in a classic data flow analysis. Instead, we explicitly merge all calculated

$$\frac{\Gamma' = \Gamma_{in} \sqcup \Gamma_{prev} \qquad \Gamma' \vdash \bar{s} \Downarrow \Gamma_{prev}}{\Gamma_{in}, \Gamma_{prev}, \Gamma_{pot} \vdash \mathit{fpiter}\ \bar{s} \Downarrow \Gamma_{pot}}\ \textsc{Find-FP-Final-Step}$$

$$\frac{\Gamma' = \Gamma_{in} \sqcup \Gamma_{prev} \qquad \Gamma' \vdash \bar{s} \Downarrow \Gamma_{next} \qquad \Gamma_{next} \neq \Gamma_{prev} \qquad \Gamma_{in}, \Gamma_{next}, (\Gamma_{next} \sqcup \Gamma_{pot}) \vdash \mathit{fpiter}\ \bar{s} \Downarrow \Gamma_{out}}{\Gamma_{in}, \Gamma_{prev}, \Gamma_{pot} \vdash \mathit{fpiter}\ \bar{s} \Downarrow \Gamma_{out}}\ \textsc{Find-FP-Search}$$

$$\frac{\Gamma_{in} \vdash \bar{s} \Downarrow \Gamma_{next} \qquad \Gamma_{in}, \Gamma_{next}, \Gamma_{next} \vdash \mathit{fpiter}\ \bar{s} \Downarrow \Gamma_{out}}{\Gamma_{in} \vdash \texttt{while}\ x\ \bar{s} \Downarrow (\Gamma_{out} \sqcup \Gamma_{in})}\ \textsc{Infer-While}$$

Figure 26: Inferring through While loops

```
var x;
while (cond) {
  x = new C();
  any(x);
}
```

Listing 14: A loop which can be inferred through in multiple ways.

loop body outputs Γ_{next} into a type environment Γ_{pot} and return this when a fix point is found.

Because of Γ_{pot}'s construction, the output type environment of any previously calculated loop body execution can easily be weakened to Γ_{pot} by applying a sequence of the sound weakening operations (see Definition 2). Mathematically speaking, for all previously calculated Γ_{next}, the property $\Gamma_{pot} \overset{c}{\Rightarrow} \Gamma_{next}$ holds.

Note that these steps often give ambiguous results. It is interesting to note that within a path in the tree of ambiguities, different commits can be done before the same AST node.

Example 28 (Ambiguity)**.** Consider the while loop in Listing 14.

Inferring through this loop, x first gets assigned a Fresh type ($\Gamma = \{x : F_s(\{x\})\}$), then it gets committed to *Rd* or *RdWr*. Looking at only one path through the tree of ambiguities, in both the operational semantics and the type inference algorithm, the decision what to commit to may be taken differently in different loop iterations.

Figure 27 shows the tree of commit possibilities when inferring over the while loop in Listing 14. Circles indicate the type environments between inferring through the loop body ($\Gamma_{prev}, \Gamma_{next}$). The loop body inference is shown as an arrow. If we found a fix point, this is indicated by a filled circle.

Although we find fix points, there are two paths where the decision

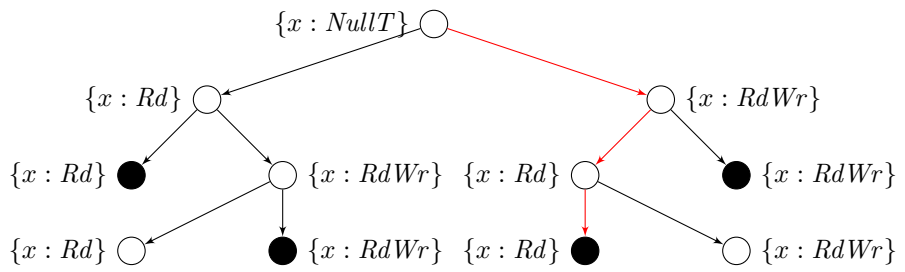

Figure 27: Tree of ambiguities when inferring through the loop in Listing 14

oscillates between committing to *Rd* and *RdWr*. Consequently, in each step, $\Gamma_{next} \neq \Gamma_{prev}$ on these paths, so they continue infinitely without ever reaching a fix point.

However, at the same time we can already find both the $\{x : Rd\}$ and $\{x : RdWr\}$ fix point very early when descending into the tree of ambiguities.

□

We have seen in Example 28 that there can be paths in the tree of ambiguities, which never lead to finding a fix point. To avoid searching infinitely in our implementation, we set a hard limit for the depth of the ambiguity tree.

It is crucial to note that unlike data flow analysis algorithms [14] [3], we cannot guarantee finding the existing fix points. (For each $n \in \mathbb{N}$, it is possible to construct a loop and an input type environment for it, whose body needs to be inferred through more than n times to find its fix point.[7]) We hope that in practice, a limited number of loop iterations are sufficient. If needed, a compiler switch can be introduced to increase the limit.

Example 29 (Unrolling loops to place explicit commit statements)**.** Consider the path along the tree of ambiguities (Figure 27) which is marked red. In the first loop iteration we commit to *RdWr*, in the second and third iteration to *Rd*.

It is not possible to directly place explicit commit statements in the source code, like we did in the TIFI introduction chapter in Example 10. However, we can give an equivalent transformed source code where it is possible. This is done by unrolling the loop's first $n - 1$ iterations, n being the number of iterations it took to find the fix point.

[7] The basic idea for constructing these loops is to introduce $n+2$ variables x_i $(0 \leq i \leq n)$, all of type *RdWr*, except x_0, which is of type *Rd*. The loop body is a sequence of statements $x_{n+1} = x_n; x_n = x_{n-1}; \ldots; x_1 = x_0$. The obvious result is that all variables are of type *Any*, but it will take $n + 1$ loop body inferences to find it.

```
var x;
if (cond) {
  x = new C();
  commit typeof(x) to RdWr;
  any(x);
  if (cond) {
    x = new C();
    commit typeof(x) to Rd;
    any(x);
    while (cond) {
      x = new C();
      commit typeof(x) to Rd;
      any(x);
    }
  }
}
```

Listing 15: A loop-unrolled variant of Listing 14

For our specific path, the loop in Listing 14 transforms to a rolled-out variant (Listing 15). The corresponding control flow graph is shown in Figure 28.

□

5.4 Running the Inference Algorithm

The inference algorithm is run on a per-method basis. Apart from trying to show that the method's statements are sound, it also needs to show that the checked methods conform to the method level constraints for handling Fresh qualifiers (Section 2.9.3). Furthermore, it also needs to be able to check qualifier polymorphic methods.

Qualifier polymorphism

Qualifier polymorphic methods are checked in a simple, but obviously correct way. Assuming that most qualifier polymorphic methods use only a small number of formal immutability type parameters, and that most methods are rather short, we enumerate all possible instantiations of a method's qualifier polymorphic signature. We then check for each of the instantiated signatures that the method body is sound.

Commit constraints

We also need to ensure that the Fresh types visible for the method's caller cannot be changed by the method. This step needs to be done for each method signature instantiated from a qualifier polymorphic method signature. The only Fresh types visible to the caller are the ones given through the method parameters. Therefore, in order to check they have not

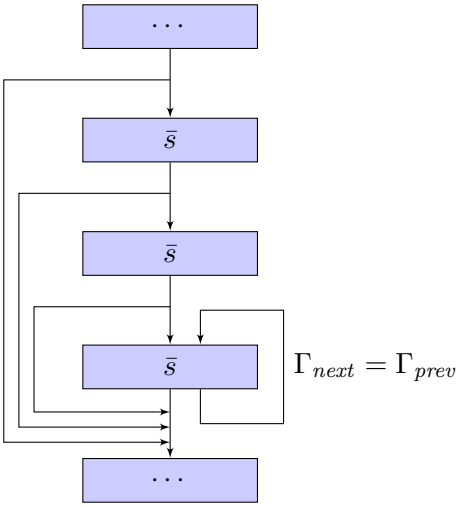

Figure 28: Control flow graph for Listing 15

been changed after the method, before inferring through the method body, we introduce a special variable $b_i \in \mathit{VarID}$ for each of the different Fresh types $q_{si}, 0 \leq i \leq n$ occurring in the instantiated signature. In the initial type environment Γ_{in} for checking the method body, each b_i's type is set to the respective Fresh type it was created for, $\Gamma_{in}(b_i) = q_{si}$.

Then, after having inferred through the method body, in the output type environment Γ_{out}, the variables $b_i, 0 \leq i \leq n$ will have tracked their Fresh types throughout the execution of the method. We can now check that these have not been changed in a way which is visible to the caller, that is,

- They have not been committed to one of the initialized types:

$$\nexists b_i \in \mathrm{dom}(\Gamma_{out})\ s.t.\ \Gamma_{out}(b_i) \in \{Rd, RdWr, Any\}$$

- There are no two externally visible Fresh types that have been merged with each other:

$$\nexists b_i, b_j \in \mathrm{dom}(\Gamma_{out}),\ \text{where}\ i \neq j\ s.t.\ \Gamma_{out}(b_i) = \Gamma_{out}(b_j)$$

These two properties are easily checkable.

5.5 Informal Soundness Proof

While a complete soundness proof is out of scope for this thesis, this section provides a short sketch of how it may be constructed.

Soundness property

Before we start with the proof idea itself, it is necessary to have a clear understanding of the soundness property:

$$\frac{\begin{array}{c}\Phi(x) = p_s \qquad q_s \in \text{types}(\Phi) \\ p_s <: q_s \qquad \Phi' = \Phi\langle x \mapsto q_s \rangle \qquad l, \Phi', h \vdash s \to l', \Phi'', h'\end{array}}{l, \Phi, h \vdash s \to l', Phi'', h'} \textsc{Upcast}$$

$$\frac{\begin{array}{c}p_s, q_s \in \text{types}(\Phi) \qquad p_s = F_s(\ldots) \\ q_s \neq NullT \qquad l, \Phi, h \vdash \text{commit } p_s \text{ to } q_s \Downarrow l', \Phi', h' \\ l', \Phi', h' \vdash s \to l'', \Phi'', h''\end{array}}{l, \Phi, h \vdash s \to l'', \Phi'', h''} \textsc{Commit or Merge}$$

Figure 29: Operational semantics rules for sound weakening of type environments

Objects of the (dynamic) type *Rd* will never be modified.

Note that the consistency of Φ and h within runtime configurations is not shown yet and must also be part of a soundness proof, which is why the soundness property is defined on dynamic types.

Recall that in a static type environment (Γ or Φ), objects of the dynamic type *Rd* can be referenced by both *Rd*- and *Any*-typed variables.

5.5.1 Operational Semantics

The (informal) proof is understood easiest using a slightly modified variant of the operational semantics presented in Section 4.7: In addition to the core rules, the rules in Figure 29 allow to weaken the type environment Φ by upcasting and commit types between statements.

The two rules (Upcast) and (Commit-Or-Merge) essentially allow to do the three sound weakening operations (Definition 2) in between statement evaluations.

5.5.2 Inferring Commits

The type inference / type checking algorithm infers commits in two distinct ways:

1. The inference algorithm rules (Field Assign) and (Method Invocation) infer commits implicitly by invoking the type compatibility rules.

2. When the inference algorithm merges type environments, it is clear that the resulting type environment can be derived from any of the input environment by applying a sequence of the sound weakening operations (Definition 2). If the inference finds a way through the checked method on which the used type environment is the result of

one or more merges, it is okay to apply the operational semantics rules (UPCAST) and (COMMIT OR MERGE) rules (Figure 29) in the corresponding places when executing the method.

5.5.3 Proof Overview

The proof is divided into two parts:

- In **Part 1**, we show that the type environment Φ in the operational semantics is always consistent with the actual dynamic types, that is, when only checking access permissions using Φ, the operational semantics can never make the mistake of writing to a dynamically *Rd*-typed object.
- In **Part 2**, we assume that the type inference / type checking algorithm found a way along the tree of ambiguous commit possibilities, on which it could reach the end of the method. Under this precondition, we show that we can derive from it a commit strategy for all possible execution paths through the method, so that these paths can be completely (and thus soundly) executed using the operational semantics.

5.5.4 Part 1

To show that Φ is always consistent to the dynamic types, we lift the definition of consistency to runtime configurations l, Φ, h.

Definition 4 (Consistent runtime configurations)**.** A runtime configuration l, Φ, h is called **consistent** iff for all variables $x \in \mathrm{dom}(\Phi)$:

$$
\begin{array}{llr}
\Phi(x) = \mathit{Any} & \text{implies} & l(x) = \mathit{nullID} \text{ or } I(h(l(x))) \in \mathit{Rd}, \mathit{RdWr} \\
\Phi(x) = \mathit{Rd} & \text{implies} & l(x) = \mathit{nullID} \text{ or } I(h(l(x))) = \mathit{Rd} \\
\Phi(x) = \mathit{RdWr} & \text{implies} & l(x) = \mathit{nullID} \text{ or } I(h(l(x))) = \mathit{RdWr} \\
\Phi(x) = F_s(x^\star) & \text{implies} & \forall y, z \in x^\star : \; l(y) = \mathit{nullID} \text{ or } \; l(z) = \mathit{nullID} \\
 & & \text{or } I(h(l(y))) = I(h(l(z))) = F_d(...) \\
\Phi(x) = \mathit{NullT} & \text{implies} & l(x) = \mathit{nullID}
\end{array}
$$

The property can be shown by structural induction over the operational semantics' core rules (statement rules): If a rule's input runtime configuration was consistent, it needs to follow that its output runtime configuration is consistent as well. If this holds for all statement rules, an inconsistent configuration can never be reached.

In the method invocation rule, we need to assume that the called method keeps the caller-visible Fresh types intact, i.e. an object passed to it cannot

be commited to one of *Rd*, *RdWr* or *Any*, and the types of two passed Fresh objects cannot be merged.

Doing this proof for the method invocation and field assignment rules quickly reveals that it must also be shown for the rules in the type compatibility and commit layer.

It helps understanding to observe that the method invocation and field assignment rules first do commits where appropriate, then apply the actual state change to the runtime configuration. For the state change, the output configuration's consistency can be proved in similar way as for the other (non-committing) statement types. The problem can then be reduced to showing that the commits done by these rules preserve configuration consistency as well.

For the commit layer rules (Fresh to Rd or RdWr) and (Fresh to Fresh) (Section 4.8), this is easy to show by checking the consistency properties for the changed variable types in Φ against the dynamic types on the updated heap[8].

The type compatibility layer has the special rule (Any below Fresh) (Section 4.9.2), which is the only higher-layer rule circumventing the two commit rules we already looked at. (Any below Fresh) is only used when evaluating the statement "`x.fid = y`", in a runtime configuration where $\Phi(x) = F_s(\{x, \ldots\})$, $\Phi(y) = Any$ and *fid* is annotated as *MyAccess*. In that case, x's dynamic type is committed to y's dynamic type *Rd* (*RdWr*), whereas x's static type in Φ is committed to *Any*. However, the commit to *Any* is just a simpler implementation strategy for committing to *Rd* (*RdWr*), depending on y's dynamic type, then upcasting to *Any*. This is easier to prove sound, as it can be rewritten using the commit layer rules we already showed to be sound.

5.5.5 Part 2

In the second part of the proof idea, we want to show that a successful run of the type inference / type checking algorithm guarantees that all paths through the method can be soundly executed.

We start by assuming that in one path through the tree of ambiguous commit possibilities, the type inference algorithm found a way through the whole method. Let us assume we're just looking at this specific path through the tree of ambiguities. We can then unroll the first iterations of all loops in the control flow graph, giving us a control flow graph where every basic block corresponds to a specific step in the inference (each basic block needed to be inferred through only once). The input type environment of

[8]The consistency properties for the variables with unchanged Φ-types stay the same, as their corresponding dynamic types weren't Fresh before. Consequently, they couldn't have changed.

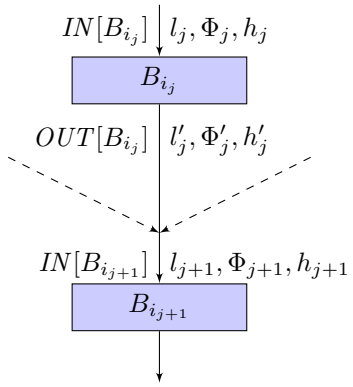

Figure 30: Control flow graph in step j. Dashed arrows indicate possible incoming control flows.

the inference step done on a block B_i is called $IN[B_i]$, the output type environment $OUT[B_i]$. (As we're looking at only one path through the tree of ambiguities, the output type environment is unambiguously defined within it.)

Let's now consider an arbitrary execution path through this control flow graph, consisting of steps $j = 0, 1, \ldots$ through basic blocks $B_{i_0}, B_{i_1}, \ldots$. Because of loops, is is possible that there are two steps j_0, j_1 referring to the same basic block $B_{i_{j_0}} = B_{i_{j_1}}$.

The initial runtime configuration l_0, Φ_0, h_0 serves as input for the first executed basic block B_{i_0}, yielding the output l'_0, Φ'_0, h'_0. This output in turn may be changed by the rules (Upcast) and (Commit-Or-Merge) to l_1, Φ_1, h_1, then serves as input for the basic block B_{i_1}, yielding l'_1, Φ'_1, h'_1, and so forth for bigger indices (Figure 30).

We now do an induction over j, showing in each step that there is a way to execute the operational semantics so that the input type environments for the next step equal in the operational semantics and the type inference algorithm, e.g. $\Phi_{j+1} = IN[B_{i_{uj+1}}]$.

The **base case** for the induction is $\Phi_0 = IN[B_{i_0}]$, which is obviously true.

Inductive step: We can assume that

$$\Phi_j = IN[B_{i_j}]$$

Because the inference steps through all basic blocks have been successful, we know that there is way to do an inference step through the next basic block B_{i_j}, which yielded $OUT[B_{i_j}]$. As the type inference rules do the same operations on the type environments as the operational semantics do, and especially, they get stuck in the same cases and have the same ambiguous branches, we know that it is possible to evaluate B_{i_j} using the operational

semantics, so that

$$\Phi'_j = OUT[B_{i_j}] \tag{1}$$

In the type inference algorithm, $IN[B_{i_{j+1}}]$ is calculated by merging the OUT-values of all predecessor blocks. From the definition of merge, we know that $IN[B_{i_{j+1}}]$ can then be derived from any of these output type environments by applying a sequence of the sound weakening operations (Definition 2).

In the operational semantics, Φ_{j+1} can be determined by applying the (Upcast) and (Commit-or-Merge) rules on Φ'_j. These allow to apply any sequence of the three sound weakening operations to Φ'_j, so we can choose to derive Φ_{j+1} in the same way as $IN[B_{i_{j+1}}]$ was derived from $OUT[B_{i_j}]$. This way, because of (1), we know that

$$\Phi_{j+1} = OUT[B_{i_{j+1}}]$$

By induction, this holds for all j, so all steps are executable using the operational semantics.

5.5.6 Proof Conclusion

Together with part 1 of the proof, which shows that the operational semantics never execute a method in an unsound way, we now know that the checked method can be executed properly using the operational semantics.

By removing all type information – both Φ and the types on the heap – from the operational semantics, all reasons for ambiguous rule evaluation disappear. We can therefore use this type system for normal, unambiguously evaluated object oriented languages like Java.

5.5.7 Comparison to Data Flow Analysis

Although strictly speaking, the TIFI+ inference algorithm is not a data flow analysis, it shares many common ideas with it. [3, Section 10.11] defines a framework for data flow analysis as a triple $(F, V, \wedge)$ where

- V is a set of *values*, corresponding to type environments
- F is a set of *transfer functions* from V to V, corresponding to inference rules for statements
- $\wedge$ is a binary *meet operation* from $V \times V$ to V, corresponding to TIFI+'s $\sqcup$.

Furthermore, [3] defines the relation $\leq$ on V so that for all $x, y \in V$: $x \wedge y \leq y$, which is a definition very similar to the check-implication $\overset{c}{\Rightarrow}$, where $\Gamma_\star \overset{c}{\Rightarrow} \Gamma_1$ and $\Gamma_\star \overset{c}{\Rightarrow} \Gamma_2$, if $\Gamma_\star$ is a result of merging Γ_1 and Γ_2.

```
void worksInTIFIPlusButNotInOriginalTIFI() {
  C c = new C();
  assertIsFresh(c);
  while (cond()) {
    commitToRd(c);  // Force c to be Rd.
    c = new C();
    assertIsFresh(c);
  }
  commitToRd(c); // Force c to be Rd.
}
```

Listing 16: A method which can be checked in TIFI+, but not in TIFI. (This source code is also included in the TIFI+ test suite.)

Although the similarities between data flow analysis and TIFI+'s inference algorithm are very high, there is still an unresolved mismatch in that TIFI+'s merge operation and its inference rules all have ambiguous results, hence it is not possible to model them as functions.

Further information on data flow analysis can be found in [3] and [14].

5.6 Comparison

Compared with the inference algorithm presented in the TIFI report [11], our algorithm is changed in a number of ways:

Cloud-style Fresh types: Fresh types are represented using sets of objects and variables, not using tokens. Using these "cloud-style" Fresh types simplifies the notion of type environment implication and merging (which are not mentioned in the original paper, but useful to understand the inference algorithm).

Another advantage of cloud-style Fresh types is that they also facilitate the equality of type environments (and thereby finding fix points): Consider the loop in Listing 16. TIFI+ can find the fix point $\Gamma_{fp} = \{c \mapsto F_s(\{c\})\}$ for this loop. In the original TIFI proposal, this is not possible: As the constructor call introduces a new Fresh token, the loop body's output type environment is always different than its input type environment. For example, if the input type environment is $\{c : \mathit{Fresh}(n)\}$, the output type environment could be $\{c : \mathit{Fresh}(n')\}$ $(n \neq n')$, which is not recognized as equal.

Not constraint-based: The original inference algorithm relies on accumulating a constraint set when traversing the method body. After traversal, if the constraint set can be fulfilled, the found `newtoken` and `commit` positions are the inference algorithm's result. If it cannot be fulfilled, these are not valid.

While a constraint-based approach is possibly formally simpler, it has the disadvantage that it becomes harder to emit reasonable error messages if the constraints cannot be fulfilled. TIFI+'s inference therefore avoids constraint sets.

Merging and Check-Implication: TIFI+ presents a way to reason about type environments using merging and check-implication, which proved to be useful to understand the ambiguity of merge. Although the TIFI type inference algorithm merges type environments in a similar way,

Fixes problems in the inference algorithm: The original inference algorithm has a bug (see Appendix A), because of which it isn't possible to infer commits for the following sequence of statements:

```
c = new C();   // c is Fresh.
any(c);   // Forces that c has a subtype of Any.
rd(c);    // Forces that c has a subtype of Rd.
```

A possible solution on basis of the original algorithm probably involves introducing more constraints [1] [2]. Our algorithm solves the problem using backtracking.

Complexity: A low algorithmic complexity was not a primary goal when developing the TIFI+ algorithm. It is not currently clear whether a solution using constraint solving is faster or slower.

6 Adaption to Java

The number of Java's features by far exceeds that of the restricted language used in the previous chapters. In some cases, mapping the type system built for that language to Java is a non-trivial task.

In this chapter, we will have a look at some of the most important aspects of transferring the TIFI+ type system to Java.

6.1 Expressions

Unlike the TIFI+ formalization language, Java has nested expressions. We introduce type inference rules for expressions of the form

$$\Gamma \vdash e \Downarrow \Gamma'$$

Expression evaluation

Nested expressions are usually evaluated left to right, then the outer operation is applied to the results ([9, Section 15.7])[9]. For example, to evaluate the expression $e_1 + e_2$ (e_1, e_2 being expressions themselves), first e_1 is evaluated, then e_2, then the addition operation is applied to both. Note that because of side effects, evaluation order matters.

Expression inference

When applying an expression inference rule to a type environment Γ, the result environment Γ' will span the variables $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{\mathcal{E}\}$, so that $\Gamma'(\mathcal{E})$ is the expression's type after its evaluation. $\mathcal{E}$ is a special variable used by all expression inference rules.

Expression inference rules expect that $\mathcal{E}$ is not part of the input environment ($\mathcal{E} \notin \mathrm{dom}(\Gamma)$).

Example 30 (Field assignment expression)**.** Consider using the inference rule

$$\Gamma \vdash x_1.\mathit{fid} = x_2 \Downarrow \Gamma'$$

where *fid* is annotated as *MyAccess* and Γ is set to

$$\Gamma := \{x_1 \mapsto F_s(\{x_1\}), x_2 \mapsto F_s(\{x_2\})\}$$

The rule merges the two fresh types and adds the variable $\mathcal{E}$ of the same type, yielding the type environment

$$\Gamma' := \{x_1 \mapsto F_s(\{x_1, x_2, \mathcal{E}\}), x_2 \mapsto F_s(\{x_1, x_2, \mathcal{E}\}), \mathcal{E} \mapsto F_s(\{x_1, x_2, \mathcal{E}\})\}$$

where all three variables are of the same Fresh type.

□

[9] An exception to this are only the conditional operators, `?:`, `&&` and `||`.

The naive approach to inferring through an expression e with nested subexpressions $\bar{e} = e_1 :: e_2 :: \ldots :: nil$ is to first infer through all of the nested subexpressions, collect the returned types, then proceed doing the actual inference, thereby possibly doing checks on the collected types. There is a problem with this approach, though: As we have seen in Example 30, some expressions, like field assigns and method calls can do commits when inferring through them. When for example e_2 does that, it can in particular commit the type of the previous expression, e_1 (Example 31).

Simply remembering each subexpression's type directly after inferring through it is therefore not sufficient. Instead, we need to find a way to track changes to these types throughout the inference of the other subexpressions.

Key to understanding the solution is to understand that every expression with subexpressions can be rewritten in a way so that all subexpressions are converted into simple variable usages. For example, the method invocation $m(x_1, (x_1.\mathit{fid} = x_2))$ can be rewritten[10] as

```
{
  var s1 = x1;
  var s2 = (x1.fid = x2);
  m(s1, s2);
}
```

Example 31 (The naive approach fails)**.** Consider inferring through a method invocation of the form "$m(x_1, (x_1.\mathit{fid} = x_2))$", where *fid* is annotated as *MyAccess*. We set $\Gamma := \{x_1 \mapsto F_s(\{x_1\}), x_2 \mapsto \mathit{RdWr}\}$ and infer by invoking the rule $\Gamma \vdash m(x_1, (x_1.\mathit{fid} = x_2)) \Downarrow \Gamma'$.

We start by inferring through the first expression, $\Gamma \vdash x_1 \mapsto \Gamma_{out1}$, which yields $\Gamma_{out1} = \{x_1 \mapsto F_s(\{x_1, \mathcal{E}\}), \mathcal{E} \mapsto F_s(\{x_1, \mathcal{E}\}), x_2 \mapsto \mathit{RdWr}\}$. We remember the first subexpression's type as $q_{s\,expr\,1} := \Gamma_{out1}(\mathcal{E}) = F_s(\{x_1, \mathcal{E}\})$, calculate $\Gamma_{in2} = \Gamma_{out1}\langle \mathit{delete}\ \mathcal{E}\rangle$, and proceed inferring through the second subexpression, $\Gamma_{in2} \vdash x_1.\mathit{fid} = x_2 \Downarrow \Gamma_{out2}$. This yields the result type environment $\Gamma_{out2} = \{x_1, x_2, \mathcal{E} \mapsto \mathit{RdWr}\}$. Again, we remember the resulting type $q_{s\,expr\,2} := \Gamma_{out2}(\mathcal{E}) = \mathit{RdWr}$ and remove $\mathcal{E}$ from the type environment.

Looking at the other subexpression type we remembered, it is $q_{s\,expr\,1} = F_s(\{x_1, \mathcal{E}\})$. However, when (abstractly) interpreting the second subexpression, we just committed this Fresh type. Consequently, when really executing this program, the type of x_1 after executing the first subexpression will be this Fresh type, but after evaluating all subexpressions, right before calling the method, the type is already committed to *RdWr*.

□

[10] For the explanation, we ignore that method calls themselves are expressions, too. To make this formally correct, we can extend the semantics of blocks to be used as expressions and to evaluate to the value of the last expression statement occurring in them. It is also possible to think about this as a `let` construct, as seen in functional languages.

Inferring through the statements in this block, the problem in Example 31 cannot occur any more: When the method call statement is only entered after introducing and assigning the variables s_1, s_2, the changes to x_1's old type have already been tracked in $\Gamma(s_1)$.

If the newly introduced *special variable identifiers* s_1 and s_2 do not occur in the rest of the program, the code transformation to an expression without subexpressions preserves the original semantics. We can thus change the inference rules for nested expressions to infer through the expressions as if they were instead written in the transformed style.

For example, a simplified[11], modified (Call) rule can be written as

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{var}\ \bar{x} = \bar{e} \Downarrow \Gamma' \qquad msdecl = methodSig(mid, \ldots) \\ \Gamma, msdecl \vdash Resolve\ \{\Gamma'(x) \mid x \in \bar{x}\} \Downarrow ms \\ \Gamma', ms \vdash \bar{x}!fpTypes(ms) \Downarrow \Gamma'', ms'\end{array}}{\Gamma' \vdash mid(\bar{e}) \Downarrow \Gamma''}\ \textsc{Call}$$

6.2 Abruptly Completing Statements

Another major difference between Java and the TIFI+ language is that in addition to loops and If-Then-Else, Java also has a number of other statements to redirect the control flow, namely `break`, `continue`, `throw` and `return`.

All of these statements complete abruptly, thereby attempting to transfer control to a specific *target* place, depending on the statement type.

- `break` (with a label) attempts to transfer control to the end of the enclosing (labeled) statement, which is then immediately completed. [9, Section 14.15]
- `continue` (with a label) attempts to transfer control to the enclosing (labeled) statement, which proceeds with the next loop iteration. The `continue` target can only be a loop statement. [9, Section 14.16]
- `return` completes abruptly with the reason "return" or "return V", V being the reference value pointing to the object to be returned. By doing this, it attempts to transfer control to the end of the method. [9, Section 14.17]
- `throw` completes abruptly with the reason "throw V", V being a reference value pointing to the exception object thrown by it at runtime. This attempts to transfer control either to the next surrounding matching `catch` clause in the same method, or to the next matching `catch` clause in one of the calling methods available at runtime. [9, Section 14.18]

[11] The receiver type is omitted.

```
void m(int i) {
foo:
  {
    try {
      System.out.println("foo");
      if (i==0) break foo;
      System.out.println("bar");
    } finally {
      if (i==0) return;
    }
    System.out.println("baz");
  }
  System.out.println("quux")
}
```

Listing 17: Abrupt completion example

Note that the term "attempting to transfer control" is explicitly chosen to designate that each of these attempts can be interrupted by an intermediate `finally` block. If such a block exists, control is first transferred to that block, then after its execution on to the actual target. If a `finally` block completes abruptly, the old attempt to transfer control is overruled.

Example 32. Consider the method in Listing 17. Executing this method with a parameter $i = 0$, will produce the output "`foo`" on the standard output. The individual steps leading to this are:

- We execute the program including `System.out.println("foo");`.
- The statement `break foo` completes abruptly, with the reason set to "break foo", attempting to immediately complete the execution of the surrounding block with the label "foo".
- Before reaching the block labeled "foo", the `finally` block needs to be executed.
- We reach the *return* statement, which completes abruptly with the reason "return". Consequently, the finally block is also completed abruptly with that reason, overruling the previous attempt to transfer control.
- The next nesting construct able to handle the reason "return" for abrupt completion is the method body itself, so the method execution is terminated immediately.

□

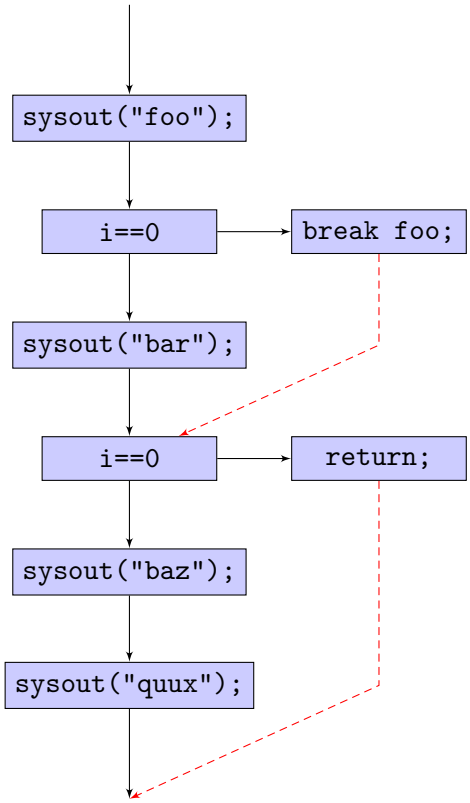

Figure 31: Method `m()` as control flow graph. Control flow caused by abruptly completing statements is depicted using dashed arrows.

Given these semantics, it is clear that this mode of transferring control is significantly different to the control flow of programs in the TIFI and TIFI+ languages, where statements are usually executed in syntactic order, sometimes entering a loop. For the abruptly completing statements, however, the outgoing edge in the control flow graph does not point to the syntactically following statement, but rather to one or multiple nodes of the control flow graph in different places.

For example, the control flow graph of method `m()` from Example 32 is depicted in Figure 31.

The inference algorithm presented in Section 5 is not directly able to handle those non-linear control flows. Luckily, there is a way to still track abruptly completing control flow: Looking at the four statements, we can observe that when the control flow is transferred to a point within the same method, this point is always syntactically placed *below* the origin, so that when the inference algorithm is in its last iteration looking at the origin statement, the target statement is still to be processed in the last iteration.

- A `break` statement attempts to jump to the next statement after the

loop it refers to. Obviously, this statement has not been looked at by the inference yet.

- A `continue` statement tries to jump to the end of the loop body of the loop it refers to. This statement may have been processed by the inference algorithm already, but the algorithm will still reach it at least once.
- A `return` statement attempts to jump to the method's end.
- Within the same method, a `throw` statement tries to jump to matching `catch` clauses it is nested in. When inferring through the `throw` statement, we will still not have looked at those `catch` clauses.

All of these attempts to transfer the control flow may be interrupted by a `finally` block. For these, the same argument as for `catch` clauses applies.

In order to correctly handle abruptly completing statements, we can thus change the algorithm in the following ways:

- Instead of inferring type environments $\Gamma \in \mathit{TypeEnv}$, we infer triples $(\Gamma, \mathcal{P}, a) \in \mathit{TypeEnv} \times \mathit{PendingJumps} \times \{\mathit{True}, \mathit{False}\}$.

  a is the type environment's *alive flag*. It is needed to account for the fact that abruptly completing statements have no leaving control flow to the following statement, although the inference algorithm proceeds in that direction.

  $\mathcal{P}$ is an (initially empty) set of *pending jumps*. A pending jump describes an outstanding unhandled jump which was seen earlier in the inference. If it is part of a set of pending jumps in an actual inference, we know that the jump target has not yet been reached in the inference. If it is reached, the type environment is merged back into the inferred environment.

  Pending jumps are modeled as tuples of (1) a reason for abrupt completion and (2) the *origin type environment* active when a statement was abruptly completed. For example, $\mathcal{P} := \{(\mathit{break}, \Gamma_1), (\mathit{break}, \Gamma_2)\}$ may be part of a triple inside a loop body which was inferred on a path with two previous break statements in that loop, one created when the type environment was Γ_1, one created when it was Γ_2.

- Whenever inferring over one of the four aforementioned statements, with an input of $(\Gamma, \mathcal{P}, \mathit{True})$, the output $(\Gamma', \mathcal{P}', a')$ is calculated as

$$\begin{aligned} \mathcal{P}' &:= \mathcal{P} \cup \{(r, \Gamma)\} \\ a' &:= \mathit{False} \end{aligned}$$

where r is the current statement's reason for its abrupt completion. Γ' is not alive, thus irrelevant.

By adding an entry to the map of pending jumps, the algorithm remembers the non-standard control flow (red arrows in Figure 31). By flagging the output environment as not alive, it remembers that the control flow cannot come from this path[12].

- At all potential jump targets, find matching jump reasons in the map of pending jumps, remove them and merge their origin type environments with the current type environment.

  For example, after a loop, the inference rule for loops removes all pending jumps with the reason "break" and "break *ID*" (*ID* being the loop's identifier), and merges their annotated origin type environments with the type environment inferred by the loop's fix point iteration.

- In order to use the triples $(\Gamma, \mathcal{P}, a)$ instead of type environments, we need to lift the definition of merge and check-implication to those triples.

  This is done in a way so that the attached pending jumps are merged, whereas the primary type environments are only used as long as they are flagged as alive, that is,

  $$(\Gamma_1, \mathcal{P}_1, a_1) \sqcup (\Gamma_2, \mathcal{P}_2, a_2) := (\Gamma_\star, \mathcal{P}_1 \cup \mathcal{P}_2, a_\star)$$

  where

  $$\Gamma_\star := \begin{cases} \Gamma_1 \sqcup \Gamma_2 & \text{if } a_1 \wedge a_2 \\ \Gamma_1 & \text{if } a_1 \wedge \neg a_2 \\ \Gamma_2 & \text{if } \neg a_1 \wedge a_2 \\ \text{don't care} & \text{if } \neg a_1 \wedge \neg a_2 \end{cases}$$

  $$a_\star := a_1 \vee a_2$$

  Check-implication is defined accordingly, so that the merged result of two triples is check-implied by both inputs to merge:

  $$(\Gamma_1, \mathcal{P}_1, a_1) \overset{c}{\Rightarrow} (\Gamma_2, \mathcal{P}_2, a_2)$$

  holds iff the following conditions are true:

  - $\mathcal{P}_1 \subseteq \mathcal{P}_2$
  - a_1 implies (a_2 and $\Gamma_1 \overset{c}{\Rightarrow} \Gamma_2$)

[12] Java already prevents users from putting statements right after abruptly completing statements. The alive flag is important however, when the result of such a statement is merged with another triple.

6.3 Method Return Types

In our Java subset, methods cannot directly return values. In Java, on the other hand, it is allowed.

In order to type check Java programs, we change the type inference / type checking algorithm in the following ways:

- Method signatures now include return types. Like the other types contained in the signature, this part of the signature can be parameterized using qualifier polymorphism.
- Method invocations are expressions now, not statements.

  The inference rule is therefore changed to set the special variable $\mathcal{E}$ to the signature's return type. When instantiating the qualifier polymorphic method signature, the return type needs to be instantiated along with the other parameters.
- The method body inference is responsible for ensuring that the method can only return values of the right type. This is done in the following way:
  - Before inferring through the method, we add a special variable $\mathbb{R}$ to Γ, tracking the methods return type along the method's execution. Recall that if the return type is a Fresh type, it not only changes by committing, but also by assigning local variables, which makes it impossible to identify types at different times in execution if it is not tracked by a special variable.
  - When a return statement is found during the inference, we additionally introduce a special variable $\mathcal{S}$, so that $\Gamma(\mathcal{S})$ is the type of the returned expression.
  - Just before the method inference is finished, we ensure that $\Gamma(\mathcal{S}) <: \Gamma(\mathcal{R})$ by invoking the type compatibility rule

$$\Gamma_{old} \vdash \mathcal{S}!\Gamma(\mathcal{R}) \Downarrow \Gamma$$

6.4 Method Overriding

When overriding methods, we need to make sure that the overriding method can be called in all places where the overridden method can be called. This compatibility can be ensured on the level of method signatures, without looking at method bodies. For overriding to be allowed, the Java Language Specification requires the overriding method's signature to be a subsignature of the overridden method's signature ([9, Section 8.4.8.1]).

In order to preserve soundness, we need to extend the notion of a subsignature ([9, Section 8.4.2]). In addition of the requirement that the two

signatures are equal (or one being the other's erasure, for backwards compatibility with Java 1.4), we require the following properties for the annotated immutability qualifiers:

For non-qualifier polymorphic signatures ms_1, ms_2, ms_1 is a subsignature of ms_2 if additionally to the usual JLS-definition:

- Formal parameters are contravariant: The type of a formal parameter in ms_2 must be a subtype of the corresponding type in ms_1.
- The receiver type needs to be contravariant: The receiver type in ms_2 must be a subtype of ms_1's receiver type.
- The result type needs to be covariant: The result type in ms_1 must be a subtype of ms_2's result type.

This already defines subsignatures on non-qualifier polymorphic signatures. For qualifier polymorphic signatures, it is required that the places where formal immutability type parameters occur are the same in both ms_1 and ms_2. In case of multiple formal qualifier parameters used in the same method signature, different formal qualifier parameters need to be used in the same places in ms_1 as in ms_2.

- In places annotated using Rd, $RdWr$ and Any, the same rules as above are applied.
- In places annotated using an formal qualifier parameter, their bounds must equal in ms_1 and ms_2. Furthermore, if two places in ms_1 use different formal qualifier parameters, the corresponding places in ms_2 must also use different formal qualifier parameters and vice versa.

This strict requirement for formal qualifier parameters is a design decision to simplify the overriding rules. Future type systems based on TIFI+ may choose to relax this rule.

Example 33 (Allowed scenarios for method overriding.)**.** In Listing 18, all method overridings are allowed.

`m1()` demonstrates co- and contravariance for non-qualifier polymorphic methods: Parameters and receiver are contravariant, the return type is covariant.

`m2()` demonstrates overriding of qualifier polymorphic signatures.

`m3()` shows that it is not necessary to use the same annotations for formal type parameters with equal meaning. It is sufficient that distinct formal type parameters in one signature are distinct in the same places in the other signature.

□

```
interface A {
  @Any C m1(@Rd C a) @Any;
  @PolyWriteable C m2(@PolyQual C a) @PolyWriteable;
  void m3(@PolyQual1 C a, @PolyQual2 C b);
}
interface C extends A {
  @Rd C m1(@Any C a) @Any;
  @PolyWriteable C m2(@PolyQual C a) @PolyWriteable;
  void m3(@PolyQual2 C b, @PolyQual1 C a);
}
```

Listing 18: Allowed scenarios for method overriding

```
interface A {
  void m1(@Any C a);
  @Rd C m2();
  void m3(@PolyQual C a, @PolyQual C b);
  void m4(@PolyQual1 C a, @PolyQual2 C b);
}
interface C extends A {
  void m1(@Rd C a);  // error: qualifier not contravariant.
  @Any C m2();       // error: result type not covariant.
  void m3(@PolyQual1 C a, @PolyQual2 C b);  // error
  void m4(@PolyQual C a, @PolyQual C b);    // error
}
```

Listing 19: Disallowed scenarios for method overriding

Example 34 (Disallowed scenarios for method overriding.)**.** In Listing 19, all method overridings lead to compile time errors.

`m1()` and `m2()` are invalid usages of co- and contravariance in parameters and return types.

The method `m3()` shows that it is not allowed for an overriding method to expect different Fresh types where the overridden method expected the same. The reverse, as demonstrated by `m4()`, is also forbidden. Future implementations may choose to allow the `m3()` case.

□

6.5 Constructors

Unlike methods, constructors are not allowed to return values. Instead, only the type of their parameters and of the implicit `this` argument have relevance.

Furthermore, the implicit argument `this` has special semantics for constructors: It is clear that when an object is created, it will initially be of a Fresh type, so that its instance variables can be initialized. Therefore, in the initial type environment for checking constructor bodies Γ_{in}, $\Gamma_{in}(\texttt{this})$'s type will always be set to a Fresh type.

It is a design question whether constructors should be able to commit the freshly constructed object to one of the initialized types, that is the constructor completely initializes the constructed object. If they do this, an object can for example be committed to *Rd* within its own constructor, which ensures that all instances created using that constructor are immutable. An obvious advantage is that this allows to ensure in the classes' constructor that all instances created through the constructor are immutable.

However, at the same time, allowing for those constructors also introduces additional complexity for both type system implementers and type system users. Consider the `Student` class in Listing 20:

```
class Person {
  Person() @Rd { ... }
}
class Student extends Person {
  University uni;
  Student(@Rd University uni) @Rd {
    // implicit call of Person() constructor.
    // Receiver object this is now Rd.
    this.uni = uni;  // Error.
  }
}
```

Listing 20: The `Student` class. (Note: This is not legal TIFI+ code.)

Note that the subclasses' constructor implicitly calls the superclass constructor. However, if the superclass constructor already commits the initialized object to *Rd*, the subclass constructor is not allowed to set its fields any more.

Allowing constructors themselves to finish the receiver object's initialization phase by committing it to *Rd* or *RdWr* makes subclassing using these constructors unnecessarily hard. Especially, for both implicit and explicit calls to superclass constructors, the type inference algorithm needs to be aware of changes to the receiver object's type, which goes against the principle of method's not modifying caller-visible types (Section 2.9.3).

Because of these design considerations, our adaption to Java does not allow constructors to commit the receiver's type to one of the initialized types or merge it with an externally visible type. A constructors possible commit behavior is then effectively the same as that of a method whose receiver type is annotated using a *@PolyWriteable* annotation. Consequently, after the execution of a constructor, every object is still in its initialization phase (Fresh).

Possible future solution

For class immutability, future versions of the type system may choose to introduce a class annotation *@ClassImmutable* (or similar), which is inherited to subclasses, and which, if present, instructs the type inference algorithm to infer a commit to *Rd* directly after each matching class instance creation expression ([9, Section 15.9]).

This way, it can be ensured that the objects of these classes stay Fresh during the execution of their constructors, but are committed to an initialized type directly when these constructor executions are finished.

6.6 Loop Statements

Unlike the simple TIFI+ language, Java has not only one, but four different kinds of loop statements:

- `while` statements ([9, Section 14.12]) of the form

  `while` (*Expression*) *Statement*

- `do` statements ([9, Section 14.13]) of the form

  `do` *Statement* `while` (*Expression*) ;

- basic `for` statements ([9, Section 14.14.1]) of the form

  `for` ($\mathit{ForInit}_{opt}$; $\mathit{Expression}_{opt}$; $\mathit{ForUpdate}_{opt}$) *Statement*

- enhanced `for` statements ([9, Section 14.14.2]) of the form

  `for` ($\mathit{VariableModifiers}_{opt}$ *Type Identifier*: *Expression*) *Statement*

In Section 5.3.3, we have already given the type inference algorithm for the while loop (Figure 26). For `do`- and `for`-loops, the algorithm can be given in a similar fashion.

6.6.1 `do` Loops

The fix point rule in Section 5.3.3 assumes that the loop body is executed at least once. This being the major difference between the `while` and the `do` loop, the inference rule for `do` can be formulated even more concise than the `while` rule:

$$\frac{\Gamma_{in}, \Gamma_{in}, \Gamma_{in} \vdash \mathit{fpiter}\ \bar{s} \Downarrow \Gamma_{out}}{\Gamma_{in} \vdash \texttt{do}\ \bar{s}\ \texttt{while}\ x \Downarrow \Gamma_{out}}\ \textsc{Do-Infer}$$

6.6.2 `for` Loops

In order to find an inference rule for the `for` rule, we can informally transform the generic form of a `for` statement to an equivalent loop using `while`.

A `for` statement of the form

$$\texttt{for}\ (\ s_{init};\ e_{cond};\ e_{update}\ )\ s$$

can be transformed to

```
{
  s_init;
  while (e_cond) {
    s;
    e_update;
  }
}
```

We can therefore infer through for loops using the rule

$$\frac{\Gamma_{in} \vdash s_{init} \Downarrow \Gamma' \qquad \Gamma' \vdash \texttt{while}\ e_{cond}\ (s @ (e_{update} :: \mathit{nil})) \Downarrow \Gamma'' \qquad \Gamma_{out} = \mathit{restrict}(\Gamma'', \mathrm{dom}(\Gamma_{in}))}{\Gamma_{in} \vdash \texttt{for}\ (s_{init}; e_{cond}; e_{update})\ s \Downarrow \Gamma_{out}}\ \textsc{For-Infer}$$

where the @ operator denotes list concatenation.

The actual implementation works on the program's actual abstract syntax tree directly, without creating parts of the intermediate representation as a `while` loop in memory. This variant of the algorithm can easily be deduced by inlining the *while* inference rule into the (For-Infer) rule.

6.6.3 Enhanced `for` Loops

Enhanced `for` loops iterate over arrays or objects implementing the `Iterable` interface. Unfortunately, TIFI+ currently has no support for generic classes and type parameters ([9, Section 8.1.2]), which means that it is not able to distinguish between iterables iterating over mutable objects and those iterating over immutable objects.

As it is impossible to give the proper immutability type for the objects iterated over, enhanced for loops are currently not supported.

6.7 Generic Classes and Type Parameters

Generic classes and type parameters, as described in [9, Section 8.1.2] are currently not supported by TIFI+.

The most common use case for type parameters are types for collections: Instead of plain types like `List`, it is common practice in Java 1.5 to use a type parameter to further specify the type of the contained objects: `List<Person>`.

In a similar fashion, it would be desirable for TIFI+ to specify the immutability type of the contained objects: Currently, the type of a list reference variable only describes the mutability of the references list itself, e.g. *Rd* for lists whose list structure cannot be changed.

Using type systems like IGJ [24], it is possible to also describe the immutability type of the contained objects. A possible way to describe a list reference, in TIFI+-like notation, could then for example be *RdWr*<*Rd*>(a mutable list of immutable objects) or *Rd*<*RdWr*>(an immutable list of mutable objects).

Considering the pervasiveness of the usage of generic classes (especially collections) in Java 1.5, the lack of support for it in TIFI+ is certainly a major shortcoming. Implementing it, however, was likely to be very time consuming, so that we chose to focus on more interesting aspects of TIFI+ and stay closer to the (non-generic) TIFI type system ([10]).

7 Implementation

This chapter describes the implementation of the TIFI+ type system for Java.

In this chapter, we assume familiarity with the design patterns Visitor, Abstract Factory, Factory Method and Singleton from the classic Design Patterns book [8]. Furthermore, a basic understanding of compilers – especially working with Abstract Syntax Trees – is helpful.

7.1 Overview

The TIFI+ type checker is implemented using the Checker framework [15], a framework for implementing pluggable type systems for the Java language.

A type checker written using the checkers framework is loaded as a plugin into the JDK's `javac` compiler, and implicitly executed for each compilation unit on compile time.

After installing the Checker framework as instructed in the Checker framework manual [4], the TIFI+ type checker can be run from the command line by executing `javac` with an additional command line switch defining the plugin's main class[13]:

```
javac -processor checkers.immutability.ImmutabilityChecker [...]
```

To annotate types, the Checker framework uses the *type qualifier annotation syntax* specified in JSR308 [7], which is going to be part of the Java 7 language. This extension to the Java language allows to annotate types using Java annotations. In our setting, the syntax is equivalent to what we used in the previous chapters, except that annotations all start with the At sign (@)[14].

7.2 Standard Architecture for Pluggable Type Checkers

The standard architecture for pluggable type checkers using the Checker framework consists of three main classes, usually named `FooChecker`, `FooAnnotatedTypeFactory` and `FooVisitor`, where `Foo` is replaced by the pluggable type system's name. The `FooChecker` class, which serves as a factory class [8], is also the interface for the compiler and, when used by the compiler implicitly creates instances of other classes. Depending on the type system's name, these instances are dynamically looked up at runtime, using Java's reflection mechanism.

These basic parts of a pluggable type checker implementation, which are depicted in Figure 32, are explained in this section. A more detailed discussion can be found in the Checker framework's manual [4].

[13] Alternatively, type checkers may also be loaded using auto discovery ([4, Section 2.2.1])

[14] For backwards compatibility with Java 5, it is also allowed to put type qualifier an-

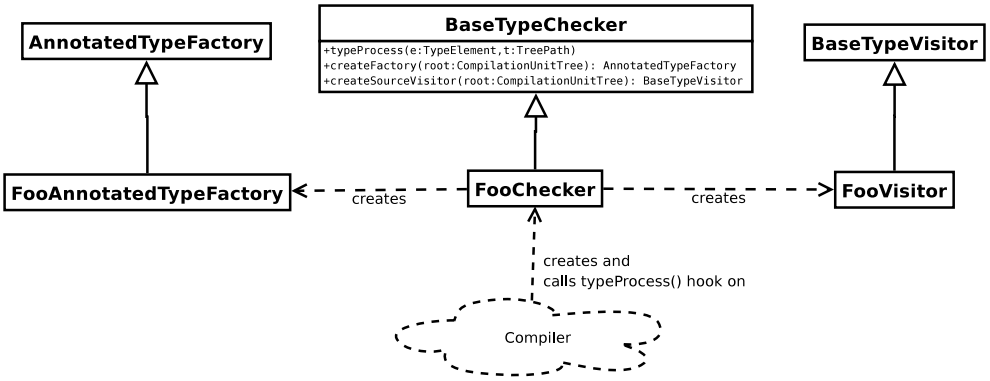

Figure 32: Basic structure and instantiation of a pluggable type checker.

7.2.1 Checker Class

A type checker's main class and entry point for the compiler is the *Checker class*, in our example `FooChecker`. Its main purposes are:

- Providing an entry point for the compiler
- Creating type checker infrastructure classes (Factory method pattern)
- Providing a means to report errors for the type checker classes, using the `report()` method

The compiler's active interaction with the checker class is described in the `AbstractTypeProcessor` classes' class documentation, from which the checker class is subclassed. After a short initialization phase, in which the checker class is instantiated, initialized and asked for the annotations it is responsible for, the compiler calls the `typeProcess(TypeElement element, TreePath tree)` method for each class definition in the source code in which such a type annotation appears. At the time when this happens, the class is guaranteed to have passed the standard Java type checking, and the program elements referenced from it have been resolved[15]. When all such classes have been visited, the compiler signals this by calling the `typeProcessingOver()` method.

In its role as a factory, the checker class creates instances of the classes `AnnotatedTypeFactory`, `BaseTypeVisitor` and instances representing the qualifier hierarchy. By default, all of this is done using reflection and class annotations, but it can easily be overridden in subclasses.

---

notations in comments.

[15]This is documented in the class documentation for the `com.sun.source.util.AbstractTypeProcessor` class, which is a superclass of `BaseTypeChecker`.

7.2.2 Type Hierarchy

Type qualifier annotations and their relationship in the type hierarchy are intended to be declaratively defined using annotations. For example, the *Any* annotation in TIFI+ is defined as

```
@TypeQualifier
@SubtypeOf({Qual.class})
public @interface Any {}
```

By annotating the checker class using a `@TypeQualifiers` annotation, the checker can now dynamically create the type hierarchy from the references to these classes and their annotations.

```
@TypeQualifiers({Qual.class, Any.class, /* etc */})
```

The type hierarchy is also represented as instances of the QualHierarchy and TypeHierarchy classes. More advanced customizations can be achieved by subclassing those and overriding the checker classes' factory methods.

7.2.3 Type Factory

Using the type hierarchy and its knowledge of the annotated types in the source code, the checker class creates an instance whose class inherits from the `AnnotatedTypeFactory` class. The type factory's purpose is to map AST nodes to the types annotated to them (instances of `AnnotatedTypeMirror`). Each of those `AnnotatedTypeMirror` instances describes a Java type and the annotations attached to it.

The annotations on an AST node can either be written out explicitly by the programmer, or they are automatically inferred by the type checker. A simple way of automatic inference is to give a default annotation for a specific kind of AST node. In the TIFI+ implementation for example, leaving method parameters unannotated in a program's source code is equivalent to annotating it with the *RdWr* type.

Apart from simple default annotations, the type factory may also return the types and annotations inferred by a flow-sensitive inference algorithm. The default implementation of the `AnnotatedTypeFactory`, `BasicAnnotatedTypeFactory`, implicitly runs a data flow analysis and uses its output for implicit type annotations (Section 7.2.5).

7.2.4 Visitor Class

Finally, the Checker class creates an instance of `BaseTypeVisitor` for each class definition found in the AST, which implements the type rules. This instance is then told to visit that class and report typing errors. The default

implementation already traverses the abstract syntax tree in depth-first order, so that the implementations for individual type systems only need to override the `visit` methods for special kinds of statements.

7.2.5 Support for Data Flow Analysis

The Checker framework also supports inferring types using data flow analysis, which is used by the Nullness type system distributed with the framework.

The data flow analysis algorithm itself is implemented in the Checker framework's `Flow` class, which is intended to be subclassed for adaption to specific type systems. The Flow class is a visitor class, which traverses the abstract syntax tree in a similar way as the TIFI+ inference algorithm does. Running the analysis produces an unambiguously defined mapping from AST nodes to annotated types.

The basic idea of how the data flow analysis integrates into the existing architecture is that its use is hidden behind the `AnnotatedTypeFactory` object. When constructing its type factory, the Nullness checker also instantiates its `Flow` subclass, `NullnessFlow`, and directly runs the data flow analysis. Then, whenever the `AnnotatedTypeFactory` is asked for the type of an AST node, it looks the type up in the `Flow` instance[16] using the `Flow` classes' `test(Tree t)` method.

7.2.6 Access to the Abstract Syntax Tree

The abstract syntax tree is passed to plugins using Sun's Compiler Tree API [13]. Each AST node implements the `com.sun.source.tree.Tree` interface, often through one of its subinterfaces.

Additionally, using the Checker framework's utility classes, it is possible to access the more abstract elements represented by an AST, like classes, interfaces, methods and types, using the mirror-based[17] API specified in JSR 269 [19]. The Checker framework provides much of the most commonly used functionality of this API using the static helper methods in `checkers.util.TreeUtils` and `checkers.util.TypeUtils`.

The class `checkers.types.AnnotatedTypeMirror` and its subclasses are based on the same design as JSR 269's `java.lang.model.type.TypeMirror` interface class. The type factory (`checkers.types.AnnotatedTypeFactory`) provides a rich set of factory methods to map program elements and different kinds of AST nodes to annotated types.

[16] It is interesting to note that the data flow analysis algorithm itself also uses the type factory to find the types of AST nodes.

[17] [6] gives an introduction to Mirror-based APIs for reflection.

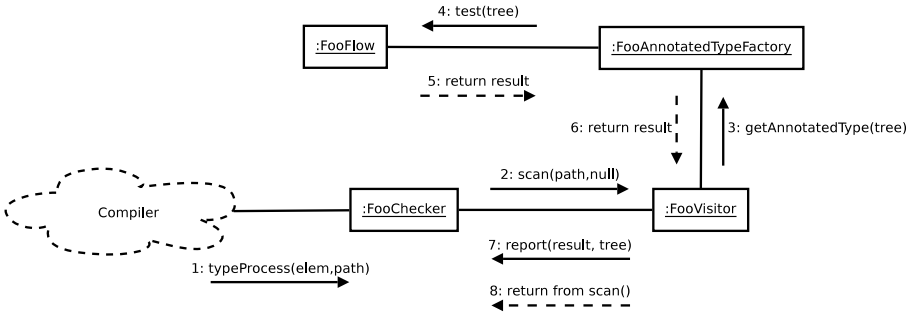

Figure 33: Typical execution of a type checker

7.2.7 Typical Type Checker Execution

The collaboration diagram in Figure 33 depicts the most relevant high-level object interactions in a type checkers typical execution. We assume that this type checker also uses data flow analysis, in the same way as the Nullness checker does.

The picture abstracts from the creation of all classes and from the `FooVisitor` instance's traversal of the abstract syntax tree. The compiler, which we do not need to understand further than that, is again represented as a cloud.

1. By calling the `typeProcess` method, the compiler signals to the checker that the class pointed to by the two parameters has been type checked using the standard Java type system and that all its used references to other program elements are resolved.

2. The checker instantiates the visitor and the type factory (which implicitly instantiates the flow class). The checker then asks the `FooVisitor` instance to search the method for typing errors in the classes' AST.

3. The visitor traverses the AST. When it finds an expression whose type is relevant for the program's soundness, it asks the type factory for this expression's type using the `getAnnotatedType()` method.

4. The annotated type factory delegates finding an expression's type annotations to the `Flow` instance by calling the `test()` method.

5. The flow instance has already run the data flow analysis directly after it was instantiated. When it did that, it saved all inferred types in a map, where they can now be looked up. It returns the looked up type to the caller.

6. The type factory, having retrieved the expression's type from the `Flow` instance, returns this type to the calling `FooVisitor` instance.

7. The visitor checks whether all types have been used appropriately. In this case, the checked program turned out to have typing errors, so the visitor calls the `report()` method on the checker instance. This lets the compiler print out an error and later cancel compilation.

8. Finally, the visitor returns from visiting the source code. In a more realistic scenario, a visitor would usually proceed checking the AST for conformance with the type conditions and try to find even more relevant errors in other places.

7.3 Differences to TIFI+

As we have seen in Section 5, the TIFI+ inference algorithm tries many different commit strategies when inferring through a method, in order to find one that allows to execute any path through the method. In the inference algorithm, the type of an AST node therefore differs depending on the chosen commit strategy.

For example, consider a method containing the variable identifier expression x ([9, Section 6.5.6.1]). Given that the variable x is of a Fresh type earlier in the method, it depends on the commits done earlier which type the expression has when reaching it in the inference.

On the other hand, the Checker framework makes the assumption that an AST node's type (as returned by the `AnnotatedTypeFactory`) is always the same, after it has been inferred. Using this assumption, existing type checkers can divide their work into the two passes of type inference and actual checking of adherence to the type conditions.

Type factory idea not usable for TIFI+

It is clear that the *two-pass approach is not a suitable implementation strategy* for the TIFI+ type checker. The types of many AST nodes within method bodies depend on the commit strategy we're currently looking at, so *it is unclear which type is to be returned by the type factory.* Furthermore, it is unclear which parts of the Checker framework (or its future versions) use the assumption that the type factory always returns the same types for the same inputs.

Type factory is only used for AST nodes outside method bodies

It is therefore a design decision in the TIFI+ implementation *to only use the annotated type factory for AST nodes outside method bodies*, that is, for method signatures and field annotations. With this decision, we stick to the basic design principle to program to an interface not an implementation ([8, Chapter 1]).

Unfortunately, this decision has the drawback that the TIFI+ type checker may not use any of the features relying on using the type factory for AST nodes within method bodies. This especially stops us from using the Checker framework's built-in support for generic methods.

7.4 TIFI+ Implementation Overview

The TIFI+ implementation follows the general basic design outlined in Section 7.2. However, as the TIFI+ type factory is only used for AST nodes outside method bodies, it is sufficient for the `ImmutabilityChecker` class to create an instance of the plain, unspecialized `AnnotatedTypeFactory` class for it.

The TIFI+ implementation has two visitor classes instead of just one:

- `ImmutabilityInferenceVisitor`: This class implements the type inference / type checking algorithm as described in Section 5. Its entry point is the method

  An instance of the class can run the type inference / type checking algorithm on a method body, with the given type environment as starting point. It is also responsible to check that the method fulfills the guarantees imposed by its method signature, that is, the return value has a suitable type and the method does not modify caller-visible Fresh types.

- `ImmutabilityMethodVisitor`: This class takes the role of the visitor class in the standard type checker design. When an instance is asked to type check a class definition, it traverses it to find all method occurrences. Each instance of `ImmutabilityMethodVisitor` holds a reference to an `ImmutabilityInferenceVisitor` instance, which it uses to run the type inference / type checking algorithm on method bodies.

The main difference between these two classes is their responsibility: While the `ImmutabilityInferenceVisitor` class is responsible for running the actual type checking / type inference algorithm, `ImmutabilityMethodVisitor` takes responsibility for checking constraints higher in the AST, namely correct method overriding, correct field annotations and invoking the inference algorithm with the right input environments.

An overview class diagram is given in Figure 34.

7.4.1 High-Level Collaboration Overview

The TIFI+ type checker implementation's main objects collaborate as depicted in Figure 35. This behavior is similar to the one discussed in Section 7.2.5. Unlike the generic design however, the visitor class `ImmutabilityMethodVisitor` is not alone responsible for type checking, but instead mostly delegates this task to the `ImmutabilityInferenceVisitor`. Both of these classes are able to report errors to the compiler over the checker object.

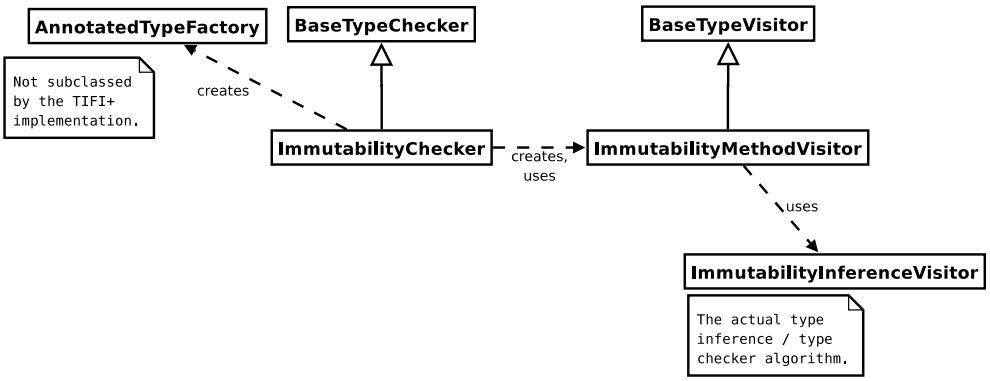

Figure 34: Structural overview over the TIFI+ implementation's main classes

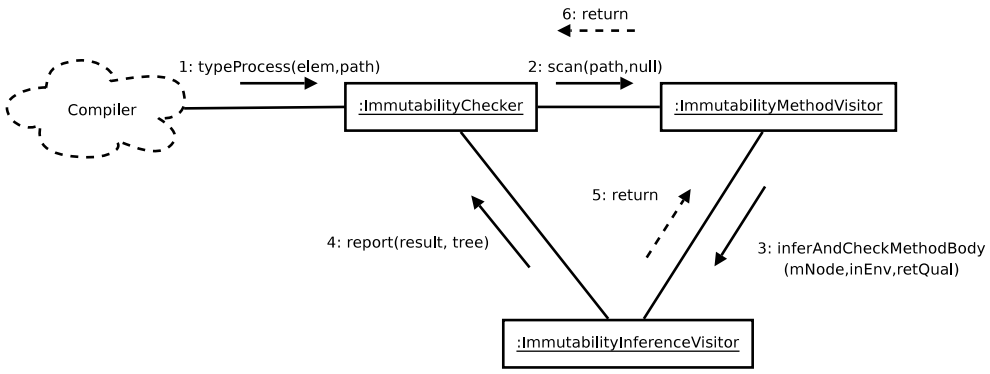

Figure 35: High-level collaboration between the TIFI+ implementation's main objects

7.5 Data Structures

The main classes making up TIFI+'s type environment representation are Qual, Variable, TypeEnv and TypeEnvSet, which will be described herein.

Class	Purpose
`Qual`	representing access qualifiers / immutability types
`Variable`	representing methods' local variables
`TypeEnv`	mappings from variables (`Variable`) to types (`Qual`), with attached pending jumps
`TypeEnvSet`	sets of type environments

All these classes reside in the package `checkers.immutability.data`.

7.5.1 `Qual` Class

Each `Qual` instance represents a type in the TIFI+ type system. There is one instance representing each the *Rd*, *RdWr* and *Any* type, and multiple instances representing the Fresh types. Instances of `Qual` are immutable.

To save memory, there is only one instance for the simple types *Rd*, *RdWr* and *Any*. The `Qual` class thus provides static singleton methods `rd()`, `rdwr()` and `any()` returning the respective instance.

A similar approach is taken for Fresh instances, which are also shared between type environments. The static method `fresh(Set<Variable> vars)` creates the `Qual` instance representing F_s(`vars`). Similarly, the static method `emptyFresh()` returns the instance representing $F_s(\emptyset)$

The `Qual` class offers a number of methods to test for types' properties and relationships between types:

Method	Description
`Qual minimalCommonSupertype(Qual other)`	the minimal type that is a supertype of both the receiver and other (if it exists, else null)
`boolean subtypeOf(Qual q)`	true iff the receiver is a subtype of q
`boolean isCommitted()`	true iff the receiver is a committed type (*Rd*, *RdWr* or *Any*)
`boolean isFresh()`	true iff the receiver is a Fresh type
`boolean isWriteable()`	true iff the receiver is *RdWr* or a Fresh type

Methods involving multiple qualifiers assume that these qualifiers may occur in the same type environment, e.g. it is not allowed to ask for the minimal common supertype of $F_s(\{x, y\})$ and $F_s(\{y, z\})$.

The current implementation of the qualifier hierarchy does not model the bottom type $NullT$.

Example 35 (Examples from the unit tests)**.**

```
assertTrue(q.subtypeOf(q));  // for all q instanceof Qual
assertTrue(Qual.rd().isCommited());
assertEquals(Qual.any(),
        Qual.rdwr().minimalCommonSupertype(Qual.rd()));
```

□

The methods

```
public Qual withVariable(Variable v);
public Qual withoutVariable(Variable v);
```

allow to derive Fresh types from each other, which is used when calculating type environments based on other type environments. For example, given the Fresh type $F_s(\{x, y\})$, the method `withVariable()` called with an argument variable z yields the Fresh type $F_s(\{x, y, z\})$.

The `withoutVariable()` method does the opposite. When not called on a Fresh type, both methods return the unmodified receiver.

Example 36 (Examples from the unit tests)**.** Note: The function `F()` takes a variable list of `Variable` instances as arguments, and constructs the appropriate Fresh type from them. `x_`, `y_`, `z_` are references to `Variable` instances.

```
assertEquals(F(x_, y_, z_), F(x_, y_).withVariable(z_));
assertEquals(F(x_), Qual.emptyFresh().withVariable(x_));
assertEquals(F(x_, y_), F(x_, y_, z_).withoutVariable(z_));
```

□

7.5.2 `Variable` Class

Objects implementing the Variable interface represent a method's local variables, including method parameters and `this`.

Variable instances are not supposed to be created directly by the user but instead by using the factory class `VariableMaker`, which offers the following methods to create immutable Variable instances:

- `public Variable makeThisVar();` returning the variable representing `this`.

- `public Variable makeLocalVar(VariableElement elem);` returning the variable instance representing the given variable element.
- `public Variable makeSpecialVar(String name);` returning a new special variable with the given base name. The created special variables are guaranteed not to be equal to any of the previously created variables.
- `public Variable makeNamedLocalVar(String name);` returning a named non-special variable. This is internally used by `makeThisVar()` and unit tests.

All method results except when creating special variables are cached. This way, when asking for the `Variable` instances for two equal `VariableElement` objects, the factory method will yield the same result both times.

Special variables are additional variables that can be used by the inference algorithm to track changes to Fresh types throughout different derived versions of a type environment.

7.5.3 `TypeEnv` Class

The TypeEnv class represents type environments with an annotated alive-flag and a set of pending jumps (Section 6.2). Again, instances of the TypeEnv class are immutable.

The TypeEnv classes' public methods can be roughly divided into six main categories:

Creation: The static method `TypeEnv.empty()` returns the empty type environment, which is alive and has no pending jumps. All other type environments are derived from it.

Merging and implication: The method `boolean implies(TypeEnv other)` can test for check-implication between the receiver `TypeEnv` instance and the `TypeEnv` instance referenced by `other`.

The merging method `TypeEnvSet merge(TypeEnv other)` merges two `TypeEnv` instances. Note that as the merge operation is ambiguous, the method returns a set of type environments.

To be formally correct, `TypeEnv` instances actually represent not just a type environment Γ, but actually the whole triple $(\Gamma, \mathcal{P}, a)$ as it is described in Section 6.2. While it is usually sufficient for an intuitive understanding to only consider the contained actual type environment, the implementations of the merge and check-implication operations actually need to be aware of pending jumps and the alive-flag (Section 6.2).

Modification: The type environment helper functions described in Section 4.10.1, $\Phi!\langle x \mapsto \ldots\rangle$, $\Phi\langle \mathit{delete}\ x\rangle$, and $\Phi\langle x \mapsto \ldots\rangle$, are directly supported by the methods

```
TypeEnv withNewVariable(Variable var, Qual q);
TypeEnv withVarType(Variable var, Qual q);
TypeEnv withoutVariable(Variable var);
```

In addition to these, `TypeEnv` also offers a number of convenience methods for conveniently adding or removing multiple variables.

Note that all of these methods do not modify the receiver type environment, but rather return a new `TypeEnv` instance with the desired properties. Like their corresponding functions in the formalization, the types (`Qual` instances) given as arguments are required to be types occurring in the receiver type environment. This is an overall requirement for all of the `TypeEnv` classes' publicly visible methods.

Type compatibility: The type compatibility methods used in the operational semantics and the inference algorithm have their counterparts in the methods

```
TypeEnv withTypeCompatibility(
    Qual use, Qual decl, Reporter reporter);
```

corresponding to the rules matching the general form

$$l, \Phi, h, X \vdash x!!q_s \Downarrow \Phi', h', X'$$

which may commit both $\Phi(x)$ and q_s, and

```
TypeEnv withTypeCompatibility2(
    Qual use, Qual decl, Reporter reporter);
```

corresponding to the rules matching the general form

$$l, \Phi, h, X \vdash x!q_s \Downarrow \Phi', h', X'$$

which may commit only $\Phi(x)$.

It is sufficient here to give $\Phi(x)$ as the `use` parameter. It can be easily verified by looking at the rules, that the variable x is only needed to do the appropriate commit in the dynamic view of the running system. In our case, this is not needed.

The `reporter` parameter is used for error reporting in case the type compatibility cannot be established.

Inspection: Basic inspection of type environments is supported by the following methods and a number of convenience methods built on top of them. `te` being the representation of the type environment Γ, x being a variable, this table gives the mapping between method invocations of these methods and expressions in terms of the formalization.

`te.variables()`	$\mathrm{dom}(\Gamma)$
`te.getType(x)`	$\Gamma(x)$
`te.hasVariable(x)`	true iff $x \in \mathrm{dom}(\Gamma)$

Pending Jumps: There are a number of methods to handle pending jumps. The methods `withAttachedBreak()` and `withCollectedBreaks()` are a specific example:

`withAttachedBreak()` returns a non-alive type environment, which has the original type environment attached to it as a pending jump with the reason "break".

`withCollectedBreaks()` collects all attached type environments with the reason "break" and merges them into the receiver type environment. The resulting type environment will not contain any pending jumps with the reason "break".

Merging an alive type environment into a non-alive type environment yields the alive type environment as a result. For example, `te` being a type environment without attached breaks, `te.withAttachedBreak().withCollectedBreaks()` yields an object equal to `te`[18].

To model the set of pending jumps, `TypeEnv` instances hold references to instances of the `PendingJumps` class in the same package, which again can only have immutable instances and which offers a number of specialized convenience methods to access the set of pending jumps.

7.5.4 `TypeEnvSet` Class

The `TypeEnvSet` class represents sets of type environments. Its main use is to model the set of possible result type environments after an (ambiguous) inference step.

The conceptual difference between the `TypeEnvSet` class and Java's standard `Set` class is that `TypeEnvSet` instances automatically reduce their contained elements to only the relevant result environments, which speeds up the inference.

[18]The objects are not identical is the sense of Java's `==` operator, but they have the same content and behave the same in all cases.

Formally speaking, after adding a set of type environments $\Gamma^{\star}$[19] to a `TypeEnvSet` instance, this instance will only contain the type environments in $\mathcal{R}(\Gamma^{\star})$, which is defined as

$$\mathcal{R}(\Gamma^{\star}) := \{\Gamma \in \Gamma^{\star} |\ \nexists \Gamma' \in \Gamma^{\star} : \Gamma \overset{c}{\Rightarrow} \Gamma'\}$$

Example 37. For example, consider

$$\Gamma^{\star} := \{\{x : Rd\}, \{x : RdWr\}, \{x : Any\}\}$$

It is clear that

$$\{x : Any\} \overset{c}{\Rightarrow} \{x : Rd\}$$
$$\{x : Any\} \overset{c}{\Rightarrow} \{x : RdWr\}$$

In other words, when a program type checks with $\{x : Any\}$, it will also type check with $\{x : Rd\}$ and $\{x : RdWr\}$.

Applying the reduction operation $\mathcal{R}$ yields

$$\mathcal{R}(\Gamma^{\star}) := \{\{x : Rd\}, \{x : RdWr\}\}$$

Applying $\mathcal{R}$ to $\Gamma^{\star}$ removes the type environment $\{x : Any\}$ from it.

The reason why this is done lies in the meaning of check-implication. As noted above, when a program type checks with $\{x : Any\}$, it will also type check with $\{x : Rd\}$ and $\{x : RdWr\}$, so by removing $\{x : Any\}$, we will not change the overall success status of running the inference on the remaining method.

Consider the two cases:

1. We would have found an inference path through the method using $\{x : Any\}$: Then, because of the check-implication, we can also find one using $\{x : Rd\}$ ($\{x : RdWr\}$). Omitting the $\{x : Any\}$ case gives the same results as not doing so.

2. We would *not* have found an inference path through the method using $\{x : Any\}$: Then, it is clearly sufficient to proceed using the remaining type environments only, in order to find all successful inference possibilities through the method. Again, we may proceed the inference without taking $\{x : Any\}$ into account.

□

Construction

The `TypeEnvSet` class, although intended to be used in an immutable way, supports in-place construction using the methods

[19] In fact, these are triples, but the argument is applicable to any data structure where check-implication can be defined on, so we will proceed using type environments in this section. The example can be easily rewritten for triples by replacing each occurrence of a type environment Γ with the triple $(\Gamma, \emptyset, \mathit{True})$

```
void unionInPlace(TypeEnvSet other);
void addInPlace(TypeEnv addedEnv);
```

When using these on a `TypeEnvSet` instance, the instance automatically cares about reducing the set of type environments on-the-fly.

Inspection

Apart from these methods, `TypeEnvSet` supports the usual ways of inspecting standard Java sets, such as the methods `isEmpty()`, `size()` and looping over its contents using the `Iterable<TypeEnv>` interface.

7.5.5 `ImmutabilityInferenceVisitor` Class

The `ImmutabilityInferenceVisitor` class, at the heart of the TIFI+ implementation, is responsible for executing the actual inference algorithm. Each of its `visit` methods is responsible for doing an inference step for one specific kind of AST node. The inference rules outlined in Chapter 5 and adapted to Java in Chapter 6 can be mapped to the `ImmutabilityInferenceVisitor`'s `visit` methods depending on the kind of the examined AST node.

Conceptually, each inference step done by a `visit` method can be seen as calculating a function from a tuple `Tree` × `TypeEnv`[20] to a set of type environments (a `TypeEnvSet` instance). To do the inference step over an AST node, the `ImmutabilityInferenceVisitor` provides the method

```
TypeEnvSet infer(Tree node, TypeEnv inputEnv)
```

Based on the kind of AST node given to it, the `infer()` method decides which `visit` method to call and returns the `visit` method's result.

For pragmatic reasons, the `ImmutabilityInferenceVisitor` class currently has two instance variables `TypeEnv input_` and `TypeEnvSet output_`, which are used by the `visit` methods to read and write their input and output type environments. Using these, it was possible to provide the visitor with a reasonable default behavior for yet-unimplemented kinds of AST nodes, by subclassing from the `com.sun.source.util.TreeScanner`[21] class. This allowed to run the inference algorithm even very early in its development, when only few `visit` methods were actually implemented. The `infer` method takes care of saving the visitor state with respect to these variables, and restores it after calling the desired `visit` method. The visitor's state change is thus transparent to the user when using `infer()`.

In the long run, when all relevant `visit` methods are implemented, future implementers may want to remove the `input_` and `output_` instance

[20] Recall that the `Tree` class, described in Section 7.2.6, represents an AST node in a Java program.

[21] The `TreeScanner` class implements all possible `visit` methods so that they recursively visit the child nodes of the AST node they are working on, which in many cases matches `ImmutabilityInferenceVisitor`'s tree traversal strategy.

variables and introduce equivalent parameters and return values to the `visit` methods.

Typical behavior of a `visit` method

A `visit` method's typical behavior is:

1. Before execution, `input_` is guaranteed to be set to the input type environment, while `output_` is `null`. The input must thus be taken from the `input_` instance variable.

2. Typically, a new empty `TypeEnvSet` instance is created to store the future result.

3. A number of output type environments are calculated, depending on the kind of AST node which is inferred over. Each of these type environments is added in-place to the `TypeEnvSet` instance.

4. Finally, the `TypeEnvSet` instance, now being fully initialized, is stored in `output_`, `input_` is set to `null` and the method returns.

Error reporting

Whenever on a path through the tree of possible commit sequences, we encounter a situation where the execution of the next statement would break existing immutability properties, the inference method noticing this emits an error.

When all such paths encountered an error before reaching the end of the currently checked method, the method is not type correct. Only then, the TIFI+ implementation prints these errors to the user.

To achieve this behavior, the `ImmutabilityInferenceVisitor` holds a reference to a `DelayedReporter` object. It is possible to report errors to this object at any time, but they will not be shown to the user unless, after all ambiguous paths have died, the `ImmutabilityInferenceVisitor` chooses to call the `flushReports()` method. In the other case, when a method turned out to be type correct, the `clear()` method is called instead, which clears the reported errors without writing them out to the user.

7.6 Supported Java Constructs

The current TIFI+ implementation supports the most common Java constructs, like assignments, method and constructor invocations, expressions for local variables, object members and identifiers, the control flow constructs `if`...`else`, `while`, `do`...`while` and `for`, as well as binary expressions (arithmetic and binary operations, string concatenation, equality, ...) and the ternary operator `?:`.

The pending jumps mechanism for abruptly completing statements has been implemented and tested for the `return` and `break` statements. Support for exception handling may be implemented in future versions.

Support for qualifier polymorphism is implemented both for checking method bodies and invoking methods, including multiple formal qualifier

parameters and their automatic inference for method invocations. Currently, we are still aware of some smaller issues in the automatic instantiation of method signatures, which will require further investigation for a complete implementation.

The TIFI+ test suite, which is bundled with the implementation, contains a collection of source code examples and unit tests for the supported Java constructs, including the examples listed in Chaper 3.

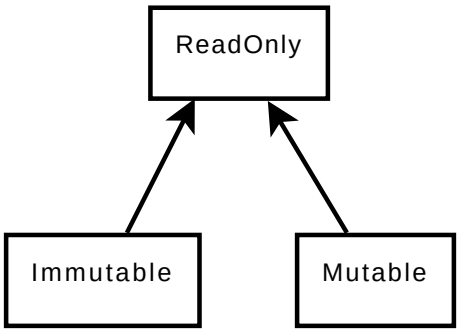

Figure 36: IGJ immutability type hierarchy

8 Related Work

Compared to other immutability type systems implemented on top of the Checker framework, TIFI+ has both advantages and disadvantages.

8.1 Immutability Generic Java (IGJ)

The IGJ type system presented by Zibin, Potarin, Ali, Artzi, Kiezun and Ernst ist very similar to TIFI and TIFI+ in that it has the same hierarchy of "initialized" immutability types, as depicted in Figure 36.

The mapping from IGJ types to TIFI+ types is as follows:

IGJ type	TIFI+ type
ReadOnly	*Any*
Immutable	*Rd*
Mutable	*RdWr*

Object and reference immutability

Just like in TIFI+, reference variables typed as *Immutable* and *Mutable* point to *objects*, which are mutable or immutable. These references not only disallow their use for modifying an object, but they also guarantee that the referenced object cannot be changed over alias references. Like TIFI+, IGJ is thus able to provide *object immutability* (*Immutable*) as well as *reference immutability* guarantees (*ReadOnly*).

8.1.1 Object Construction

Introducing object immutability to an immutability type system quickly raises the question of how to construct immutable objects: It is clear that the constructor must still be able to modify the object, but at the same time, it must be ensured that the constructor does not leak `@Mutable`-typed references to partially initialized objects.

In IGJ, this problem is solved by annotating the constructor using a special `@AssignsFields` annotation. It is then possible for the constructor

to assign the fields of the constructed object, but it is not possible to mutate the objects referenced by them.

Whenever the constructed object is passed out by an `@AssignsFields`-annotated constructor, it can only be given out as `@ReadOnly`. Consequently, whenever another part of the program has an alias to an object which is just being constructed, it has a `@ReadOnly` reference to it: It is only restricted in that it cannot modify the object, but there are no guarantees whether the referenced object may still be changed via other references or not.

This is a drawback of IGJ compared to TIFI+: In TIFI+, using qualifier polymorphism with Fresh types, it is easily possible to write helper methods to faciliate the construction of objects, while in IGJ, only the constructor itself is allowed to modify an object.

Furthermore, there are object graphs which can be built using TIFI+, but not using IGJ. For instance, consider building an object graph where two objects circularly reference each other using `@Immutable` references, as we constructed it using TIFI+ in Example 7. Using IGJ, these objects' instance variables can only be assigned by their constructors, and the constructors finish their execution at different times. However, only when a constructor finishes, the constructed object is first given out using an `@Immutable` reference (instead of a `@ReadOnly` reference). So, the first of the two objects to finish its constructor execution cannot possibly have gotten hold of an `@Immutable` reference to the other object until then. The argument can easily be expanded to bigger cyclic object graphs. Therefore, in IGJ, it is not possible to build reference cycles between objects where each reference has an `@Immutable` type.

8.2 Javari

The Javari type system [22] [21] provides only reference immutability, not object immutability.

Javari's hierarchy of immutability qualifiers consists of only the two qualifiers `@Mutable`, which is the default, and `@ReadOnly` (Figure 37), which are equivalent to TIFI+'s *Any* and *RdWr* types.

As in TIFI+, these qualifiers extend Java's types in an orthogonal way, so that for a class `C`, `@Mutable C` is a subtype of `@ReadOnly C`, and for `C` being a subclass of a class `A`, `@Mutable C` is a subtype of `@Mutable A` and `@ReadOnly C` is a subtype of `@ReadOnly A`.

In Javari, there is no need to specifically support object construction: At construction phase, objects are `@Mutable`. When an object of a type `@Mutable C` is not supposed to be changed any more, the references given out can be upcasted to `@ReadOnly C`.

Unlike with real object immutability, it is up to the programmer to ensure that there are no `@Mutable`-typed alias references to such objects. Javari cannot give the guarantee that objects referenced by `@ReadOnly`-

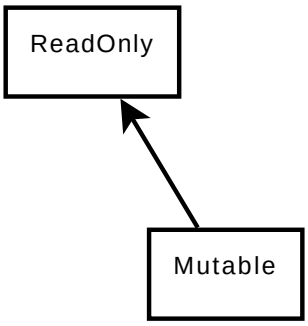

Figure 37: Javari qualifier hierarchy

typed variables do not change their abstract state. It only restricts the holders of these references, but cannot give them any guarantees.

Unlike in TIFI+, local variables in Javari cannot change their type and are annotated. The Javarifier [18] tool can infer Javari annotations to alleviate the task of adapting programs to Javari.

Being very well-integrated into the Checker framework, Javari supports most Java constructs including Java 1.5's Generics. In addition to those, there is an option to template methods over Javari qualifiers alone, using the `@romaybe` annotation, which works in the same way as TIFI's qualifier polymorphism.

8.3 Type Inference

Unlike TIFI+, both Javari and IGJ heavily rely on the user not only providing type annotations in method signatures, but also in their bodies. This requires an additional up-front investment for initial annotations and requires program maintainers to change annotations appropriately. For Javari, the separately available Javarifier [18] tool can be used to reduce these efforts. On the other hand, due to its programming model and its inference algorithm, TIFI+ can give error messages which are harder to understand than those of Javari and IGJ.

As the TIFI+ implementation is currently still not usable for real-world Java programs, it is hard to say which efforts are bigger in practice.

Avoiding type annotations for local variables is a natural choice for the TIFI+ type system: In TIFI+, an object's type always reflects its immutability life cycle phase. As these may change over time, even within the same method, a simple look to a variable's declaration is not sufficient to find out its immutability type in a specific use further down in the method. The same approach is also taken by Adam Warski's typestate checker [23], which is also implemented on top of the Checker framework.

9 Conclusion

This thesis provides a prototype implementation of a modified variant (TIFI+) of the immutability type system proposed by Haack and Poll [10] [11]. The type system is able to give guarantees on object and reference immutability, and provides the *Flexible Initialization* mechanism for object construction. The implementation is built on top of the Checker framework [15] and supports a large, although not complete, subset of Java's language constructs.

Furthermore, we have formalized a simple Java subset and informally shown the soundness of the type inference algorithm used in TIFI+.

TIFI+ shows that it is possible to implement TIFI-like systems on top of existing Java-like programming languages. We have also seen some larger code examples in Chapter 3 supporting the thesis that TIFI+ and Flexible Initialization are a useful way to guarantee the construction of immutable objects in a safe way.

During the course of this work, we have changed a number of aspects of the original TIFI formalism, which made the inference algorithm more flexible than proposed in [11], and also simplified some aspects of it.

Interesting topics for further study are error reporting and support for generic types. We expect that especially the latter will open the door for using TIFI+ with a significantly larger amount of real-world Java code.

The examples we have seen suggest that Flexible Initialization is in fact a very flexible approach for the safe construction of immutable objects. We believe that the TIFI+ implementation presented herein provides not only a good basis for future refinement, but can also serve as a vehicle for experimentation with object immutability type systems, which can be used as a stepping stone for further insights on this topic.

A Non-Completeness of the Original TIFI Inference Algorithm

The inference algorithm for inferring `newtoken` and `commit` statements presented in the original TIFI report [11] is unfortunately not complete. (It is not able to find the `newtoken` and `commit` statements for all type correct programs.)

This section assumes that the reader is familiar with the inference algorithm presented in [11].

Consider the following program:

```
// x:Fresh(n) C
any(x);
rd(x);
```

An obvious type-correct placement of a `commit` statement would be to insert `commit Fresh(n) as Rd` in front of the call to the `any()` method. This way, `x` is of type *Rd* before calling any of the methods. Since through subtyping, `x` is also of type *Any*, both methods can be called without further commits.

The type inference algorithm given in the paper splits up the program using the (Infer Seq) rule. As the following derivation tree for the expression `any(x)` shows, it infers the commit statement in the same place, but commiting to *Any* instead of *Rd*.

The original TIFI inference algorithm infers a commit to *Any* in front of the first method call. Consequently, there is no more way to call the `rd()` method using `c` as argument. It is not able to find the real solution of committing to *Rd* in front of the first method call.

In the TIFI report [11], it is shown that if the inference algorithm can infer the placement of `newtoken` and `commit` statements, the method is indeed type correct. It can be verified using the aforementioned example that the converse is unfortunately not true.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{true}}{\{n\} \vdash commit(Fresh(n), Any) \Downarrow (\{(n \to Any)\}, \{n\})\}}\,(\textsc{Commit})}{\{n\} \vdash Fresh(n) <: Any \Downarrow (\{(n \to Any)\}, \{n\}, \emptyset)}\,(\textsc{Subqual})}{\{n\} \vdash Fresh(n)\,C <: Any\,C \Downarrow (\{(n \to Any)\}, \{n\}, \emptyset)}\,(\textsc{Subtyping}) \qquad \dfrac{}{\{n\} \vdash () : ().() \Downarrow ((), \emptyset, \{n\}, \emptyset, \emptyset)}\,(\textsc{Match})}{\{n\} \vdash (x) : ().(Fresh(n)\,C) \Downarrow ((), \{(n \to Any)\}, \{n\}, \emptyset, \emptyset)}\,(\textsc{Match})}{\{n\}; \underbrace{\{x : Fresh(n)\,C\}}_{\text{type environment before call to any()}} \vdash unused = any(x) \Downarrow (\underbrace{\{x : Any\,C\}}_{\text{type environment after call to any()}}, \{(n \to Any)\}, \{n\}, \emptyset, \underbrace{\emptyset}_{\text{empty constraint set}})}\,(\textsc{Call})$$

In the actual language subset used in [11], the program would need to be written as shown in Listing 21.

```
C x;                    // x : Fresh(n_old) C
C y;                    // y : Fresh(m) C
C z;                    // z : Fresh(o) C
x = new C();            // x : Fresh(n) C
// Γ = {x : Fresh(n) C, y : Fresh(m) C, z : Fresh(o) C},
// δ = ∅,
// ts = {n_old, n, m, o}
y = any(x);             // enforces x : Any C, y : Any C
z = rd(x);              // enforces x : Rd C, z : Rd C
```

Listing 21: Counterexample

List of Figures

References

[1] Private e-mail conversation with Christian Haack. October 30 – November 11, 2010.

[2] Private telephone conversation with Christian Haack. November 11, 2010.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] Mahmood Ali, Telmo Correa, Michael D. Ernst, and Matthew M. Papi. *Checker Framework manual v1.0.6*, February 2010.

[5] Joshua Bloch. *Effective Java Programming Language Guide*. Sun Microsystems, Inc., Mountain View, CA, USA, 2001.

[6] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 331–344. ACM, 2004.

[7] Michael D. Ernst. *Type Annotations specification (JSR 308)*. http://types.cs.washington.edu/jsr308/specification/java-annotation-design.pdf, November 2009.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 06 2005.

[10] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 520–545, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. Technical Report ICIS–R09001, Radboud University Nijmegen, January 2009.

[12] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, fifth edition, December 2009.

[13] Sun Microsystems. Javac compiler tree API. `http://java.sun.com/javase/6/docs/jdk/api/javac/tree/`, Last accessed: March 25, 2010.

[14] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[15] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, New York, NY, USA, 2008. ACM.

[16] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[17] Prof. Dr. Arnd Poetzsch-Heffter. Advanced aspects of object-oriented programming (slides). `http://softech.informatik.uni-kl.de/Homepage/FASOOPSS09`, Last accessed March 27, 2010, 2009.

[18] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 616–641, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Sun Microsystems. Java Specification Request 269: Pluggable annotation processing API. `http://jcp.org/en/jsr/detail?id=269`, Last accessed: March 25, 2010.

[20] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby. The Pragmatic Programmer's Guide*. Pragmatic Programmers, 2004.

[21] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, Massachusetts Institute of Technology, August 2006.

[22] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. *SIGPLAN Not.*, 40(10):211–230, 2005.

[23] Adam Warski. Typestate checker homepage. `http://www.warski.org/typestate.html`, Last accessed: March 25, 2010.

[24] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kie—un, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 75–84, New York, NY, USA, 2007. ACM.